Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

o1js

Veridise Inc.

August 27, 2024

► **Prepared For:**

o1Labs
https://www.o1labs.org

► **Prepared By:**

Benjamin Sepanski
Sorawee Porncharoenwase
Alp Bassa
Daniel Dominguez

► **Contact Us:** contact@veridise.com

► **Version History:**

Aug. 27, 2024	V3 - Including missing fix PR and developer response
Aug. 21, 2024	V2 - Incorporated fix validations
Jun. 28, 2024	V1
Jun. 26, 2024	Initial Draft

Contents

1 Executive Summary

From Mar. 18, 2024 to Jun. 21, 2024, o1Labs engaged Veridise to review the security of their zero-knowledge circuit smart contract library o1js*. The review covered the TypeScript encoding of various common types and proof gadgets into ZK-circuits using o1Labs's Kimchi and Pickles† proof libraries, as well as the constraints for building ZkApps‡ with o1js.

Veridise conducted the assessment over 39 person-weeks, with 2 security analysts reviewing code over 13 weeks, and 2 additional analysts providing further reviews and engineering effort to aid with critical cryptographic regions of the code and to increase the level of automated testing. The review took place on commits `02f2ffb6-8dde2c3b` for the o1js repository and `e7ded4b6-3c68a0da` for the o1js-bindings repository. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual code review.

Project summary. The security assessment covered the o1js TypeScript library. o1js is a library for developing ZK circuits without any need for a deep understanding of the underlying cryptographic techniques. It provides common programming constructs such as boolean types, integer types, checked arithmetic, bitwise operations, comparisons, structs, tuples, and conditional data flow. o1js's API also includes various cryptographic operations such as signature verification, encryption, nullifiers, proof recursion, and elliptic curve operations. Perhaps most notably, o1js can be used to write "zkApps": smart contracts with logic encoded into a ZK-circuit which may be deployed onto the Mina blockchain.

o1js is built on top of a PLONK-style proof-system called Kimchi, along with its sister library Pickles. Both projects are developed and maintained by o1Labs, and made available to o1js through the use of js-of-ocaml§. Veridise's review covered the use of the defined APIs and gadgets to properly implement and constrain programs written in o1js.

Code assessment. The o1Labs developers provided the source code of the o1js library for review. o1js appears to be original code written by o1Labs. It contains several layers of documentation. Internal documentation comments on functions and storage variables record the functionality of individual program constructs. Online documentation for o1js¶, Kimchi, and Pickles is also available. In addition, several example applications written using o1js help illustrate the intended use case of various APIs.

The source code contained a test suite, which the Veridise auditors noted covered most of the critical functionality. The test suite flexed the exported modules from each file, checking for consistency between provable evaluation and out-of-circuit evaluation. The o1js developers leverage property-based testing throughout the codebase, testing for correctness of various

* https://github.com/o1-labs/o1js
† https://o1-labs.github.io/proof-systems/introduction.html
‡ https://docs.minaprotocol.com/zkapps/zkapp-development-frameworks
§ https://ocsigen.org/js_of_ocaml/latest/manual/overview
¶ https://docs.minaprotocol.com/zkapps/o1js

operations on random input. These tests helped Veridise auditors understand intended usage and aided in quickly running the code and developing proofs-of-concepts.

While most portions of the code are well-tested, the Veridise team recommends o1js incorporate code coverage metrics into their testing workflow. Some changes adding features just before the audit began led to issues which would likely have been caught by tests (see, for example, V-O1J-VUL-004 and V-O1J-VUL-003). In addition, the Veridise auditors recommend adding more tests which supply "faulty" witness generators. This type of test may help to further increase confidence that each circuit is properly constrained.

During the first two weeks of the audit, the o1Labs developers made 1 change to the code. The portion of the code the Veridise auditors worked on for those two weeks was unchanged by the new commits.

Summary of issues detected. The audit uncovered 63 issues, 9 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, these issues consist of under-constrained circuits (V-O1J-VUL-001, V-O1J-VUL-006, V-O1J-VUL-008), vulnerable updates via actions and reducers (V-O1J-VUL-005, V-O1J-VUL-008), cryptographic errors (V-O1J-VUL-009, V-O1J-VUL-002), and logical errors introduced during rapid development (V-O1J-VUL-003, V-O1J-VUL-004). The Veridise auditors also identified 8 medium-severity issues, including other under-constrained circuits (V-O1J-VUL-010, V-O1J-VUL-015) and incorrectly modeled semantics (V-O1J-VUL-016, V-O1J-VUL-011, V-O1J-VUL-012, V-O1J-VUL-014). The Veridise auditors additionally identified 11 low-severity issues, 22 warnings, and 13 informational findings.

The Veridise auditors wish to note that some of the issues were identified by the o1Labs developers in parallel with the review and fixed without the aid of the Veridise auditors. This includes the high-severity issues V-O1J-VUL-005 and V-O1J-VUL-007, as well as issue V-O1J-VUL-041.

Among the 63 issues, 63 issues have been acknowledged by the o1Labs. Of the 63 acknowledged issues, o1Labs has fixed 24 issues and provided partial fixes to 5 more. This includes all issues of medium severity or higher, and all but one low issue (V-O1J-VUL-024). o1Labs does not plan to fix the other 34 acknowledged issues (all but one of warning or informational severity) at this time.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the o1js library.

Stronger Warnings. Some parts of the library may not be suitable for use in some or all environments. Veridise recommends the o1js developers add strong warnings to all APIs, documentation, and examples involving these library components. Three risks in particular should be made clear to users:

- The cryptographic implementations used are not designed to be constant-time, nor are secrets guaranteed to be zeroed out in memory when de-allocated. Any ZK-circuits which operate on sensitive material (e.g. private keys, mnemonics, or nullifier pre-images) should be executed within a secure environment, and be aware of the possibility of timing side-channels. Note that while each ZK-circuit itself cannot use control-flow statements typically associated with timing side channels, witness generators are still free to do so.

- As described in V-O1J-VUL-036, some hash functions do *not* use an injective padding scheme. This should be highlighted clearly to avoid major failures in cryptographic protocols.
- V-O1J-VUL-005 describes how actions and reducers are not yet currently safe for production. At the time V-O1J-VUL-005 was disclosed to o1Labs, this fact was not documented in the user-facing o1js documentation, on the APIs themselves, or in the examples which use them.

Note, however, that the o1Labs developers do provide extensive documentation on other security risks, which any o1js developer should consult before designing or deploying their protocol.

Testing. As described in the Code assessment above, the Veridise team recommends adding code coverage metrics to the testing workflow and ensuring all public APIs are thoroughly tested.

Additional Review. Given the number of severe issues identified through the course of the review, the Veridise auditors recommend the o1Labs developers take on another review of the library infrastructure, in addition to the increased testing recommended above. In particular, the cryptography library was the source of several issues (V-O1J-VUL-002, V-O1J-VUL-006, V-O1J-VUL-007, V-O1J-VUL-009, V-O1J-VUL-013, V-O1J-VUL-017), and the components tied to mina appear to still be under active development.

TypeScript Dangers. While TypeScript comes with a host of convenient and expressive language features, this can also lead to risks for users.

Any source-code verification tools should be very careful about the dependencies allowed in a program, and any future reviewers of o1js applications should ensure to review the dependencies used by the program. For example, a malicious dependency could load the o1js library, import a module, and mutate important methods (such as `check()` functions, which check type invariants). Since modules are cached, any subsequent imports by the victim o1js application developer will use the vulnerable methods.

Additionally, some quirks of JavaScript may not be well-known to all users. For example, integers may lose precision when they are larger than Number.MAX_SAFE_INT. Since very large values are common in ZK-applications, the Veridise auditors recommend adding checks to any API which accepts a `number` to ensure that it is within the safe range, preferring string or `bigint` representations for larger values.

Finally, users should be sure to compile using the strict flag to ensure all `undefined` values are properly handled.

TypeScript Best Practices. While most TypeScript conventions are adhered to throughout the codebase, use of some additional features and patterns may make the project more robust. Informational issue V-O1J-VUL-059 covers these in more detail. Largely, these recommendations consist of the following potential improvements:

- *Avoiding* `any`: In several portions of the code, especially in some of the files with a high-density of meta-programming, prevalent use of the `any` type ensures the file type-checks. This loss of safety may lead to hard-to-identify bugs down the road, especially in corner cases related to proper handling of `undefined`, `null`, or `void` values.
- *Advanced TypeScript Features*: Making more use of features like Symbols, `readonly` attributes, and private fields may improve the developer and user experience by avoiding accidental name collisions or unsupported field mutations.

Meta-Programming Refactoring. The complicated nature of meta-programming can be difficult to understand by nature. The added complexity of multiple possible ambient environments (such as circuit compilation, witness generation, and proving) adds another layer of complexity. Certain parts of the codebase which address these complex problems (such as `zkapp.ts` and `account-update.ts`) could be made easier to reason about by further separating functions based on the ambient environment and breaking down the files into smaller components.

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 Veridise or any of its employees 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
o1js	`02f2ffb6-8dde2c3b`	TypeScript	Mina
o1js-bindings	`e7ded4b6-3c68a0da`	TypeScript	Mina

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Mar. 18 - Jun. 21, 2024	Manual & Tools	2	39 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	2	2	2
High-Severity Issues	7	7	6
Medium-Severity Issues	8	8	6
Low-Severity Issues	11	11	8
Warning-Severity Issues	22	22	2
Informational-Severity Issues	13	13	0
TOTAL	63	63	24

Table 2.4: Category Breakdown.

Name	Number
Logic Error	18
Maintainability	13
Data Validation	10
Under-constrained Circuit	9
Hash Collision	4
Over-constrained Circuit	3
Usability Issue	2
Denial of Service	1
Race Condition	1
Authorization	1
Transaction Ordering	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of o1js's zero-knowledge circuits. In our audit, we sought to answer questions such as:

- Are any circuits under-constrained, i.e. do they admit unexpected or manipulable solutions?
- Are any circuits over-constrained, i.e. do any valid inputs have no solution to the constraints?
- Are cryptographic primitives/protocols implemented safely and correctly?
- Do standard types/operations have the expected semantics?
- Are custom user-types properly encoded into a circuit?
- Are the expected type invariants checked (in-circuit) whenever an o1js object is created using the public APIs?
- Does zkApp execution properly check input proofs for method calls?
- Does o1js observe common TypeScript best practices?
- Are assumptions/caller-responsibilities properly documented and fulfilled?
- Can zkApps be DoS'ed using any public APIs?
- Are `AccountUpdate` preconditions properly tracked and implemented?
- How do user extensions of the library affect the (in-circuit) type-safety of o1js values?
- Is any unintended information about the execution or input data leaked to the verifier?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we wrote custom Semgrep rules, automatically scanning for common anti-patterns or vulnerable control-flow paths identified by the Veridise auditors.
- *Formal verification.* We leverage our custom formal verification tool Picus and a tool Rosette to verify properties of the zero-knowledge circuits and various functions (see Chapter 5 for discussion and results). Picus is designed to prove or find violations of determinism, which is an important safety property for zero-knowledge circuits, while Rosette is a tool that can be used to prove or find violations of functional correctness.
- *Fuzzing/Property-based Testing.* We also leverage fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we defined a set of invariants that the protocol must hold and fuzz tested it to find an input that breaks one of these invariants. Our fuzzing methodology is explained in detail in Chapter 6.

Scope. The scope of this audit is limited to the below files in the o1js* and o1js-bindings† repositories, focusing on the constraints defined within those files, and the methods used to encode objects to and from fields.

- o1js
  - `src/lib/mina/`
    - `zkapp.ts`
    - `account-update.ts`
    - `mina.ts`
    - `precondition.ts`
    - `state.ts`
    - `events.ts`
  - `src/lib/proof-system/`
    - `circuit.ts`
  - `src/lib/provable/`
    - `core`
    - `field.ts`
    - `bool.ts`
    - `int.ts`
    - `provable.ts`
    - `crypto/`
      - `signature.ts`
      - `hash.ts`
      - `nullifier.ts`
      - `foreign-curve.ts`
      - `foreign-ecdsa.ts`
      - `keccak.ts`
      - `encryption.ts`
    - `group.ts`
    - `scalar.ts`
    - `gates.ts`
    - `gadgets/`
    - `merkle-tree.ts`
    - `merkle-map.ts`
    - `types/`
      - `circuit-value.ts`
      - `provable-derivers.ts`
    - `foreign-field.ts`
  - `src/`
    - `snarky.d.ts`
    - `snarky.js`
  - `src/lib/ml/`
    - `base.ts`

* https://github.com/o1-labs/o1js
† https://github.com/o1-labs/o1js-bindings

- o1js-bindings
  - `lib/`
    - `generic.ts`
    - `provable-generic.ts`
    - `provable-snarky.ts`
    - `from-layout.ts`
  - `mina-transaction/`
    - `transaction-leaves.ts`
    - `derived-leaves.ts`
    - `gen/transaction.ts`
  - `ocaml/lib/`
    - `snarky-bindings.ml`
    - `pickles-bindings.ml`
  - `crypto/`
    - `finite-field.ts`
    - `random.ts`
    - `bigint-helpers.ts`
    - `elliptic-curve.ts`
    - `elliptic-curve-endomorphism.ts`

Methodology. Veridise auditors reviewed the reports of previous audits for o1js, inspected the provided tests, and read the o1js documentation. They then began a manual review of the code assisted by static analyzers, automated testing, and formal verification. During the audit, the Veridise auditors regularly met with the o1js developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise 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	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

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 in Table 3.3:

Table 3.3: Impact Breakdown

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 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. 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 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-O1J-VUL-001	unhash() on Field3s	Critical	Fixed
V-O1J-VUL-002	Collisions in Merkle map key to index	Critical	Fixed
V-O1J-VUL-003	Incorrect inequality direction	High	Fixed
V-O1J-VUL-004	UInt64.rightShift() performs left shift	High	Fixed
V-O1J-VUL-005	Reducer may be DoSed	High	Fixed
V-O1J-VUL-006	check()s in EcdsaSignature and . . .	High	Fixed
V-O1J-VUL-007	Provers may falsely prove ECDSA signatures	High	Fixed
V-O1J-VUL-008	Action state updated on empty action list	High	Fixed
V-O1J-VUL-009	Non-CCA-secure encryption	High	Partially Fixed
V-O1J-VUL-010	Incorrect default for toBits()	Medium	Fixed
V-O1J-VUL-011	MayUseToken fields may both be true	Medium	Fixed
V-O1J-VUL-012	Asserting preconditions may overwrite . . .	Medium	Partially Fixed
V-O1J-VUL-013	Improperly constrained scale()	Medium	Fixed
V-O1J-VUL-014	Inheritance should not be supported for . . .	Medium	Fixed
V-O1J-VUL-015	isPositive() underconstrained at 0	Medium	Fixed
V-O1J-VUL-016	Provable.equals() unsound for some types	Medium	Partially Fixed
V-O1J-VUL-017	Missing canonical form check in verifyEcdsa	Medium	Fixed
V-O1J-VUL-018	Incorrect permissions for setVerificationKey	Low	Fixed
V-O1J-VUL-019	Incorrect usage of parametric polymorphism	Low	Partially Fixed
V-O1J-VUL-020	Incorrect array handling and type . . .	Low	Fixed
V-O1J-VUL-021	Missing check on infinity flag	Low	Fixed
V-O1J-VUL-022	Bound checked on length instead of . . .	Low	Fixed
V-O1J-VUL-023	quotientBits bound too low in divMod32	Low	Fixed
V-O1J-VUL-024	Unsafe use of toProjective	Low	Acknowledged
V-O1J-VUL-025	Missing range check in assertOnCurve	Low	Fixed
V-O1J-VUL-026	Reduction of ForeignField element	Low	Fixed
V-O1J-VUL-027	Unchecked prefix could lead to collision	Low	Fixed
V-O1J-VUL-028	Field/curve operations assuming . . .	Low	Partially Fixed
V-O1J-VUL-029	Infinite loop from user error	Warning	Acknowledged
V-O1J-VUL-030	Prover errors when calling set() on same . . .	Warning	Acknowledged
V-O1J-VUL-031	Over-constrained doubling circuit	Warning	Acknowledged
V-O1J-VUL-032	Negative powers unhandled	Warning	Acknowledged
V-O1J-VUL-033	Almost reduced described incorrectly	Warning	Acknowledged
V-O1J-VUL-034	Off-by-one in some base conversions	Warning	Acknowledged
V-O1J-VUL-035	Hash collisions in curve domain separator	Warning	Acknowledged
V-O1J-VUL-036	Non-injective padding for hash functions	Warning	Acknowledged

V-O1J-VUL-037	Cofactor clearing over-constrained	Warning	Acknowledged
V-O1J-VUL-038	Unhandled Map/Set corner cases	Warning	Acknowledged
V-O1J-VUL-039	Unsafe construction of UInt64 with FieldVar	Warning	Acknowledged
V-O1J-VUL-040	Over-constrained foreign-field operations	Warning	Acknowledged
V-O1J-VUL-041	Incorrect Merkle witness key computation ...	Warning	Fixed
V-O1J-VUL-042	Missing public key validation checks	Warning	Acknowledged
V-O1J-VUL-043	New layout types may lead to hash collisions	Warning	Acknowledged
V-O1J-VUL-044	Missing high-limb constraints when bit- ...	Warning	Acknowledged
V-O1J-VUL-045	Avoid mutating config input	Warning	Acknowledged
V-O1J-VUL-046	toFields depends on declaration order	Warning	Acknowledged
V-O1J-VUL-047	Failure of group addition during witness ...	Warning	Fixed
V-O1J-VUL-048	Possible issues during serialization	Warning	Acknowledged
V-O1J-VUL-049	No override of check() for derived leaf types	Warning	Acknowledged
V-O1J-VUL-050	Missing length check in dot product	Warning	Acknowledged
V-O1J-VUL-051	Typos and incorrect comments	Info	Acknowledged
V-O1J-VUL-052	Incorrect squeeze computation	Info	Acknowledged
V-O1J-VUL-053	Missing hash-to-curve best practices	Info	Acknowledged
V-O1J-VUL-054	Over-constrained neg()	Info	Acknowledged
V-O1J-VUL-055	console.assert() only prints error	Info	Acknowledged
V-O1J-VUL-056	Use bitlen() instead of log2()	Info	Acknowledged
V-O1J-VUL-057	Recommended documentation	Info	Acknowledged
V-O1J-VUL-058	Missing checks on constants	Info	Acknowledged
V-O1J-VUL-059	TypeScript recommendations/best practices	Info	Acknowledged
V-O1J-VUL-060	Inverse assertion allows for common anti- ...	Info	Acknowledged
V-O1J-VUL-061	Assignment instead of copy	Info	Acknowledged
V-O1J-VUL-062	Duplicate, unused, or outdated code	Info	Acknowledged
V-O1J-VUL-063	Potentially incorrect rounding	Info	Acknowledged

4.1 Detailed Description of Issues

4.1.1 V-O1J-VUL-001: unhash() on Field3s

Severity	Critical	**Commit**	8dde2c3, e7ded4b
Type	Under-constrained Circuit	**Status**	Fixed
File(s)	`src/lib/provable/gadgets/elliptic-curve.ts`, `src/lib/provable/scalar.ts`		
Location(s)	multiScalarMul()		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1669`, `https://github.com/o1-labs/o1js/pull/1757`		

Inside `multiScalarMul()`, a `HashedPoint` is used to make the circuit more efficient when selecting from different `Points`.

```
        const HashedPoint = Hashed.create(Point.provable);
        // [VERIDISE] ...
        let sjP =
          windowSize === 1
            ? points[j]
            : arrayGetGeneric(
                HashedPoint.provable,
                hashedTables[j],
                sj
              ).unhash();

        // ec addition
        let added = add(sum, sjP, Curve);
```

Snippet 4.1: Snippet from `multiScalarMul()`

A `HashedPoint` is a pair `(value, hashedValue)`, with a proof that `hash(value) = hashedValue`. "Un-hashing" proves that a fresh variable `x` also satisfies `hash(x) = hashedValue`.

However, there is one major caveat. Before computing the hash, `x` is packed. For example, if `x` is an array of two `Bools` `x = [b1 b2]`, then the hash computed is `hash(b2 + 2 * b1)`. It is very important that the range checks required on `x`'s type are applied to `x` so that this packing is injective. If `b1` or `b2` were not constrained to be in `{0,1}`, then the map `b2 + 2 * b1` could have many exploitable collisions.

In fact, this is the case for `Point`, which is a provable 2-tuple of `Field3`s. `Field3`'s provable type claims it can be packed into 88 bits, but does not override `check()`. Only `ForeignField` overrides `check()` to perform a multi-range-check.

This can be validated using the following test

```
class Secp256k1 extends createForeignCurve(Crypto.CurveParams.Secp256k1) {}

const HashedPoint = Hashed.create(Point.provable);
await Provable.runAndCheck(() => {
  let gConst: Point = {
    x: Secp256k1.generator.x.value,
    y: Secp256k1.generator.y.value
```

```
  }

  let g = Provable.witness(Point.provable, () => gConst);
  Provable.log("hashing!")
  let hg = HashedPoint.hash(g);

  let customHashInverter = () => {
    let inverseBeforePacking = gConst;
    // add and subtract
    let shift = inverseBeforePacking.x[0].toBigInt();
    inverseBeforePacking.x[0] = inverseBeforePacking.x[0].sub(shift);
    inverseBeforePacking.x[1] = inverseBeforePacking.x[1].add(shift * 2n**88n);
    return inverseBeforePacking;
  }

  Provable.log("Unhashing!")
  Provable.log("g:", g);
  let hInvHg = hg.unhash(customHashInverter);

  Provable.log("Hinv(H(g)):", hInvHg);
})
```

which succeeds, outputting

```
hashing!
Unhashing!
g: {
  x: [
    '24923162292477743273765058 4',
    '11918217268833954807813610 9',
    '57491861141639725661123 2'
  ],
  y: [
    '16118428522310728324696184 8',
    '30463067655878880881516765 4',
    '34109604001639692274013 2'
  ]
}
Hinv(H(g)): {
  x: [
    '0',
    '77133451268664514553348723976062470028622621308672813',
    '57491861141639725661123 2'
  ],
  y: [
    '16118428522310728324696184 8',
    '30463067655878880881516765 4',
    '34109604001639692274013 2'
  ]
}
```

(note that we modified `unhash()` to accept a custom witness generator).

```
diff --git a/src/lib/provable/packed.ts b/src/lib/provable/packed.ts
index 6bf7d8690..23d38c818 100644
```

```
--- a/src/lib/provable/packed.ts
+++ b/src/lib/provable/packed.ts
@@ -1,13 +1,13 @@

 /**
  * ‘Packed<T>‘ is a "packed" representation of any type ‘T‘.
@@ -237,10 +237,8 @@ class Hashed<T> {
   /**
    * Unwrap a value from its hashed variant.
    */
-  unhash(): T {
-    let value = Provable.witness(this.Constructor.innerProvable, () =>
-      this.value.get()
-    );
+  unhash(witnessGenerator: () => T = () => this.value.get()): T {
+    let value = Provable.witness(this.Constructor.innerProvable, witnessGenerator);

     // prove that the value hashes to the hash
     let hash = this.Constructor._hash(value);
```

The same issue applies to `Scalars`. `Scalars` are packed into individual bits when hashing, but not directly checked by `check()`. This may allow unsafe `unhashing`.

Impact By exploiting this fact, attackers can control the output of the multi-scalar computation. This affects any invocation of `scale()`, and may also allow verification of ECDSA signatures for invalid keys.

The above PoC allows one to choose a `shift` freely, so that `[x0-shift, x1+2**88*shift, x2]`represents

```
shifted(X) := (x0 - shift mod n) + (x1 + shift *2**88 mod n) * 2*88 + x2 * 2**176
```

If we want this value to become equivalent to `[z0, z1, z2]`, we can choose `shift` so that, for some `q` (yet to be chosen)

```
x0 - shift = z0 + q f0 mod n
```

Next, we look at

```
  x1 + shift * 2**88
= x1 + (x0 - z0 - q f0) * 2**88 mod n
= x1 + (x0 - z0) * 2**88 - q f0 * 2**88 mod n
```

Setting this equal to our target of `z1 + q f1 + (z2 - x2 + q f2) * 2**88`, we have

```
q = (z1 - x1 + q f1 + (z2 - x2 + q f2 - x0 + z0) * 2**88) / (f0 * 2**88) mod n
```

Then, we have

```
shifted(X) = z0 + (z1 + q f1 + (z2 - x2 + q f2) * 2**88 + x2 * 2**176
           = z0 + q f0 + (z1 + q f1)* 2**88 + (z2 + q f2) * 2**176
           = [z0 + qf0, z1 + qf1, z2 + qf2]
```

In particular, by choosing the appropriate `shift`, we can make `sjP` represent *any value we desire* modulo the foreign field.

It is possible other constraints in the later EC gates would lead this shifted value to fail an assertion. However, since this attack violates an invariant assumed by almost all later circuits (i.e. that the `Field3` is multi-range-checked), it is most likely exploitable in numerous locations. This is just a rough sketch of how one might go about crafting an attack.

Recommendation Override the `check()` function of `Field3` and add warnings to the documentation of `unhash()`.

Consider using the `ForeignField` abstraction in `elliptic-curve.ts` to prevent this and other errors (see V-O1J-VUL-025).

Developer Response We resolved this issue by no longer hashing or packing points.

Updated Veridise Response The provided fix resolves the vulnerability described above. We recommend adding some additional safety precautions for users:

1. `multiScalarMul()` now defaults to `hashed: boolean = true`, which is the unsafe case. This is exposed to the user by `EllipticCurve.multiScalarMul()` and implicitly in `EllipticCurve.scale()`. Consider
   a) Defaulting `hashed` to `false`.
   b) Changing `scale()` to explicitly set `hashed` to `false`.
2. Adding documentation to `Hashed` describing this potential vulnerability.
3. Adding warnings which print to the terminal when deprecated functionality is used.

Updated Developer Response We have removed the deprecated methods and performed the requested fixes in the V2 branch.

4.1.2 V-O1J-VUL-002: Collisions in Merkle map key to index

Severity	Critical	**Commit**	8dde2c3
Type	Hash Collision	**Status**	Fixed
File(s)	`src/lib/provable/merkle-map.ts`		
Location(s)	computeRootAndKey()		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1694,https://github.com/o1-labs/o1js/pull/1715/files`		

`MerkleMaps` are maps from `Fields` to `Fields`, represented as a Merkle tree. As described in V-O1J-VUL-041, each `key` is associated with an index in the underlying Merkle tree. To compute the value stored in the `MerkleMap` for a given `key`, one first computes the `index`, and then opens the Merkle tree leaf at that index.

The key-to-index map is computed as a bit-reversal. For example, the `key` 1 (represented as the 255 bits 000...01) corresponds to the index 10...000, i.e. 2^{254}. However, since the Pasta field modulus is smaller than 2^{255}, multiple distinct `indices` may be associated to multiple `keys`. This leads to potential collisions.

For example, in the below code snippet, we can see that the index corresponding to the bit-reversal of `Fp.modulus` has the same key (`0`) as index `0`.

```
class MyMerkleWitness extends MerkleWitness(bits + 1) {}
const map = new MerkleMap();

// Get a key which will collide with other keys, even though it is
// associated to a different index
const overflowKey = Fp.modulus;
let reversedOverflowIndex = overflowKey;
let overflowIndex = 0n;
for(let i = 0; i < 255n; ++i) {
  overflowIndex *= 2n;
  overflowIndex += reversedOverflowIndex & 1n;
  reversedOverflowIndex = reversedOverflowIndex >> 1n;
}

const collisionKey = Fp.mod(overflowKey)
const collisionIndex = map._keyToIndex(new Field(collisionKey));
assert(collisionIndex == map._keyToIndex(new Field(collisionKey)));

// Validate the keys collide, but the indices do not
assert(Fp.mod(collisionKey) == Fp.mod(overflowKey), `${collisionKey} != ${overflowKey
    } % ${Fp.modulus}`);
assert(collisionIndex != overflowIndex, `${collisionIndex} == ${overflowIndex}`);
```

Snippet 4.2: Computation of two distinct indices which have the same `key`

Using this fact, we may produce a proof for a value at the specified `key`, but choose to use either `index`.

```
// Store a non-zero value into the collision key
const zero = new Field(0n);
const one = new Field(1n);
map.set(new Field(collisionKey), one);
```

```
5
6  // Prove that 1 is stored at collisionKey
7  const witness = new MyMerkleWitness(map.tree.getWitness(collisionIndex));
8  const provableWitness = new MerkleMapWitness(witness.isLeft, witness.path);
9
10 const [hash, key] = provableWitness.computeRootAndKey(one);
11 // validate the hash matches
12 assert(hash.toBigInt() === map.getRoot().toBigInt(), `Hashes do not match!\n${hash}\n
       !=\n${map.getRoot()}`);
13 // validate the key is the collision key
14 assert(key.toBigInt() === collisionKey, `Keys do not match!\n${key}\n!=\n${
       collisionKey}`)
15
16 // Get a witness for the rightmost leaf
17 const evilWitness = new MyMerkleWitness(map.tree.getWitness(overflowIndex));
18 const provableEvilWitness = new MerkleMapWitness(evilWitness.isLeft, evilWitness.path
       );
19
20 // Compute the hash/key of this evil witness to prove that 0 is stored at collision
       key
21 const [evilHash, evilKey] = provableEvilWitness.computeRootAndKey(zero);
22 // validate the hash matches
23 assert(evilHash.toBigInt() === map.getRoot().toBigInt(), `Evil hash does not match!\n
       ${evilHash}\n!=\n${map.getRoot()}`);
24 // validate the key is the collision key
25 assert(evilKey.toBigInt() === collisionKey, `Evil key does not match!\n${evilKey}\n
       !=\n${collisionKey}`)
```

Snippet 4.3: Store `1` at index `0`. We may then prove that `map` at `0` is either `1` or `0`.

Impact As shown in the above proof of concept, a user may prove that some entries of a `MerkleMap` are empty, even after they have been set. This can have devastating consequences to an application.

For example, suppose a reputation-system increments a user's reputation during registration as a one-time bonus, then adds them to a Merkle tree. A user may use this technique to repeatedly "re-register," increasing their reputation arbitrarily.

As another example, a user could prove their address is not in a blacklist, or that a certain nullifier is not in a Merkle tree.

Recommendation Add a range-check on to prevent overflow.

Note also that the developers should consider using `key.toBits(bits-1)` instead of `key.toBits().slice(0,bits)`. This will ensure collisions do not occur due to ignoring high-bits.

Developer Response To prevent collisions, we now assert that the high bit of the `key` is 0.

After applying this fix, if someone were to pass in a key with 255 bits, this assertion would fail. Developers hitting this error would have to redesign how they use `MerkleMap` and do additional

work to ensure their keys fit in 254 bits. In practice, this will not happen. Existing usage of `MerkleMap` will almost always compute the key as a Poseidon hash or at least as a small field element. A hash will only have 254 bits with overwhelming probability.

Updated Veridise Response This addresses the described issue. However, it seems that `nullifier.ts` still uses the deprecated `computeRootAndKey()`.

4.1.3 V-O1J-VUL-003: Incorrect inequality direction

Severity	High	**Commit**	8dde2c3
Type	Logic Error	**Status**	Fixed
File(s)	`o1js/src/lib/provable/gadgets/foreign-field.ts`		
Location(s)	assertLessThan()		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1723/files`		

The `assertLessThan()` function asserts that `x < y`. It handles three different cases: both constant, one variable, and both variable. Parts of the first two cases are shown in the below snippet.

```
function assertLessThan(x: Field3, y: bigint | Field3) {
    let y_ = Field3.from(y);

    // constant case [VERIDISE: REMOVED FOR BREVITY] ...

    // case of one variable, one constant

    if (Field3.isConstant(x)) return assertLessThan(y_, x);
```

Snippet 4.4: Snippet from `assertLessThan()`

In the case that `y` is variable, but `x` is constant, this function calls `assertLessThan(y_, x)`. This is incorrect, since the `x` should be the smaller element.

Impact This function is used by several public APIs, including `Field.assertLessThan()` (the public API invokes `assertLessThanFull()`, which calls `assertLessThan()`).

For example, the first assertion in the below snippet passes, while the second fails.

```
class SmallField extends createForeignField(17n) {}

function checkValues() {
  let [zero, oneVar] = Provable.witness(
    provableTuple([Field, Field]),
    () => [new Field(0n), new Field(1n)]
  )
  let oneConst = new Field(1n);

  Provable.log("Beginning asserts...")
  oneConst.assertLessThan(zero);
  Provable.log("Success one");
  oneVar.assertLessThan(zero);
  Provable.log("Success two");
}
await Provable.runAndCheck(checkValues);
console.log("Check passed!");
```

Output:

```
finished build
Beginning asserts...
Success one
[Error: multi-range check failed
```

```
Constraint unsatisfied (unreduced):
```

Recommendation Handle the (constant, variable) case and (variable, constant) cases separately.

Developer Response We re-used the (variable, variable) implementation for the (constant, variable) case.

Updated Veridise Response This does resolve the core issue. One inline comment is now out of date. We also recommend adding all four combinations of constant, non-constant to the test suite.

Updated Developer Response We addressed the inline comment.

4.1.4 V-O1J-VUL-004: UInt64.rightShift() performs left shift

Severity	High	**Commit**	8dde2c3
Type	Logic Error	**Status**	Fixed
File(s)	`src/lib/provable/int.ts`		
Location(s)	UInt64.rightShift()		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1617/`		

The function `UInt64.rightShift()` shifts the bits left instead of right.

```
1 rightShift(bits: number) {
2   return new UInt64(Bitwise.leftShift64(this.value, bits).value);
3 }
```

Snippet 4.5: Definition of `UInt64.rightShift()`.

Impact Use of the function will result in the opposite of the intended behavior.

Recommendation Use `Bitwise.rightShift64()`. Add unit tests for each of these user-facing methods. Consider creating a shared base class as recommended in V-O1J-VUL-061.

Developer Response We applied the recommendation.

4.1.5 V-O1J-VUL-005: Reducer may be DoSed

Severity	High	**Commit**	8dde2c3
Type	Denial of Service	**Status**	Fixed
File(s)	`src/lib/mina/zkapp.ts`		
Location(s)	reduce()		
Confirmed Fix At	`https://github.com/o1-labs/docs2/pull/1011`, `https://github.com/o1-labs/o1js/pull/1728`		

The `reduce()` function performs a sequence of actions, allowing for concurrent submission of requests. The goal of `reduce()` is to perform the below for-loop on the provided action state

```
newActionDigest = processedActionsDigest
for( actionLists in submittedAndUnprocessedUpdates ):
    for (action in actionList ):
            newState = f(newState, action) // f() is the user-provided reduce()
    function
      newActionStateDigest = nextDigest(newActionStateDigest, actionList)

// Ensure all processed actions were actually submitted and therefore
// verified as valid action submissions
assert(newActionStateDigest == submittedActionsDigest)
```

Snippet 4.6: Pseudo-code for `reduce()`

Unfortunately, the number of unprocessed `AccountUpdates` is unknown at compile time, and the number of actions emitted by an `AccountUpdate` is also unknown at compile time. To account for this, the `reduce()` function unrolls both loops based on the arguments supplied to `reduce()`:

- The outer loop is unrolled `maxTransactionsWithActions` times. By default, this is `32`.
- The inner loop is unrolled `maxActionsPerMethod+1` time. By default, this is `1+1=2`. The additional unrolling is necessary to handle the case of 0 provided actions.

To support the possibility of *fewer* than `maxTransactionsWithActions` account updates needing to be processed, or a method outputting *fewer* than `maxActionsPerMethod`, two distinct approaches are taken:

1. An empty list of actions may be supplied, effectively producing a NOP iteration of the outer loop. Effectively, an honest prover should always pad `actionLists` out to `maxTransactionsWithActions`.
2. For the inner loop, every possible iteration count is executed. More precisely, the inner loop is executed with 0 iterations, 1 iteration, 2 iterations, . . . , up to `maxActionsPerMethod` iterations. Then, boolean selectors (called `lengths`) are used to select which execution represents the "true" computation.

The correctness of (2.) is argued as follows: since the `newActionStateDigest` must match the (Mina-managed, on-chain value) `submittedActionsDigest`, the correct selector must have been used to compute the action digest, and the supplied actions must have previously been submitted using `Actions.dispatch()`. Therefore, the state computed on those actions is the correct final state. See V-O1J-VUL-008 for an edge case in which this argument breaks down.

An honest prover supplies "dummy" data (i.e. `actionType.empty()`) for each other iteration.

There are three important consequences to this design:

1. One execution of `reduce()` must be able to process *all* submitted, unprocessed actions in order for verification to succeed.
2. Executing the user-supplied reduction function (called `f()` in the above pseudo-code, but referred to as the `reduce` argument passed to the `reduce()` function as a lambda in the o1js repository) must not have any assertion failures on the "non-selected" iteration counts from the inner loop. For example, suppose the user-supplied reduction function asserts that the `actionState` is non-zero. Since the default prover supplies an empty `actionType` to the function in un-selected executions of the inner loop, this will lead to verification errors.
3. Since the default prover calls `Provable.witness()` on an empty action type, `emptyValue(type)` must provide a valid instance of the `actionType`. However, `emptyValue(type)` calls `type.fromFields()` on an array of all zeros, rather than using `type.empty()` when defined.

Impact The most severe consequence is item one above. Any user may DoS a reducer by simply submitting a large number of actions.

The second of the above consequences may lead to more subtle verification issues. So long as an honest prover may be found for a user-supplied `reduce` function, a full DoS will not occur. However, this may still lead to temporary shutdown of a contract. In rare cases where the `reduce` function is highly structured, it may be impossible to find a valid prover.

Finally, types for which the all-zero-fields value is not a valid instance of the type will need a custom prover.

Normally, this would be a critical bug. However, as mentioned in the developer response, actions are not yet intended to be supported in production. Since the feature is still included in the core documentation of o1js without mention of this attack vector, we have still listed the issue as high. See `https://docs.minaprotocol.com/zkapps/writing-a-zkapp/feature-overview/actions-and-reducer`.

Recommendation All documentation and examples referencing actions/reducers should be updated to indicate that they are not safe for production. The above issues should be fully documented as well.

We also recommend renaming several variables to make their purposes more clear:

- Consider renaming `maxTransactionsWithActions` to `maxAccountUpdatesWithActions`.
- Consider using different names for the `actionState` representing the digest of processed actions and the `account.actionState` represent the digest of submitted actions.
- Consider renaming "event" terminology to use "actions" terminology.
- Consider using a different name than `reduce()` for the user-provided `reduce()` function.

Developer Response This is a well-known issue, and fixes are in progress. See Batch reducer and recursive reducer. The vulnerability is now clearly stated in our documentation, and other refactors have improved the variable/function naming.

4.1.6 V-O1J-VUL-006: check()s in EcdsaSignature and ForeignCurve miss multi-range-check

Severity	High	**Commit**	8dde2c3
Type	Under-constrained Circuit	**Status**	Fixed
File(s)	`src/lib/provable/crypto/foreign-ecdsa.ts`, `src/lib/provable/crypto/foreign-curve.ts`		
Location(s)	check()		
Confirmed Fix At	https://github.com/o1-labs/o1js/pull/1703		

Both `EcdsaSignature` and `ForeignCurve` override `check()` to use `assertAlmostReduced`.

```
1  static check(signature: EcdsaSignature) {
2    // more efficient than the automatic check, which would do this for each scalar
     separately
3    this.Curve.Scalar.assertAlmostReduced(signature.r, signature.s);
4  }
```

Snippet 4.7: Snippet from `check()` from `EcdsaSignature`

`assertAlmostReduced` in turn calls `Gadgets.ForeignField.assertAlmostReduced` with `skipMrc` set to true

```
1  static assertAlmostReduced<T extends Tuple<ForeignField>>(
2    ...xs: T
3  ): TupleMap<T, AlmostForeignField> {
4    Gadgets.ForeignField.assertAlmostReduced(
5      xs.map((x) => x.value),
6      this.modulus,
7      { skipMrc: true }
8    );
9    return Tuple.map(xs, this.AlmostReduced.unsafeFrom);
10 }
```

Snippet 4.8: Snippet from `assertAlmostReduced()`

Thus, a precondition to call `assertAlmostReduced` is that the arguments are already multi-range-checked. However, this check is missing. A malicious witness generator could then supply overflow values that pass all assertions.

Impact `check()` will not properly prevent prevent invalid signature (resp invalid foreign curve), as demonstrated here:

```
1  class Ecdsa extends createEcdsa(CurveParams.Secp256k1) {}
2
3  let gx: Field3 = Ecdsa.Curve.generator.x.value;
4  let gy: Field3 = Ecdsa.Curve.generator.y.value;
5
6  gx[0] = gx[0].add(gx[1].mul(2n**88n));
7  gx[1] = new Field(0n);
8
9  gy[0] = gy[0].add(gy[1].mul(2n**88n));
10 gy[1] = new Field(0n);
11
```



```
  console.log(`pt: {
    x: ${gx},
    y: ${gy}
  }`);

  // Not really a signature, just some non-mrc'd points to illustrate
  // the EcdsaSignature check() function does not check properly
  let sig = new Ecdsa({
    r: gx,
    s: gy,
  });
  Ecdsa.check(sig); // assertion passed
```

Snippet 4.9: Code snippet showing non-multi-range-checked `Field3`s passing the `check()`.

This, then, would cause invariant violations of functions like ECDSA verification.

Recommendation Perform a multi-range check on arguments passed to the `assertAlmostReduced` function.

Developer Response We added a range check on inputs within the `check` functions of `EcdsaSignature` and `ForeignCurve`.

Updated Veridise Response This addresses the core issue. We recommend adding warnings which print to the terminal when deprecated functionality is used.

Developer Response Regarding the final comment, we believe printing warnings is a breaking change, and should not be done as part of v1.

4.1.7 V-O1J-VUL-007: Provers may falsely prove ECDSA signatures

Severity	High	**Commit**	8dde2c3
Type	Logic Error	**Status**	Fixed
File(s)	`src/lib/provable/gadgets/elliptic-curve.ts`		
Location(s)	multiScalarMul()		
Confirmed Fix At	https://github.com/o1-labs/o1js/pull/1545,https://github.com/o1-labs/o1js/pull/1752		

The `multiScalarMul()` function computes $\sum_i s_i P_i$ for provided scalars s_i and points P_i. However, since there is no valid `Point` representation of the identity, zero or non-zero assertions on the result are difficult to describe.

To get around this, the developers introduce an "initial aggregator" `ia`. This is a point with a discrete logarithm known to no one. Rather than compute the above sum directly, `multiScalarMul()` computes `2^(b-1) * ia + sum(s[i] * P[i])` (for some known constant `b`). Then, the result may be compared to `2^(b-1) * ia` to determine whether or not it is zero.

When performing ECDSA, `multiScalarMul()` is invoked with scalars `[u1, u2]` and points `[g, Q]`, with `g` the group generator and `Q` the public key. By constructing `Q` appropriately, a malicious prover may create signatures which they can "prove" are valid, but do not actually verify under ECDSA. They can do this by leveraging the under-constrained nature of `add()`, which outputs a result fully under prover control when adding a point to itself.

Attack (Motivating Example)

The basic goal of this attack is to construct `Q` and a signature `(r,s)` so that `sum(s[i] * P[i]) == 2^(b-1} * ia`, thus giving the prover full control of the result in the last iteration of the sum.

To do so, an attacker will choose a random `d` in the scalar field, and set

```
Q := d * g + 2^(b-1) * ia
```

Then, provided message hash `m`, they will compute

```
r := -m / d
s := -m / d
```

ECDSA will then compute

```
u1 := m / s
    = m / (-m / d)
    = -d
u2 := r / s
        = s / s
        = 1
```

Finally, we have

```
  u1 * g + u2 * Q
= -d * g +  1 * (d * g + 2^(b-1) * ia)
= -d * g + d * g + 2^(b-1) * ia
= 2^(b-1) * ia
```

as desired.

Order of Operations

The above attack will not actually work, since the sum is accumulated into `ia` term by term, not all at once.

```
for (let i = maxBits - 1; i >= 0; i--) {
  // add in multiple of each point
  for (let j = 0; j < n; j++) {
    let windowSize = windowSizes[j];
    if (i % windowSize === 0) {
      // pick point to add based on the scalar chunk
      let sj = scalarChunks[j][i / windowSize];
      let sjP =
        windowSize === 1
          ? points[j]
          : arrayGetGeneric(
              HashedPoint.provable,
              hashedTables[j],
              sj
            ).unhash();

      // ec addition
      let added = add(sum, sjP, Curve);

      // handle degenerate case (if sj = 0, Gj is all zeros and the add result is
    garbage)
      sum = Provable.if(sj.equals(0), Point.provable, sum, added);
    }
  }
```

Snippet 4.10: Main loop from `multiScalarMul()`

However, by instead choosing `r` and `s` so that

```
u1 = d
u2 = 1
```

`Q` will contribute nothing to the sum until the very last `i=0, j=1` iteration, since the bits are iterated over in reverse order. Therefore, immediately before the `i=0, j=1` iteration, the sum will equal

```
2^(b-1) * ia + d * g = Q = u2 * Q
```

Of course, this ignores the GLV decomposition. We expect the same attack may be applied in that case, simply with more algebra.

Impact Attackers may use this in settings in which they must provably generate a valid signature which may then be redeemed at a later date.

For example, consider the following scenario:

- Someone implements a distributed oracle as follows:

- Anyone can register an ECDSA public key by signing a message and depositing a stake associated to that public key.
- Registered users may vote on the current price by submitting a (signature, new price) pair, with the signature provably validated.
- If a registered user violates some slashing condition, other users can provide (signature1, price1), (signature2, price2) as evidence to slash the malicious user's deposit, and remove them from the oracle system.

By using the above false verification, a user may become immune to slashing in this system.

Note, however, that this "false proof" can only be constructed for keys which the attacker can represent as `a * g + b * ia` for some known `a`, `b`. Consequently, in many applications, this will not be an issue.

Generally, this may cause problems if

1. Signing a "bad" piece of data should provoke a penalty.
2. Signing a "good" piece of data generates some sort of debt, paid when the signature is "cashed back in" by another user.

Recommendation Implement the `g = h` case in `add()`.

Developer Response We fixed this by fully constraining EC addition.

There is one caveat: our fix likely breaks completeness for small curves with base fields < 2*88 bits. We should still address that, by marking small curves as not supported more generally and explicitly.

Updated Veridise Response As described in the developer response, this is no longer sound for small base fields. In particular, in V-O1J-VUL-033, `x` being almost reduced does not always imply that `x < 2f`. This equality check will be under-constrained for ECDSA over foreign fields with modulus less than 2^{176}, e.g. for secp128r1.

```
// check that x1 != x2
// we assume x1, x2 are almost reduced, so deltaX <= x1 - x2 + f < 3f
// which means we need to check that deltaX != 0, f, 2f
let deltaX = ForeignField.sub(x1, x2, f);
let deltaX01 = deltaX[0].add(deltaX[1].mul(1n << l)).seal();
assertNotVectorEquals([deltaX01, deltaX[2]], [0n, 0n]); // != 0
assertNotVectorEquals([deltaX01, deltaX[2]], [f0 + (f1 << l), f2]); // != f
deltaX[2].assertNotEquals(fx22); // != 2f (stronger check bc assuming deltaX < f
    doesn't harm completeness)
```

Snippet 4.11: The `x1 != x2` check.

Using this function with unsupported base fields should throw an error, and this expectation should be documented.

Updated Developer Response We added a check to prevent creating foreign elliptic curves over small fields or using the `Ecdsa.add()` function (which contains the above code snippet) over small fields.

4.1.8 V-O1J-VUL-008: Action state updated on empty action list

Severity	High	**Commit**	8dde2c3
Type	Under-constrained Circuit	**Status**	Fixed
File(s)	`src/lib/mina/zkapp.ts`		
Location(s)	reduce()		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1577`		

The `reduce()` function performs a sequence of actions, allowing for concurrent submission of requests. As described in V-O1J-VUL-005, the goal of `reduce()` is to perform the below for-loop on the provided action state

```
newActionDigest = processedActionsDigest
for( actionLists in submittedAndUnprocessedUpdates ):
    for (action in actionList ):
            newState = f(newState, action) // f() is the user-provided reduce()
    function
      newActionStateDigest = nextDigest(newActionStateDigest, actionList)

// Ensure all processed actions were actually submitted and therefore
// verified as valid action submissions
assert(newActionStateDigest == submittedActionsDigest)
```

Snippet 4.12: Pseudo-code for `reduce()`

The inner loop is unrolled 0 times, 1 time, 2 times, . . . , `maxActionsPerMethod` times. It is executed once for each unrolling, and then prover-provided `Bools` are used to select the "actual" execution.

```
// [VERIDISE] Prover-supplied Bools used to switch between different lengths
let lengths = possibleActionsPerTransaction.map((n) =>
  Provable.witness(Bool, () => Bool(length === n))
);
// create dummy actions for the other possible action lengths,
// -> because this needs to be a statically-sized computation we have to operate on
    all of them
let actionss = // [VERIDISE] elided for brevity

// for each action length, compute the events hash and then pick the actual one
let eventsHashes = // [VERIDISE] elided for brevity
let eventsHash = Provable.switch(lengths, Field, eventsHashes);
let newActionsHash = Actions.updateSequenceState(
  actionState,
  eventsHash
);
let isEmpty = lengths[0];
// update state hash, if this is not an empty action
actionState = Provable.if(isEmpty, actionState, newActionsHash);
// also, for each action length, compute the new state and then pick the actual one
let newStates = // [VERIDISE] elided for brevity
// update state
state = Provable.switch(lengths, stateType, newStates);
```

Snippet 4.13: Snippet from `reduce()`

In most cases, using the untrusted `lengths` variables is okay. This is because, as shown in the outer for-loop, the computed `actionState` is compared against the (Mina-managed, on-chain value) `contract.account.actionState` at the end of the outer for-loop. This should ensure that the prover-provided `actionList` is in fact a valid pre-image of the `contract.account.actionState`, which is a digest of the submitted actions.

There is one special case. If `lengths[0]` is `true` (i.e. `isEmpty` is `true`), then the `actionState` is *not* updated. Instead, the previous `actionState` is used. However, `state` switches on the *entire* `lengths` array of `Bools` to determine what the new state should be. By setting multiple `lengths` `Bools` to `true`, an attacker may use actions to update the `state` *without* having those actions affect the `actionState`. This gives the attacker near-total control of the computed `state`.

Impact By setting `lengths[0]` and `lengths[maxActionsPerMethod]` to `true`, the attacker may choose to insert `maxActionsPerMethod` arbitrary transactions into the "true" submitted action list. Since these actions may often directly correspond to access to funds (e.g. insertion of values into a Merkle tree representing rights to funds in a shielded pool), this can lead directly to theft or otherwise near-arbitrary breaking of protocol invariants.

Normally, this would be a critical bug. However, as mentioned in the developer response to V-O1J-VUL-005, actions are not yet intended to be supported. For that reason, we have downgraded the likelihood, leaving this as a high issue.

Recommendation To resolve this issue, state updates should be skipped whenever the action-state digest is not updated.

We also recommend renaming several variables to make their purposes more clear:

- Consider renaming `maxTransactionsWithActions` to `maxAccountUpdatesWithActions`.
- Consider using different names for the `actionState` representing the digest of processed actions and the `account.actionState` represent the digest of submitted actions.
- Consider changing the "event" terminology and replacing it with "actions" terminology.
- Consider using a different name than `reduce()` for the user-provided `reduce()` function.

Additionally, as recommended in V-O1J-VUL-005, all documentation and examples referencing actions/reducers should be updated to indicate that they are not safe for production.

Developer Response The reducer logic has significantly changed from the code where this issue was found. We must validate that the current logic does not have this issue.

Updated Veridise Response This seems correct. One portion of the code relies on an out-of-scope assumption. When `maxActionsPerUpdate` is 1, it must not be possible for an `AccountUpdate` to "emit" 0 actions, (i.e. `!isDummy` in the outer iterator must imply `!isDummy` in the inner iterator in this case).

Updated Developer Response Yes, if an account update has 0 actions then the on chain logic will not update the action state, i.e. skip the empty action list.

4.1.9 V-O1J-VUL-009: Non-CCA-secure encryption

Severity	High	**Commit**	8dde2c3
Type	Logic Error	**Status**	Partially Fixed
File(s)	`src/lib/provable/crypto/encryption.ts`		
Location(s)	encrypt(), decrypt()		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1729`		

The `encrypt()` function uses Poseidon in duplex mode. This is a well-known construction (see, for example, Section 5 from Duplexing the sponge: single-pass authenticated encryption and other applications or Chapter 4 of Cryptographic sponge functions).

```
P.absorb(sharedSecret)
for i=0,1,2,...,n/2:
    [p0, p1] := P.squeeze()
    c[2*i  ] = m[2*i  ] + p0
    c[2*i+1] = m[2*i+1] + p1
    P.absorb([c[2*i], c[2*i+1]])
tag = P.squeeze()
```

Snippet 4.14: Pseudocode of the algorithm used in `encrypt()`

One key difference between the above algorithm and the standard constructions is the lack of a "frame bit." In the standard construction, each plaintext field has a frame bit concatenated, indicating whether the field is the last or not.

For example, Algorithm 3 in Section 5 from Duplexing the sponge: single-pass authenticated encryption and other application breaks the message into fields of size 1-bit smaller than the field size, pads each non-terminal input field with a 1, and then the final field with a zero.

for $i = 0$ to $w - 1$ **do**
  $Z = D.\text{duplexing}(B_i||1, |C_{i+1}|)$
  $B_{i+1} = C_{i+1} \oplus Z$
end for
$Z = D.\text{duplexing}(B_w||0, \rho)$

The missing frame-bit makes this encryption scheme vulnerable to chosen-ciphertext attacks.

More concretely, observe that if `c[0]`, `c[1]`, . . . , `c[n]` is a valid ciphertext (note here that `c[n]` is the authentication tag of an `n`-field message `m[0]`, `m[1]`, . . . , `m[n-1]`), then `c[0]`, `c[1]`, . . . , `c[n-1]-m[n-1]` is *also* a valid ciphertext.

In particular, the scheme does not have ciphertext integrity.

Impact Any application for which

1. Uses the same public key for different message lengths
2. Has low-entropy messages (e.g. each 254 bits of a message has only 60 bits of entropy)
3. An attacker may determine (either directly or by some side channel) whether or not a given ciphertext is valid

may be vulnerable to a chosen ciphertext attack enabling attackers to decrypt large portions of messages.

This attack would work as follows:

```
cipherText := listenOnNetwork()
for i in range(numAttempts):
    possibleLastMessage := sampleLikelyMessageField()
    testCipherText := [c[0], c[1], ..., c[n-1] - possibleLastMessage]
    if successfullyDecrypts(testCipherText):
            print("Successfully decrypted last message block: ", possibleLastMessage)
```

Repeating the attack on the (now shorter) message allows decryption of the other message blocks.

We validated that this does in fact produce a valid ciphertext.

```
// message
let message = 'This is a secret which is exactly 3 fields long:' + '0'.repeat(40);
let messageFields = Encoding.stringToFields(message);
console.log(`${messageFields}`)

await Provable.runAndCheck(() => {
  // encrypt
  let cipherText = Encryption.encrypt(messageFields, publicKey);

  Provable.log(cipherText);
  // mess with cipherText
  cipherText.cipherText.pop()
  let lastIdx = cipherText.cipherText.length - 1
  cipherText.cipherText[lastIdx] = cipherText.cipherText[lastIdx].sub(messageFields[
    lastIdx]);
  Provable.log(cipherText);

  // decrypt: This should fail
  let decryptedFields = Encryption.decrypt(cipherText, privateKey);
});
```

Snippet 4.15: Example ciphertext manipulation. When using full authenticated encryption, `Encryption.decrypt()` should fail with extremely high probability on mutated ciphertexts.

Recommendation

1. Include a frame bit to indicate which message field is the final one.
2. Add additional documentation that the private key used in this method should never be reused. While the current implementation is correct, users who attempt to mutate the method to re-use the provided `privateKey` will leak information about `message[0]`.

Developer Response We now absorb tuples of size `(frame_bit, message_chunk)` to prevent these CCA attacks.

Updated Veridise Response This does appear to address the core issue. Three additional recommendations could improve security:

1. Algorithmic recommendation: Consider using overwrite mode (instead of adding messages directly to the hash output). This has been cryptanalyzed (see Sec 6.2 of `https://keccak.team/files/SpongeDuplex.pdf`) and may be cheaper.
2. The padding used is non-injective, consider encoding the length of the message into the padding.
3. Using a constant-time equality check with the authentication tag (by the prover) before the provable, non-constant time check would reduce the possibility of side channels.

4.1.10 V-O1J-VUL-010: Incorrect default for toBits()

Severity	Medium	Commit	8dde2c3
Type	Under-constrained Circuit	Status	Fixed
File(s)	src/lib/provable/foreign-field.ts		
Location(s)	ForeignField.toBits()		
Confirmed Fix At	https://github.com/o1-labs/o1js/pull/1617/		

The `ForeignField` class acts as a base class for its three variants: `Unreduced`, `AlmostReduced`, and `Canonical`. The `toBits()` method converts the `ForeignField` element to a sequence of bits, representing the binary decomposition of that foreign field element's representative.

In particular, the method allows a user-provided `length` parameter so that the user may optimize for values which are known to be bounded. For `toBits()` to properly constrain the `ForeignField` element, the representative must be constrained to be in the range $[0, 2^{\text{length}})$. The below code snippet shows the default value for `length`, `Constructor.sizeInBits` (i.e. $\lceil \log_2 p \rceil$ of the foreign field modulus p).

```
1 toBits(length?: number) {
2   const sizeInBits = this.Constructor.sizeInBits;
3   if (length === undefined) length = sizeInBits;
4   checkBitLength('ForeignField.toBits()', length, sizeInBits);
```

Snippet 4.16: Beginning of `ForeignField.toBits()` function.

However, in the most general case, the representative is only guaranteed to be in the range $[0, 2^{264})$. The above default is only valid for the `Canonical` variant, or `AlmostReduced` variants when the modulus is greater than $2^{176} - 1$ (see V-O1J-VUL-033).

Impact Users calling `toBits()` may create an under-constrained circuit.

For example, if the field modulus is 17, then only $5 = \lceil \log_2 17 \rceil$ bits will be returned. The low limb will be constrained to equal this 5-bit `ForeignField` representative. However, this can be manipulated by adding large multiples of 17 to the chosen representative (only altering the unconstrained high limbs).

Recommendation Ensure that the default `length` for `Unreduced` elements and `AlmostReduced` elements (with small moduli) is large enough to fully constrain the returned bits.

Developer Response The model we chose is that `toBits()` only works if the foreign field representation fits in that many bits, so in the case that the higher limbs are discarded, we assert that they must be zero.

4.1.11 V-O1J-VUL-011: MayUseToken fields may both be true

Severity	Medium	**Commit**	8dde2c3
Type	Data Validation	**Status**	Fixed
File(s)	`src/lib/mina/account-update.ts`		
Location(s)	isParentsOwnToken()		
Confirmed Fix At	`https://github.com/o1-labs/o1js-bindings/compare/5bf4a92ab7467412f9953bbc29d97bfd19230db0...563778c3b01a31f2b0960668b04e79ef30018c9e,https://github.com/o1-labs/o1js/pull/1750,https://github.com/o1-labs/o1js/pull/1741`		

The `MayUseToken` field of an `AccountUpdate` describes which token id the `AccountUpdate` is permitted to use.

```
type: provablePure({ parentsOwnToken: Bool, inheritFromParent: Bool }),
// ...
isParentsOwnToken(a: AccountUpdate) {
  return a.body.mayUseToken.parentsOwnToken;
},
isInheritFromParent(a: AccountUpdate) {
  return a.body.mayUseToken.inheritFromParent;
},
```

Snippet 4.17: Snippet from `AccountUpdate.MayUseToken`

Note, however, that if both flags are set to `true`, the inherited token is used rather than the parent's own token. This can be seen in the definition of "caller_id" found in `zkapp_command_logic.ml`. When `inheritFromParent` is `true`, `parentsOwnToken` is ignored.

In either case, the default token ID (1 for Mina) may be used when either flag is true.

Impact When iterating over an `AccountUpdateForest`, a parent `AccountUpdate` may call `child.isParentsOwnToken()`, but not check `child.tokenId`. If both `parentsOwnToken` and `inheritFromParent` are `true`, then the parent will treat the child as if it is using the parent's own derived token, when really it may be inheriting the token id or using the default token id.

While technically a misuse of the API, this can lead to severe issues like miscalculation of a total balance change by failure to consider certain `AccountUpdates`' effects properly during iteration.

Recommendation Check that `inheritFromParent` is not `true` when validating if an account update is for a parent's own token.

Consider adding a check to ensure the flags are not both set to true.

Developer Response We updated `check()` inside of o1js-bindings to validate that not both `inheritFromParent` and `parentsOwnToken` are set. Additionally, the Mina `account_update` constructor now parses the `MayUseToken` into one of the three allowed variants, erroring on the `{true, true}` case.

4.1.12 V-O1J-VUL-012: Asserting preconditions may overwrite previous assertions, no memory consistency

Severity	Medium	**Commit**	8dde2c3
Type	Race Condition	**Status**	Partially Fixed
File(s)	`src/lib/mina/precondition.ts`, `src/lib/mina/account-update.ts`		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1712`		

The `AccountUpdate` data structure allows users to provide several preconditions which must hold in order for the `AccountUpdate` to be valid. This is important to ensure that unexecuted transactions expire and that claimed blockchain state values in a circuit match the *actual* state values.

Performing assertions on an `AccountUpdate` typically corresponds to setting its preconditions fields. In contrast to in-circuit assertions, performing assertions more than once on the same precondition will overwrite any previously asserted preconditions.

For example, the below snippet shows the default `requireBetween()` method. Even if `property.isSome` is already set to `true`, this method will overwrite the previous value.

```
requireBetween(lower: any, upper: any) {
  context.constrained.add(longKey);
  let property: RangeCondition<any> = getPath(
    accountUpdate.body.preconditions,
    longKey
  );
  property.isSome = Bool(true);
  property.value.lower = lower;
  property.value.upper = upper;
},
```

Snippet 4.18: Definition of `requireBetween()` defined in `preconditionSubClassWithRange<>()`. This method is used to define the method for most of the account-update properties.

This may lead to confusing cases. For example, the below snippet shows a user attempting to constrain the timestamp to be in the interval $[0, 2)$ and the interval $[1, 3)$. However, instead of requiring the intersection, the final constraint on timestamp is simply the $[1, 3)$ interval.

```
timestamp.requireBetween(newUInt32(0n), new Uint32(2n))
timestamp.requireBetween(newUInt32(1n), new Uint32(3n))
```

Snippet 4.19: Example use case overwriting previous requires.

Similarly, `requireEquals()` also overwrites the previous value. This way, `a.requireEquals(b)` followed by `a.requireEquals(c)` implies $a = c$, but not $a = b$.

This pattern is repeated throughout `o1js/src/lib/mina`, including in

- account-update.ts
  - `AccountUpdate.requireSignature()`
  - `AccountUpdate.assertBetween()`

- `AccountUpdate.assertEquals()`
- `AccountUpdate.approve()` — this will overwrite other children in the case of receiving a `Forest`, rather than appending them.

▶ precondition.ts.

- `timestamp.requireBetween()` in `Network()`
- `preconditionSubclass().requireEquals()`
- `preconditionSubclass().requireNothing()`

▶ state.ts

- `StateAttachedContract.requireEquals()`
- `StateAttachedContract.requireNothing()`
- `StateAttachedContract.getAndRequireEquals()`

Impact If users set lower bounds separately from upper bounds, they may accidentally overwrite upper/lower bounds. Furthermore, multiple sequential equality assertions will lead to only the final constraint being checked.

Even worse, state variables read multiple times may discard the check against previous reads. For example, consider the below toy example which attempts to only allow an "admin" address to be set once.

```
export class Vault extends SmartContract {
  @state(PublicKey) admin = State<PublicKey>();

  init() {
    super.init();
    this.admin.set(PublicKey.empty());
  }

  @method async setup(newAdmin: PublicKey) {
    // prevent calling multiple times
    this.admin.requireEquals(PublicKey.empty());
    newAdmin.equals(PublicKey.empty()).assertFalse()

    // Set up admin and initialize with token supply (dummy value put here)
    this.admin.set(newAdmin);
    this._mint(newAdmin, new Field(10n **12n));

    // Perform some other checks, e.g. restrict the caller.
    // This pattern might be used if this contract is deployed before the admin,
    which
    // is also intended to be a smart contract.
    // ... REDACTED
  }

  check() {
    let admin = this.admin.getAndRequireEquals()
    // Check signature, or if initializing just default to true
    Provable.if(admin.equals(PublicKey.empty()), Bool,
      new Bool(true),
      this.sender.getAndRequireSignature().equals(admin)
```



```
    ).assertTrue();
  }

  _mint(receiver: PublicKey, value: Field) {
    this.check();
    // update state to record minting value to receiver...
  }

  @method async mint(receiver: PublicKey, value: Field) {
    this._mint(receiver, value);
  }

}
```

Snippet 4.20: Toy example

`setup()` calls `requireEquals()` on `admin` to ensure it has not been set yet. However, since `setup()` calls `_mint()`, which also calls `admin.getAndRequireEquals()`, this initial `requireEquals()` is ignored. The `admin` may then call `setup()` many times in order to mint more for themselves.

While this toy example is a bit contrived, since here the `admin` has full permission to mint by just calling `mint()`, this pattern in the wild could lead to rather devastating results by allowing users to silently discard assertions. The further apart the assertions are in the codebase, the more difficult this will be to find. While not likely to manifest in simple applications, this effect could have incredibly severe consequences.

Finally, note that `set()` does not change any in-circuit values. Since reads (i.e. calls to `get()`) are cached, there is **not** memory consistency between `get()`s and `set()`s.

Recommendation When setting a precondition, throw an error if it has already been set.

When `get()`ting a value, throw an error if `set()` has been called on that value.

Developer Response The library now throws an error if a precondition has already been set, rather than overwriting the previous precondition. This can be avoided by calling the unsafe `requireNothing()` function.

At this point in time, we have decided not to change the behavior of `get()/set()`.

4.1.13 V-O1J-VUL-013: Improperly constrained scale()

Severity	Medium	**Commit**	8dde2c3
Type	Under-constrained Circuit	**Status**	Fixed
File(s)	`src/lib/provable/group.ts`		
Location(s)	scale()		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1759`		

The `Group` and `Scalar` classes abstract elements of the Pallas curve and its scalar field (Vesta), respectively. `Group` points are represented as a standard `(x,y)`-coordinate pair, with the point at infinity $\mathcal{O}$ represented as `(0,0)`. `Scalars` are represented as a 255-length array of `Fields`, each associated to one bit of the scalar.

The interpretation of a `Scalar`'s underlying array `b = [b[0], b[1], ..., b[254]]` is as follows:

1. Define `k` to be the element of the Vesta field with big-endian representation `b`, i.e. $k = \sum_{i=0}^{254} 2^{254-i} b[i]$.
2. Define `s = 2^{255} + 2k + 1`.

The `Scalar` array `b` represents the field-value `s`.

This representation is motivated by an improvement made to ZCash described in this PR: "Faster variable-base scalar multiplication in zk-SNARK circuits", written by Daira-Emma Hopwood. To compute the value $[2^n + 2 \cdot k + 1] \cdot T$ for some `Group`-element T, the following algorithm computes the desired value into the variable `Acc`.

```
Acc := [2] T
for i from n-1 down to 0 {
    Q := k_i ? T : -T
    Acc := (Acc + Q) + Acc
}
```

In the above code, `k_i` refers to the `i`th-least significant bit of the `n+1`-bit representation of `k`. Rewriting this algorithm using our above notation, we have

```
n := 255
Acc := [2] T
for (let iter = 0; iter < n ; ++iter) {
    Q := b[iter] ? T : -T
    Acc := (Acc + Q) + Acc
}
```

Conceptually, each iteration of the above loop computes $(P, T) \rightarrow 2P \pm T$. The loop then implements a double-and-add scalar multiplication scheme. The kimchi varbasemul gate implements 5 iterations of this loop. A driver inside of pickles repeats the gate until the required number of iterations have been applied.

Now, the linked kimchi implementation (documented here) performs group addition with formulas assuming that the two supplied operands have unequal `x`-coordinates. In particular, it assumes that `Acc != +/-Q` and `Acc + Q != +/-Acc`.

Problems may arise if, at any point during iteration, either

- `Acc == +/-T` (assuming `T != 0` and therefore `Q!=0`)

or

- `2 * Acc == -Q`. Assuming `Q` is non-zero, this could only occurs when `Acc + Q == -Acc`. Note that, unlike the `Acc == +/-T` case in which `Acc` may match either `Q` or `-Q`, in this case `Acc` must match `Q` exactly. See the Details section at the end of this issue for a more thorough discussion.

Reaching either of these cases corresponds to a divide-by-zero error which may leave the circuit under-constrained. However, as pointed out by the o1js developers, the `2 Acc = -Q` case is never unsound, because it corresponds to `(Acc + Q) = -Acc` where the two points to be added are inverse (not equal) to each other, so the numerator is non-zero. Thus, the `2 * Acc == -Q` case will only cause an over-constrained circuit.

Conveniently, the linked PR has already provided us with an expression for `Acc` at the start of each iteration. Recall that the entire loop computes $[2^n + 2 \cdot k + 1] \cdot T$. Stopping the loop earlier corresponds to the same computation, but with a "smaller n" and ignoring some of the low-bits of k. More precisely

- `iter = 0`: `Acc = [2] T`
- `0 < iter <= n`: Let `kIter` be the top `iter` bits of `k`, i.e. `kIter := k >> (n-iter)`. Then, `Acc = [2**iter + 2*kIter + 1] T`.

Now we may analyze modulo the Vesta modulus `q`, since `[s] T == +/-T` if and only if `s == +/-1 mod q`. We consider the possible error cases below:

1. `iter = 0`: In this case, `Acc = [2] T`.
   a) `Acc == +/-T`: `[2] T == +/-T` only if `T == 0` or `[3] T == 0`. Since `3 != q` and we are assuming `T ! = 0`, this cannot occur.
   b) `2 * Acc == -Q`: `[4] T == +/-T` only if `[5] T == 0` or `[3] T == 0`. Again assuming `T != 0`, this cannot occur.
2. `0 < iter < n`: In this case, `Acc = [2**n + 2*kIter + 1] T`.
   a) `Acc == +/-T`: This case holds when `2**iter + 2*kIter + 1 == +/-1 mod q`.
   b) `2 * Acc == -Q`: This case holds when `2**iter + 2*kIter + 1 == +/- 2^{-1} mod q` (where by `2^{-1}` we mean the inverse of 2 modulo `q`).

To search for candidate error-values, we must find values `iter` and `kIter` for which:

1. `0 < iter <= n`
2. `2**iter + 2*kIter + 1` is either `1`, `-1`, `2^{-1}`, or `-2^{-1}` modulo `q`.
3. `0 <= kIter < 2**iter`

Then, for any `remainder < 2**(n-iter)`, running the above loop with `k := 2**(n-iter) kIter + remainder` should lead to a summation of two values with equal `x`-coordinates at iteration `iter`.

Note this search may be easily performed, since for each of the `n` possible values of `iter`, we may compute `kIter` directly by solving the equation in (2.), and then check criteria (3.) to see if it is a valid value.

```python
def egcd(a, b):
    """ Extended Euclidean Algorithm to find the GCD and coefficients x and y such
    that ax + by = gcd(a, b) """
    if a == 0:
        return (b, 0, 1)
    else:
        g, x, y = egcd(b % a, a)
        return (g, y - (b // a) * x, x)

def modinv(a, m):
    """ Find the modular inverse of a under modulus m """
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('Modular inverse does not exist')
    else:
        assert x * a % m == 1  # sanity check
        return x % m

def find_k(q, n=255):
    results = []

    oneHalf = modinv(2, q) % q
    targetScalars = [q-1, 0, 1, oneHalf, q-oneHalf]
    names = ["-1", "0", "1", "1/2", "-1/2"]

    for iter_ in range(n + 1):
        power_2iter_mod_q = pow(2, iter_, q)
        # Calculate for 2^iter_ + 2kIter + 1 \equiv targetScalar mod q
        for targetScalar, name in zip(targetScalars, names):
            kIter = (q - power_2iter_mod_q + targetScalar-1) * oneHalf % q
            assert (2**iter_ + 2*kIter + 1) % q == targetScalar

            if kIter < 2**iter_:
                k = kIter*2**(n-iter_)
                results.append((iter_, k, name))

    return results

q = 0x4000000000000000000000000000000224698fc0994a8dd8c46eb2100000001
results = find_k(q)

print("q:")
print(q)
for result in results:
    print(result)
```

Snippet 4.21: Python code to search for problematic values.

As a sanity check, we include checks for `k`'s which map to an `s`-value of `0`. These should correspond to `k`-values which hit the `2 * Acc == -Q` case in a previous iteration. The results are shown below

```
# Potential problem at 4 of the 8 next values
(252, 2894802230932904885589274625217197696340861679747315374949508949480844348620 8,
```



```
    '1/2')
# sanity check: corresponds to a problem in iteration 252
(253, 2894802230932904885589274625217197696340861679747315374949508949480844348620 8,
    '0')
# Potential problem at 2 of the 4 next values
(253, 455603155315063698153467464150805381 12, '1/2')
# Potential problem at 2 values
(254, 2894802230932904885589274625217197696340861679747315374949508949480844348620 8,
    '-1')
  # sanity check: corresponds to a problem in iteration 253
(254, 455603155315063698153467464150805381 12, '0')
# Potential problem at 2 values
(254, 2894802230932904885589274625217197696340861679747315374949508949480844348621 0,
    '1')
# Potential problem at 1 of the next 2 values
(254, 434220334639935732838391193782579654450901450384439774393349608690051249602 58,
    '1/2')
# Potential problem at 1 of the next 2 values
(254, 144740111546645244279463731260859884817270885565023300596552181206117620121 60,
    '-1/2')

# Sanity checks: correspond to issues in previous iterations
(255, 455603155315063698153467464150805381 12, '-1')
(255, 144740111546645244279463731260859884817270885565023300596552181206117620121 61,
    '0')
(255, 455603155315063698153467464150805381 13, '1')
# Irrelevant: iteration 255 is not actually executed
(255, 723700557733226221397318656304299424088632443601691821473528243351342127513 7, '
    1/2')
(255, 217110167319967866419195596891289827225678526769877419045751538077101027491 85,
    '-1/2')
```

Summarizing the above output, there are twenty possible values for `k` at which the circuit may be under-constrained. Testing `Group.scale()` for each of the above `k`s, we find that 12 of the 20 fail with the error `Field.inv: zero`, indicating a divide-by-zero issue in the out-of-scope Pickles code.

```
s   | k < q? | k
             | Group.scale() Fails?
-------------------------------------------------------------------------------------------------------------------
-3n | false  |
    28948022309329048855892746252171976963408616797473153749495089494808443486208n |
        Field.inv: zero
-1n | false  |
    28948022309329048855892746252171976963408616797473153749495089494808443486209n |
        Field.inv: zero
1n  | false  |
    28948022309329048855892746252171976963408616797473153749495089494808443486210n |
        Field.inv: zero
3n  | false  |
    28948022309329048855892746252171976963408616797473153749495089494808443486211n |
        Field.inv: zero
5n  | false  |
```

```
    28948022309329048855892746252171976963408616797473153749495089494808443486212n |
                -
8   7n  | false  |
    28948022309329048855892746252171976963408616797473153749495089494808443486213n |
                -
9   9n  | false  |
    28948022309329048855892746252171976963408616797473153749495089494808443486214n |
                -
10  11n | false  |
    28948022309329048855892746252171976963408616797473153749495089494808443486215n |
                -
11  -1n | true   | 455603155315063698153467464150805381l2n
                |     Field.inv: zero
12  1n  | true   | 455603155315063698153467464150805381l3n
                |     Field.inv: zero
13  3n  | true   | 455603155315063698153467464150805381l4n
                |            -
14  5n  | true   | 455603155315063698153467464150805381l5n
                |            -
15  -3n | false  |
    28948022309329048855892746252171976963408616797473153749495089494808443486208n |
        Field.inv: zero
16  -1n | false  |
    28948022309329048855892746252171976963408616797473153749495089494808443486209n |
        Field.inv: zero
17  1n  | false  |
    28948022309329048855892746252171976963408616797473153749495089494808443486210n |
        Field.inv: zero
18  3n  | false  |
    28948022309329048855892746252171976963408616797473153749495089494808443486211n |
        Field.inv: zero
19  0n  | false  |
    43422033463993573283839119378257965445090145038443977439334960869005124960258n |
        Field.inv: zero
20  2n  | false  |
    43422033463993573283839119378257965445090145038443977439334960869005124960259n |
                -
21  -2n | true   |
    14474011154664524427946373126085988481727088556502330059655218120611762012160n |
                -
22  0n  | true   |
    14474011154664524427946373126085988481727088556502330059655218120611762012161n |
        Field.inv: zero
```

Note that, as discussed above (and described in full detail in the "Details" section), only "half" of the `k`s fail in the `2 * Acc == T` case, so the 12 of 20 failures is exactly as expected.

Additionally, note that most of these representations of `k` are non-canonical (i.e. `k >= q`), but still fit within the required 255 bits. The possible scalar values (i.e. `s`) which may cause a failure are `-3`, `-1`, `0`, `1`, and `3`.

Impact Someone may create a scalar in `o1js` with one of these "bad" `k`'s using the following snippet.


```
let scalar = Provable.witness(Scalar, () => {
    let fqBits = toBits(badKs[i])
  let scalar = Scalar.from(reducedValue);
  scalar.value = fqBits;
  return scalar;
});
```

Snippet 4.22: Example code creating a malicious `Scalar` value

If the user also alters the witness generator in pickles, they may abuse the divide-by-zero error to change the result of the scalar multiplication.

However, as described above this only applies the the `Acc == +/-T` case. In the `1/2` or `-1/2` case, the circuit is only over-constrained. Thus, only four of the above points are relevant for an under-constrained attack. This could lead to a user completely controlling the output of $s \cdot T$ for $s \in \{-1, 0, 1\}$. Additionally, the $T = \mathcal{O}$ case appears to be entirely unhandled, and to leave large portions of the gates improperly constrained.

For this to work, the attacker must either be able to set T equal to the identity, or control the value of s. This is *not* the case in the most prominent example of Schnorr signature verification. However, a user could use a private key from $\{-1, 0, 1\}$ to control the result of a Schnorr signing operation. This is likely less impactful, as most applications which sign values in-circuit do so to protect the signer's own funds or privacy. A similar situation holds for the implemented nullifiers.

Recommendation

1. Handle the $T = \mathcal{O}$ case separately in the `scale()` function.
2. Review the above analysis to determine if the o1js developers agree with Veridise's conclusion. Given the complicated nature of this issue, we also recommend adding thorough documentation to the `scale()` operation indicating at which points it may be under-constrained.

Finally, consider handling the above twelve cases separately, or constraining the denominators in the Kimchi gate to be non-zero.

Developer Response Let's first address soundness separately from completeness.

Say an in-circuit division `y/x = z` is implemented by constraining `y = x*z`. This is only unsound if both `y=0` and `x=0`. If `x=0` but we can guarantee that `y != 0`, then the constraint will fail because of equating 0 with something non-zero, so it's not unsound.

With this reasoning, we can come to the conclusion that the `2 Acc = -Q` case is never unsound, because it corresponds to `(Acc + Q) = -Acc` where the two points to be added are inverse (not equal) to each other, so the numerator is non-zero. And in fact, in the simplified expressions that the gate uses, the numerator is `2y_i` which can't be 0.

Regarding completeness, it's a bit involved to write out the full analysis here, but it should suffice to say that a PR added after this investigation handles all the incomplete cases explicitly and, empirically, scalar multiplication can now be run successfully without constraint failures for all scalars in -10,. . . ,10.

A final remark regarding completeness: There, we only have to handle cases where `k < p/2` (smaller than half the native modulus), which is significantly smaller than `2^254`, because the witnesses we generate will in practice ensure that (and we do not care about completeness for malicious witness inputs).

Updated Veridise Response We have updated the above writeup based on the o1js team's review.

Updated Developer Response I'm no longer convinced of the statement I made in the prior comments: "we do not care about completeness for malicious witness inputs."

Actually, we do care about that, since part of our smart contract system relies on being able to write circuits which

- process inputs from users in the form of actions
- will never produce failing constraints on these inputs

Currently, I believe our implementation would fail to produce a proof on non-canonically encoded scalars where `k = -2^254 - 1/2`. If someone would post such a scalar as part of an action, and the reducer logic involved scaling by that scalar, a malicious user could break the reducer.

Updated Veridise Response This is a very good point, and could also show up in other use cases such as `ForeignField` (e.g. with the operations described in V-O1J-VUL-040), or if users are providing a key to a `MerkleMap` with the high-bit set to `1` (see the fix for V-O1J-VUL-002).

Given that this can be very tricky, it might be good to consider some additional mitigations including:

1. Adding extra docs/examples of this potential issue
2. Whitelisting classes for use in `ActionTypes`. This could be done by adding some reserved keyword/symbol the type.

Updated Developer Response We have added a `toCanonical()` method to provable types. This is a sound and complete map to a canonical representation, usually the identity function. However, for types with multiple valid representations this will map to a canonical form. For example, foreign field elements will map to their reduced representation.

This method is now used inside the `Provable.equals()` call and also for scaling.

Details To understand why only some of these lead to actual failures, we must dig deeper into the Kimchi gate. The Kimchi gate performs each iteration of the loop (i.e. computing $2P \pm Q$, as mentioned above) by computing $(P + (2b - 1)T) + P$. The Kimchi gate adapts the standard curve formulas for a=0, improving efficiency by combining the two additions. This avoids needing to compute the intermediate "output" gate. So, writing $T = (x_t, y_t)$ and $P = (x_i, y_i)$, Kimchi computes the result $(x_o, y_o) = (P + (2b - 1)T) + P)$ as shown below.

$$
\begin{aligned}
s_1 &:= \frac{y_i - (2 \cdot b - 1) \cdot y_t}{x_i - x_t} \\
s_2 &:= \frac{2 \cdot y_i}{2 * x_i + x_t - s_1^2} - s_1 \\
x_o &:= x_t + s_2^2 - s_1^2 \\
y_o &:= s_2 \cdot (x_i - x_o) - y_i
\end{aligned}
$$

Note that the divisions are computed (as is typical in ZK-circuits) by multiplying both sides and constraining equality. Thus, when denominators are zero, the circuit is under-constrained. The `Field.inv: zero` error above comes from a witness-generation error. Consequently, there may be errors when either:

- $x_i = x_t$: this corresponds to the `Acc == +/-T` case above.
- $2x_i + x_t = s_1^2$: Let $(x_{mid}, y_{mid}) := P + (2b - 1)T$. In this case, we may assume $x_i \neq x_t$. Therefore, using the standard curve formulas for a=0, we have.

  $$s_1^2 = x_{mid} + x_i + x_t$$

  Therefore, $2x_i + x_t = s_1^2$ exactly when $x_i = x_{mid}$, i.e. when $P + (2b - 1)T$ and P have the same x-coordinate, i.e. $P + (2b - 1)T = \pm P$. Assuming $T \neq 0$, this occurs only when $P + (2b - 1)T = -P$. Consequently, an issue arises when

  $$(2b - 1)T = -2P$$

  This corresponds to the `2 * Acc == -Q` case, where b is the bit in the next iteration. As can be seen above, only *one* choice of `b`.

4.1.14 V-O1J-VUL-014: Inheritance should not be supported for in-circuit values

Severity	Medium	**Commit**	8dde2c3
Type	Logic Error	**Status**	Fixed
File(s)	`src/lib/provable/types/circuit-value.ts`, `src/lib/provable/types/struct.ts`		
Location(s)	check()		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1707`		

`check()` is a static method that can be used to make sure an in-circuit value is correctly constrained.

When `check()` is not explicitly given in a circuit class (extending either a `CircuitValue` or a `Struct`), the default implementation of the static method is to recursively call `check()` on fields of the class. More specifically, a field of type `propType` will be checked using `propType.check()`.

```
  static check<T extends AnyConstructor>(this: T, v: InstanceType<T>) {
    const fields = (this as any).prototype._fields;
    if (fields === undefined || fields === null) {
      return;
    }
    for (let i = 0; i < fields.length; ++i) {
      const [key, propType] = fields[i];
      const value = (v as any)[key];
      if (propType.check === undefined)
        throw Error('bug: CircuitValue without .check()');
      propType.check(value);
    }
  }
```

Snippet 4.23: Snippet from the default implementation of `check()` for `CircuitValue` based classes

However, because `check()` is a *static* method, it will always be associated with the *exact* provided circuit class, ignoring inheritance. As a result, if a circuit class is further extended with a more specific `check()`, but `propType` is from a parent class, then the more specific `check()` will not be called.

As a concrete example, consider:

```
class WrappedStruct extends Struct({
  wrappedA: Struct,
}) {}

class WrappedBool extends Struct({
  wrappedB: Bool
}) {}

let notBool = new Bool(false);
notBool.value = (new Field(2n)).value; // set to a non-bool

let wrappedBool = new WrappedBool({
  wrappedB: notBool
```

```
});
let wrappedWrappedBool = new WrappedStruct({
  wrappedA: wrappedBool
});

WrappedStruct.check(wrappedWrappedBool);
```

Here, we construct an invalid boolean value, and put it inside `WrappedStruct`. Calling `WrappedStruct.check` should be able to detect the invalid boolean value.

However, because `wrappedA` has type `Struct`, `Struct`'s `check()` is used, ignoring the `check()` method associated to `WrappedBool` (and thus `Bool`).

Impact This issue could arise due to incorrect usage by a developer who thinks that the inheritance chain is respected. Alternatively, an attacker could attempt to exploit this issue by intentionally specifying a weak type to backdoor the circuit.

In either case, when an appropriate `check()` is not called, constraints could be missing, causing the circuit to be under-constrained.

Recommendation We recommend that inheritance on circuit value classes should not be supported (except a direct inheritance from `Struct` and `CircuitValue` themselves). Detecting such cases and erroring could prevent users from having constraints be silently dropped.

At the very least, this should be clearly documented. Furthermore, it should be documented that the provided type of a field must be neither `Struct` nor `CircuitValue`.

Developer Response We explicitly disallowed using `Struct` as a field of a `Struct` and added documentation describing the issue.

4.1.15 V-O1J-VUL-015: isPositive() underconstrained at 0

Severity	Medium	**Commit**	8dde2c3
Type	Under-constrained Circuit	**Status**	Fixed
File(s)	`src/lib/provable/int.ts`		
Location(s)	Int64.isPositive()		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1660`, `https://github.com/o1-labs/o1js/pull/1735`, `https://github.com/o1-labs/o1js/pull/1742`		

`Int64`s represent signed integers as a `sign: Sign` and `magnitude: UInt64`. This allows for many efficient implementations. However, it also means that `0` has two representations: `+0` and `-0`.

This leads to ambiguity in the `isPositive()` function, which ignores the magnitude.

```
isPositive() {
  return this.sgn.isPositive();
}
```

Snippet 4.24: Definition of `isPositive()`.

Impact A malicious prover may choose to use either positive or negative zero.

Most arithmetic operations on `Int64`s use `fromField` internally (shown below). `fromField()` merely asserts that `sign * magnitude == x` (for some valid `sign` and `magnitude`). This means that, after any arithmetic operation which results in 0, the attacker may choose `+1` or `-1` for the sign, making this attack very easy to run.

```
static fromField(x: Field): Int64 {
  // constant case - just return unchecked value
  if (x.isConstant()) return Int64.fromFieldUnchecked(x);
  // variable case - create a new checked witness and prove consistency with original
     field
  let xInt = Provable.witness(Int64, () => Int64.fromFieldUnchecked(x));
  xInt.toField().assertEquals(x); // sign(x) * |x| === x
  return xInt;
}
```

Snippet 4.25: Definition of `fromField()`

For example, `Int64.mod()` uses the expression `Provable.if(this.isPositive(), rest, y_.value.sub(rest))` to select between `rest` and `y_ - rest` when computing the remainder modulo `y_`. By choosing `this` to `-0`, an attacker may output `y_` instead of `0`.

This could also impact protocols using `Int64`s to track PNL by allowing users to prove that `0` is positive.

Recommendation The simplest fix is to constrain 0 to a single representation. One way to do this is to extend `Sign` from `-1 | 1` to `-1 | 0 | 1`, and require `sign == 0` exactly when `magnitude == 0`.

One alternative is to add constraints to the `this == 0` case of `Int64.mod()`, document this ambiguity, and change the name of `isPositive()` to `isDefinitelyPositive()`.

Note that if a canonical representation of 0 is not required, this may have other surprising effects. For example, `+0.equals(-0)` will return `true`, but `+0.toString().equals(-0.toString())` may return `false`.

Developer Response Fixed in the PR https://github.com/o1-labs/o1js/pull/1660.

Updated Veridise Response We recommend the following additional changes:

- Consider adding checks to avoid creating a malformed `Int64` via computation.
  - `constructor` (https://github.com/o1-labs/o1js/blob/a8b9b5e761599a8168ec858b3671ffbd1e771fe0/src/lib/provable/int.ts#L1051) allows `new Int64(UInt64zero, SignMinusOne)`
  - `fromField` (https://github.com/o1-labs/o1js/blob/a8b9b5e761599a8168ec858b3671ffbd1e771fe0/src/lib/provable/int.ts#L1139) allows the witnesses to take `(UInt64zero, SignMinusOne)` for `0`.
    - As a result, `add`, `sub`, `mul` may still lead to invariant violations.
- `div()` may need additional changes:
  - Consider `(5, -1)` divided by `(100, 1)`. Because the resulting sign is the multiplication of the input signs, we will have `new Int64(UInt64zero, SignMinusOne)`, which does not follow the invariant.
- Consider changing the terminology to be consistent across functions:
  - `constructor` (https://github.com/o1-labs/o1js/blob/a8b9b5e761599a8168ec858b3671ffbd1e771fe0/src/lib/provable/int.ts#L1063) uses `positive` in a sense that it is `>= 0`.
  - `isPositive` (https://github.com/o1-labs/o1js/blob/a8b9b5e761599a8168ec858b3671ffbd1e771fe0/src/lib/provable/int.ts#L1261) uses `positive` in a sense that it is `> 0`.
- Ensure that the documentation of `negV2` and `modV2` do not refer to `neg` and `mod`.

Updated Developer Response There are two parts to the fixes. First is a backwards compatible change, that deprecates and changes existing functions: https://github.com/o1-labs/o1js/pull/1735. This PR

- Deprecates the constructor and adds a safe construction routine (`create()`).
- Adds a `Unsafe.fromObjectV2()` and deprecates the old `Unsafe.fromObject()`.
- Adds thorough documentation.

Second, we have the breaking changes here, that updates everything on a new release: https://github.com/o1-labs/o1js/pull/1742. This PR

- Updates the `check()` function to enforce `0` with sign `+1`.
- Provides methods for checking negativity, non-negativity, and positivity.

4.1.16 V-O1J-VUL-016: Provable.equals() unsound for some types

Severity	Medium	**Commit**	8dde2c3
Type	Logic Error	**Status**	Partially Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1759`		

For two instances `x` and `y` of a provable type, `Provable.equals(x, y)` returns a `Bool` which should be true when `x` is equal to `y`, and false otherwise. To do this, `Provable.equals()` uses the provable `toFields()` method on `x` and `y`, then returns true when all the fields are equal.

```
let xs = x.toFields();
let ys = y.toFields();
checkLength('Provable.equal', xs, ys);
return xs.map((x, i) => x.equals(ys[i])).reduce(Bool.and);
```

Snippet 4.26: Implementation of the `Provable.equals()` equality check.

This method is certainly sound: if it returns `true`, then `x` and `y` must represent the same in-circuit value. However, the converse only holds if every valid instance of a provable type has only one representation in fields.

The most prominent examples violating this property come from finite fields. A value x in a finite field $\mathbb{F}_p$ may be represented as $x + kp$ for any integer k. However, one would consider any two of these representations to be "equal" under the field's equivalence relation. This problem is described for the `ForeignField.assertEquals()` function in V-O1J-VUL-057, which recommends a name-change.

Several provable types may have multiple valid representations of a single instance:

- `UnreducedForeignField`: This case is described above.
- `AlmostForeignField`: Note the `AlmostForeignField.equals()` checks true field equality, while `Provable.equals(AlmostForeignField, x, y)` checks equality of the representative.
- `Scalar`: Elements of this class correspond to members of the Vesta field $\mathbb{F}_q$, and are represented using 255 bits. Since $q \approx 2^{254}$, most of its elements have two valid 255-bit representations.
- Until V-O1J-VUL-015 is fixed, `Int64` has two representations of `0`.

Note that types derived from the above types such as `ForeignCurve` points, `EcdsaSignature`, and `PrivateKey` also share this property.

Impact In some cases, there may be semantic differences between `type.equals(x,y)` and `Provable.equals(type, x, y)`. This can allow malicious users to "prove" that two equal points are not equal.

The most severe consequence of this could be in replay protection, allowing someone to perform restricted actions more than once if desired. However, this would require a very specific design, as most users rely on a mapping of some kind for replay protection.

Recommendation Require provable types to override the `equals()` method. Use that method inside of `Provable.equals()`.

The current `Provable.equals()` implementation may act as a sensible default for a provable type's `equals()` function.

Developer Response We have added a `toCanonical()` method to provable types. This is a sound and complete map to a canonical representation, usually the identity function. However, for types with multiple valid representations this will map to a canonical form. For example, foreign field elements will map to their reduced representation.

Updated Veridise Response `ForeignField.toCanonical()` may not be fully complete as mentioned in the documentation.

The output satisfies `xR = 1 * x - q * f` with `q` almost-reduced. For large `f`, this works out. For small `f`, we have `q < 2^3l`, so `xR = x - qf > x - 2^2l * f`, so for `x > 2^2l * f` it may be over-constrained. This should not cause a direct security problem, since it will still fail during dispatch, but may be worth updating the comment so people are aware of the weird corner case

Updated Developer Response Just adding that ForeignField really shouldn't be used for fields smaller than 127 bits, because in that case you can just multiply two elements without overflow directly, i.e. prove foreign field multiplication with a single equation in the native field; and the whole machinery of foreign fields is superfluous.

4.1.17 V-O1J-VUL-017: Missing canonical form check in verifyEcdsa

Severity	Medium	**Commit**	8dde2c3
Type	Logic Error	**Status**	Fixed
File(s)	`src/lib/provable/gadgets/elliptic-curve.ts`		
Location(s)	verifyEcdsa()		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1737/files`		

According to the full public key validation Appendix D.1 of NIST Recommendations for Discrete Logarithm-based Cryptography, the following checks should be performed to determine that a signature is valid.

- (r,s) are canonical
- (r,s) is not the identity
- (r,s) is on the curve
- (r,s) is in the subgroup

```
// reduce R.x modulo the curve order
let Rx = ForeignField.mul(R.x, Field3.from(1n), Curve.order);

// we have to prove that Rx is canonical, because we check signature validity based
   on whether Rx _exactly_ equals the input r.
// if we allowed non-canonical Rx, the prover could make verify() return false on a
   valid signature, by adding a multiple of 'Curve.order' to Rx.
ForeignField.assertLessThan(Rx, Curve.order);

return Provable.equal(Field3.provable, Rx, r);
```

Snippet 4.27: Snippet from `verifyEcdsa()`

In the function `verifyEcdsa()`, however, `s` is not checked to be canonical.

Impact This could allow an invalid signature to be incorrectly verified as valid by supplying a non-canonical representation of `s`.

Usually the only signature malleability is computed by flipping `s`. For example, OpenZeppelin enforces non-malleability by checking that `s` is in the lower half of the Secp256k1 base field. Without a canonical check, this will not be the case. This may lead to user confusion, or serious errors if an application tracks signatures to avoid replays (note that using `ForeignField.assertEquals()` will not reduce modulo the foreign field).

Recommendation A simple fix would be to assert that `r` and `s` are in the canonical form. This slightly diverges from the spec, because invalid (r, s) will cause an assertion failure rather than returning false. However, it would be easy to implement. In this case, the function should clearly document the deviation from the spec.

A more complete fix would be to properly check that `r` and `s` are in the canonical form, and if not, return false. Note also that `Rx` is asserted to be canonical, implying that the returned value will be false when `r` is non-canonical. This is correct, but may confuse readers at first glance. We recommend documenting the additional distinction here, i.e. that a non-canonical `Rx` is due to

an "incorrect computation" and should throw an error, while a non-canonical `r` is an invalid signature, and should return `false`.

Developer Response We modified the implementation to assert false if `s` is not canonical, and return false if `r` is not canonical

4.1.18 V-O1J-VUL-018: Incorrect permissions for setVerificationKey

Severity	Low	**Commit**	8dde2c3
Type	Authorization	**Status**	Fixed
File(s)	`src/lib/mina/account-update.ts`		
Location(s)	Permissions		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1639`		

The permissions for `dummy` and `impossible` are set incorrectly for `setVerificationKey`'s `auth`.

```
// [ELIDED - VERIDISE]
dummy: (): Permissions => ({
    // [ELIDED - VERIDISE]
    setVerificationKey: {
      auth: Permission.signature(),
      txnVersion: TransactionVersion.current(),
    },
    // [ELIDED - VERIDISE]
}),

allImpossible: (): Permissions => ({
    // [ELIDED - VERIDISE]
    setVerificationKey: {
      auth: Permission.signature(),
      txnVersion: TransactionVersion.current(),
    },
    // [ELIDED - VERIDISE]
}),
// [ELIDED - VERIDISE]
```

Snippet 4.28: Snippet from `Permissions`

Recommendation Fix the permissions as appropriate

Developer Response We added two new types of permissions: `impossibleDuringCurrentVersion` and `proofDuringCurrentVersion`. Additionally, we fixed the permissions and documented how `setVerificationKey` should be treated to avoid the problem with a hard fork.

4.1.19 V-O1J-VUL-019: Incorrect usage of parametric polymorphism

Severity	Low	**Commit**	3c68a0d
Type	Logic Error	**Status**	Partially Fixed
File(s)	`o1js-bindings/lib/provable-generic.ts`		
Location(s)	provable(), signable()		
Confirmed Fix At	`https://github.com/o1-labs/o1js-bindings/pull/273`		

The functions `provable()` and `signable()` are generic. From the type checker's perspective, `typeObj` can have any type, denoted `A`. However, the functions only support certain kinds of `typeObj`s. For this reason, unbounded parametric polymorphism permits invalid values. Instead, type bounds should be added to precisely indicate the kind of values that `typeObj` could be.

For example, `toFields()` has two important implicit assumptions for objects:

1. the keys in `typeObj` and `obj` are identical.
2. there are no `undefined` values.

```
function toFields(typeObj: any, obj: any, isToplevel = false): Field[] {
  if (primitives.has(typeObj)) return [];
  if (!complexTypes.has(typeof typeObj))
    throw Error(`provable: unsupported type "${typeObj}"`);
  if (Array.isArray(typeObj))
    return typeObj.map((t, i) => toFields(t, obj[i])).flat();
  if ('toFields' in typeObj) return typeObj.toFields(obj);
  return (isToplevel ? objectKeys : Object.keys(typeObj))
    .map((k) => toFields(typeObj[k], obj[k]))
    .flat();
}
```

Snippet 4.29: Definition of `toFields()`

Now, consider the below snippet:

```

import { Field } from '../../lib/core.js';
type PointWithOptionalsT = {
  x: typeof Field,
  y?: typeof Field
  z: typeof Field,
};

let PointWithOptionals: PointWithOptionalsT = {
  x: Field,
  z: Field,
};

let {provable} = createDerivers<Field>();
let ProvablePoint2D = provable(PointWithOptionals);

let p1 = {x: Field(0n), y: Field(1n), z: Field(2n)};
console.log("p1 =", ProvablePoint2D.toFields(p1).map(f => f.toBigInt()));
```

Here, `PointWithOptionals` has key `y` from the type checker's perspective with the value `undefined`. The code type checks. However, the key `y` is actually missing at run-time. Thus, the above program skipped the field of `y`.

```
p1 = [ 0n, 2n ]
```

Impact Clients may misunderstand the supported type of `typeObj`.

Further, the presence of `undefined` values of objects could cause an issue with `toFields()` similar to the one described in V-O1J-VUL-020. More generally, values present in `obj` which do not match the structure of `typeObj` exactly will be silently ignored.

This leads to additional surprising behavior. For example, as shown below, functions are converted to fields of empty type, rather than being ignored.

```
let Example = {
  wrapped: Field,

  toFields: (wrapped: Field): Field[] => {
    return [wrapped];
  },
}
type ProvableExample = InferProvable<typeof Example, Field>;
// The inferred type of ProvableExample is
// type ProvableExample = {
//     wrapped: Field;
//     toFields: {};
// }
```

Recommendation We recommend making two key changes:

1. `A` should have an upper bound instead of using the unbounded parametric type. This upper bound should be a recursive type similar to `JSONValue` (as it is a datum that is inductively defined).
    a) This upper bound type should prohibit `undefined` values of objects.
    b) This upper bound type should prohibit arrays from being mutated via `readonly`.
2. Avoid use of `any` in the implemented recursive methods. Instead, use `InferProvable` to infer an appropriate output type for each recursive call.
3. Either do not allow functions on the inferred type, or ignore them in the inferred type.

Developer Response We introduced a `NestedProvable` type which is suitable as the "upper bound" type that you recommend.

Currently, this type is used internally in the `provable()` deriver, but not yet as a bound on the type in the public API; but we plan to make this refactor in the future.

4.1.20 V-O1J-VUL-020: Incorrect array handling and type inference caveat

Severity	Low	Commit	3c68a0d
Type	Data Validation	Status	Fixed
File(s)	o1js-bindings/lib/provable-generic.ts		
Location(s)	toFields()		
Confirmed Fix At	https://github.com/o1-labs/o1js-bindings/pull/285		

TypeScript, by default, generalizes types such as arrays when doing type inference. If users are not careful, tuples (arrays with known length) could be widened into a general arrays. This then bypass static checks that some operations such as `toFields` implicitly rely on.

`provable().toFields()` assumes that any array is fixed-length. In the `Array.isArray()` case of the below implementation, the indices of `typeObj` are mapped over, rather than the indices of `obj`.

```
function toFields(typeObj: any, obj: any, isToplevel = false): Field[] {
  if (primitives.has(typeObj)) return [];
  if (!complexTypes.has(typeof typeObj))
    throw Error(`provable: unsupported type "${typeObj}"`);
  if (Array.isArray(typeObj))
    return typeObj.map((t, i) => toFields(t, obj[i])).flat();
  if ('toFields' in typeObj) return typeObj.toFields(obj);
  return (isToplevel ? objectKeys : Object.keys(typeObj))
    .map((k) => toFields(typeObj[k], obj[k]))
    .flat();
}
```

Snippet 4.30: Definition of `toFields()`

For example, consider the below snippet:

```
import { Field } from '../../lib/core.js';
let StrangePoint2D = {
  x: [Field, Field],
  y: [Field],
};

let {provable}  = createDerivers<Field>();

let ProvablePoint2D = provable(StrangePoint2D);

let p1 = {x: [Field(0n), Field(1n)], y: [Field(2n)]};
let p2 = {x: [Field(0n)], y: [Field(1n), Field(2n)]};

console.log("     Point2D: p1 =", ProvablePoint2D.toFields(p1).toString())
console.log("     Point2D: p2 =", ProvablePoint2D.toFields(p2).toString())
```

Here, the intention is that `Point2D` consists of a field `x` of exactly two `WrappedBigInt` and a field `y` of exactly one `Field`.

However, the type inference of `Point2D` makes generalization so that `x` and `y` are simply arrays. This then allows the invalid `ProvablePoint2D.toFields(p1)`, even though `p1` has a different shape than what `Point2D` intends to be.

The above program produces:

```
    Point2D: p1 = 0,1,2
    Point2D: p2 = 0,,1
```

Impact The resulting conversion to fields could be invalid. Certain field elements may be silently ignored.

Recommendation Add dynamic checks in `toFields` by making sure that `typeObj` and `obj` have the same length.

Even with the dynamic checks, getting static type checking to work would be highly beneficial, since the the feedback is immediate and it has higher coverage. One possible repair in the above client's code is to not rely on type inference, and to specify the type of `Point2D` (and `p1` and `p2`) explicitly, with `readonly` to prevent mutation on the tuple.

Another possible repair is to use const assertion (`as const`), which is available in TypeScript 3.4, to prevent widening.

```
// [VERIDISE] ...elided...

let StrangePoint2D = {
  x: [Field, Field],
  y: [Field],
} as const;

// [VERIDISE] ...elided...

let p1 = { x: [Field(0n)], y: [Field(1n), Field(2n)] } as const;
let p2 = { x: [Field(0n), Field(1n)], y: [Field(2n)] } as const;

console.log("p1 =", ProvablePoint2D.toFields(p1)); // correctly fails to type check
console.log("p2 =", ProvablePoint2D.toFields(p2));
```

The caveat and repair strategies should be clearly documented so that clients can still fully leverage static type checking. See also V-O1J-VUL-019.

Developer Response We implemented the recommendation to add dynamic length checks.

4.1.21 V-O1J-VUL-021: Missing check on infinity flag

Severity	Low	**Commit**	e7ded4b
Type	Logic Error	**Status**	Fixed
File(s)	o1js-bindings/crypto/eliptic-curve.ts		
Location(s)	AffineCurve.equal()		
Confirmed Fix At	https://github.com/o1-labs/o1js-bindings/pull/281,https://github.com/o1-labs/o1js/pull/1758		

As shown in the below code snippet, the `equal()` method in `AffineCurve` is intended to check whether two points on the curve are equal to one another.

```
equal(g: GroupAffine, h: GroupAffine) {
  return mod(g.x - h.x, p) === 0n && mod(g.y - h.y, p) === 0n;
},
```

Snippet 4.31: Definition of `AffineCurve.equal()`

However, this ignores the possibility of `g` or `h` being the point at infinity. As can be seen in the below type definition, an additional `boolean` flag is supplied to indicate whether a point is the identity element

```
type GroupAffine = { x: bigint; y: bigint; infinity: boolean };
```

Snippet 4.32: Definition of the `GroupAffine` type

In the definition of `equal()`, this `infinity` flag is ignored.

Impact Equality checks may succeed when they should fail, or vice versa, when the point at infinity is involved.

For example, this method is used in `verifyEcdsaConstant()`, which verifies ECDSA signatures. This allows public keys which are zero, but have `(x,y) != (0,0)` to be used. As mentioned in the resources linked in V-O1J-VUL-042, this does not conform to the standard ECDSA specification.

More generally, users may rely on this function returning `false` when points are not equal for many cryptographic applications.

This would be much more serious if the API was closer to the user. However, this is still exposed to user-facing functionality, as can be seen in the below script

```
import { createEcdsa, createForeignCurve, Crypto } from "o1js";

// create a secp256k1 curve
class Secp256k1 extends createForeignCurve(Crypto.CurveParams.Secp256k1) {}
// create an instance of ECDSA over secp256k1
class Ecdsa extends createEcdsa(Secp256k1) {}

// get a random point
let r = Secp256k1.Scalar.random();
let p = Secp256k1.generator.scale(r);

let pAsPt = Ecdsa.Curve.Bigint.from(p.toBigint());
```

```
let identityPt = {
    x: p.x.toBigInt(),
    y: p.y.toBigInt(),
    infinity: true
}
let isEqual = Ecdsa.Curve.Bigint.equal(pAsPt, identityPt);
console.log(pAsPt);
console.log(identityPt);
console.log(isEqual);
```

Example output:

```
{
  x: 135813378588513567858979651276039795096918535338382735390884365396975127962 66n,
  y: 101653036562733440740133669793482911760156824248243822141614287709741717068105n,
  infinity: false
}
{
  x: 135813378588513567858979651276039795096918535338382735390884365396975127962 66n,
  y: 101653036562733440740133669793482911760156824248243822141614287709741717068105n,
  infinity: true
}
true
```

Recommendation Change the type definition of `GroupAffine` to be a union type of the point at infinity and a point at non-infinity, and implement `equal` accordingly.

An alternative fix is to keep the current `GroupAffine`, but return `g.infinity == h.infinity` if `g` or `h` is the point at infinity.

Developer Response We updated the `GroupAffine` point to be a union type and re-implemented `equal` to handle the infinity-case correctly.

4.1.22 V-O1J-VUL-022: Bound checked on length instead of padLength

Severity	Low	**Commit**	8dde2c3
Type	Under-constrained Circuit	**Status**	Fixed
File(s)	`src/lib/provable/gadgets/bitwise.ts`		
Location(s)	xor(), not(), and()		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1745`		

The `xor()` function uses `buildXor()` to chain together several XOR gates.

```
function xor(a: Field, b: Field, length: number) {
  // check that both input lengths are positive
  assert(length > 0, 'Input lengths need to be positive values.');

  // check that length does not exceed maximum 254 size in bits
  assert(length <= 254, 'Length ${length} exceeds maximum of 254 bits.');

  // obtain pad length until the length is a multiple of 16 for n-bit length lookup
    table
  let padLength = Math.ceil(length / 16) * 16;

    // [VERIDISE: witness generation code removed for brevity]

  // builds the xor gadget chain
  buildXor(a, b, outputXor, padLength);
```

Snippet 4.33: Snippet from `xor()`

Each gate checks 16 bits. It is important that the number of gates used does not exceed the number of bits in the base field, otherwise an overflow may occur when constraining the weighted sum of the 16-bit limbs to equal the final output result.

To prevent against this, `xor()` checks that `length <= 254`. However, the actual length used is `padLength`, which rounds `length` up to the nearest multiple of 16. For `240 < length <= 254`, `padLength = 256`.

Similar issues occur in `and()` and `not()`, which both rely on `xor()` to function properly, and only require their inputs to fit within `padLength` bits rather than `length` bits (see V-O1J-VUL-057 for further discussion on the assumptions required by `not()`).

Impact If a user sets `padLength` in the range `(240, 255]`, their circuit will be underconstrained.

Note that `xor()`, `not()`, and `and()` are supplied to the user as gadgets.

Recommendation

1. Require `length <= 240`.
2. Consider refactoring the check on `length` into a single function called by `xor()`, `not()`, and `and()`.

Developer Response We implemented the recommendation.

4.1.23 V-O1J-VUL-023: quotientBits bound too low in divMod32

Severity	Low	Commit	8dde2c3
Type	Under-constrained Circuit	Status	Fixed
File(s)	`src/lib/provable/gadgets/arithmetic.ts`		
Location(s)	divMod32()		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1763`		

In some parts of the codebase, a user may provide a parameter to a gadget indicating a known bound on a given value. For example the user may supply `quotientBits` to `divMod32()`, indicating the maximum allowed (logarithm) of the `quotient`. This, in turn, leads to a bound of `quotientBits+32` on the supplied `n` value. This setting allows users to optimize for the small `n`-case when desired.

However, this also opens the door for user error. If `quotientBits >= 255`, then the range-check on `quotient` (shown in the below) snippet is meaningless.

```
if (quotientBits === 1) {
  quotient.assertBool();
} else {
  rangeCheckN(quotientBits, quotient);
}
rangeCheck32(remainder);

n.assertEquals(quotient.mul(1n << 32n).add(remainder));
```

Snippet 4.34: Snippet from `divMod32()`

To protect against this, `rangeCheckN` asserts that `quotientBits` is not too large. Note that the below snippet requires the number of bits to be a non-zero multiple of 16 less than or equal to `Fp.sizeInBits = 255`. The largest such value is `numBits = 15 * 16 = 240`.

```
// [VERIDISE NOTE: We renamed the variable n -> numBits for ease of presentation]
function rangeCheckN(numBits: number, x: Field, message: string = '') {
  assert(
    numBits <= Fp.sizeInBits,
    `bit length must be ${Fp.sizeInBits} or less, got ${numBits}`
  );
  assert(numBits > 0, `bit length must be positive, got ${numBits}`);
  assert(numBits % 16 === 0, ``length` has to be a multiple of 16.`);
```

Snippet 4.35: Beginning of `rangeCheckN()`.

However, this bound is not small enough. `divMod32()` asserts that `n == quotient * 2**32 + remainder`. To be a meaningful computation over the integers, the right-hand-side must not overflow the base field.

We know that `quotient * 2**32 + remainder` achieves its maximum (over the integers) when `quotient = 2**quotientBits - 1` and `remainder = 2**32 - 1`. So, the maximum value of the right-hand side (before reduction modulo the base field) is `2**(quotientBits+32) - 1`.

So, for `divMod32`'s constraints to work correctly, we must require `quotientBits + 32 < 255`, i.e. `quotientBits < 223`.

Impact If `quotientBits = 240` or `quotientBits = 224` is used, then an overflow may occur. This means that the final equation will have two solutions for some values of `n`, one using the correct division algorithm, and another leveraging the overflow of to compute an incorrect remainder.

Note that this function is exposed to library users as a gadget.

Recommendation

1. Rather than supplying `quotientBits`, supply `nBits` (a bound on `n`) and assert it is in the range `[0, 255)`. One may then compute `quotientBits` as `max(0, nBits - 32)`.
2. Consider requiring explicitly that `numBits < Fp.sizeInBits` in the functions `rangeCheckN()` and `isDefinitelyInRangeN()`. Although this is functionally equivalent, it better conveys the intent.

Developer Response The developers applied the recommendation.

4.1.24 V-O1J-VUL-024: Unsafe use of toProjective

Severity	Low	**Commit**	8dde2c3
Type	Data Validation	**Status**	Acknowledged
File(s)	`o1js/src/lib/provable/group.ts`		
Location(s)	scale()		
Confirmed Fix At	N/A		

The method `scale()` of `Group` performs the elliptic curve scalar multiplication.

```
  scale(s: Scalar | number | bigint) {
    let scalar = Scalar.from(s);

    if (isConstant(this) && scalar.isConstant()) {
      let g_proj = Pallas.scale(toProjective(this), scalar.toBigInt());
      return fromProjective(g_proj);
    } else {
      let [, ...bits] = scalar.value;
      bits.reverse();
      let [, x, y] = Snarky.group.scale(toTuple(this), [0, ...bits]);
      return new Group({ x, y });
    }
  }
```

Snippet 4.36: Snippet from `scale()`

The method uses the `toProjective` helper function, which implicitly assumes that it is not the point at infinity.

```
function toProjective(g: Group) {
  return Pallas.fromAffine({
    x: g.x.toBigInt(),
    y: g.y.toBigInt(),
    infinity: false,
  });
}
```

Snippet 4.37: Snippet from `toProjective()`

However, this assumption doesn't hold in the above usage, because `this` could be zero.

Impact An elliptic curve scalar multiplication could fail, as shown below (and with a poor error message).

```
let g = Group.zero;
let g5 = g.scale(5);

// Error: projectiveDouble: unhandled case
```

Recommendation Handle the case where `this` is zero. `toProjective` should also document its assumptions clearly.

Developer Response The circuit gadget does not handle zero for performance reasons.

4.1.25 V-O1J-VUL-025: Missing range check in assertOnCurve

Severity	Low	**Commit**	8dde2c3
Type	Logic Error	**Status**	Fixed
File(s)	src/lib/provable/gadgets/elliptic-curve.ts		
Location(s)	assertOnCurve()		
Confirmed Fix At	https://github.com/o1-labs/o1js/pull/1743		

The function `assertOnCurve` asserts that a point is on a given curve.

```
let { x, y } = p;
let x2 = ForeignField.mul(x, x, f);
let y2 = ForeignField.mul(y, y, f);
let y2MinusB = ForeignField.Sum(y2).sub(Field3.from(b));

// (x^2 + a) * x = y^2 - b
let x2PlusA = ForeignField.Sum(x2);
if (a !== 0n) x2PlusA = x2PlusA.add(Field3.from(a));
let message: string | undefined;
if (Point.isConstant(p)) {
  message = `assertOnCurve(): (${x}, ${y}) is not on the curve.`;
}
ForeignField.assertMul(x2PlusA, x, y2MinusB, f, message);
```

Snippet 4.38: Snippet from `assertOnCurve()`

The assertion is done by calling `ForeignField.assertMul` to check that `x2PlusA` multiplied by `x` is equal to `y2MinusB`. A precondition of `ForeignField.assertMul` is that `x2PlusA` multiplied by `x` is bounded to be less than $2^{264}M$ where M is the native modulus. This precondition holds when the multiplication inputs are almost reduced.

The value of `x2PlusA` is computed from `x2`. Note, however, that the result of foreign field multiplication might not be almost reduced. For example, a malicious prover could add large multiples of `f` to `x2`. If that happens, the precondition could be violated.

Impact This may be difficult to exploit, as many cryptographic protocols are still difficult to exploit even with elements off of the curve. However, it will still prevent, for example, full compliance with the ECDSA specification.

Recommendation Add checks to make sure that the multiplication result is almost reduced.

Developer Response We added an almost-reduced check on `x^2` (as well as `x` and `y`) to the `assertOnCurve()` method.

4.1.26 V-O1J-VUL-026: Reduction of ForeignField element

Severity	Low	**Commit**	8dde2c3
Type	Data Validation	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1779`		

An element in a foreign field can be constructed from an element from another foreign field or `Field3`.

```
  constructor(x: ForeignField | Field3 | bigint | number | string) {
    const p = this.modulus;
    if (x instanceof ForeignField) {
      this.value = x.value;
      return;
    }
    // Field3
    if (Array.isArray(x)) {
      this.value = x;
      return;
    }
    // constant
    this.value = Field3.from(mod(BigInt(x), p));
  }
```

Snippet 4.39: Snippet from the constructor of `ForeignField`

This has several issues. For example, consider the case where an element from another foreign field is constant and larger than the modulus of the current foreign field. In this case, the constructor would simply adopt the value without performing any reduction. This violates the invariant, and could then cause failure in subsequent computation.

```
class SmallField extends createForeignField(17n) {}
class SlightlyLargerField extends createForeignField(19n) {}

let x1 = new SmallField(0);
let x2 = new SlightlyLargerField(17);
let x2inSmallField = new SmallField(x2);
x2inSmallField.assertEquals(x1, "bad!");
```

Similarly, a constant `Field3` can be directly supplied to the constructor, and the same failure could occur.

```
class SmallField extends createForeignField(17n) {}

let x1 = new SmallField(0);
let x2inSmallField = new SmallField([Field(17), Field(0), Field(0)]);
x2inSmallField.assertEquals(x1, "bad!");
```

Similar issues occur in

1. `src/lib/provable/crypto/foreign-curve.ts`: `ForeignCurve`
2. `src/lib/provable/crypto/foreign-ecdsa.ts`: `EcdsaSignature`

Impact For the constant case, depending on the intention, this could either cause a failure when the program is expected to work, or result in a poor error.

For the non-constant case, this could result in missing constraints.

Recommendation

1. If the construction from another foreign field of different size is not intended, the field modulus should be checked to guarantee that they are the same.
2. For constant cases, either the above programs are expected to work, or they are not.
   - In case they are expected to work, reduction should be performed so that constant `ForeignField` / `Field3` are in the canonical form.
   - In case they are not, the above foreign field element construction should add checks so that it fails right away, with a more understandable error message.
3. Range check the the elements from non-constant `ForeignField` / `Field3` during the construction, or clearly document that this must be done.
4. Ensure any classes built from `Fields` check that supplied elements are of the correct field. This applies to, for example
   a) `src/lib/provable/crypto/foreign-curve.ts`: `ForeignCurve`
   b) `src/lib/provable/crypto/foreign-ecdsa.ts`: `EcdsaSignature`
5. Address similar issues in `ForeignCurve` to ensure the provided point refers to the same equation and subgroup defined over the same field.

Developer Response We have added documentation indicating the need to reduce when constructing foreign fields from another foreign field or curve points from another curve.

An error is now produced when constructing a foreign field element from a foreign field element with a different modulus.

4.1.27 V-O1J-VUL-027: Unchecked prefix could lead to collision

Severity	Low	**Commit**	8dde2c3
Type	Data Validation	**Status**	Fixed
File(s)	`src/lib/provable/crypto/poseidon.ts`		
Location(s)	prefixToField()		
Confirmed Fix At	`https://github.com/o1-labs/o1js/pull/1739`		

The function `prefixToField` can be used to convert a prefix string to a list of `Field`s.

```
function prefixToField(prefix: string) {
  if (prefix.length * 8 >= 255) throw Error('prefix too long');
  let bits = [...prefix]
    .map((char) => {
      // convert char to 8 bits
      let bits = [];
      for (let j = 0, c = char.charCodeAt(0); j < 8; j++, c >>= 1) {
        bits.push(!!(c & 1));
      }
      return bits;
    })
    .flat();
  return Field.fromBits(bits);
}
```

Snippet 4.40: Snippet from `prefixToField()`

For each character, the function assumes that the character fits within 8 bits and decomposes it into 8 bits.

However, this assumption would not be true for unicode characters, which could exceed 8 bits.

Impact Following is one possible scenario where an attack could occur.

Suppose that "protocol A" has users send signatures, using a special prefix "`protocolAOnly`" as a domain separator. An attacker could then set up protocol B, which is a fork of protocol `A`. Protocol B may use a domain separator which is different in unicode, but matches `protocolAOnly` when interpreted as ASCII. Once the attacker gets enough signatures on the fork, they could start using them to spend the funds of users who signed up on both protocols.

Recommendation If only ASCII is intended, then there should be a check that `c` has no left over.

Developer Response We added an assertion to only support the default ascii set, requiring that the character is fully consumed after 7 bits.

4.1.28 V-O1J-VUL-028: Field/curve operations assuming canonical form

Severity	Low	**Commit**	e7ded4b
Type	Logic Error	**Status**	Partially Fixed
File(s)	`o1js-bindings/crypto/finite-field.ts`, `o1js-bindings/crypto/elliptic-curve.ts`		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/o1-labs/o1js-bindings/pull/287`		

`finite-field.ts` implements several common operations performed on fields. For example, utilities for field addition, multiplication, exponentiation, and inversion are defined. These operations perform computations on field elements, represented as `bigints`.

Since the field elements are represented as `bigints`, they are not necessarily represented in their canonical form. For example, the `0` element in `Fp` may be presented by `0`, `p`, `2p`, etc. While many of the field operations work correctly on any `bigint` representation of a field element, others assume that the provided `bigint` is in its canonical form, i.e. that it has been reduced modulo `p`.

Below, we list the functions which assume that the `bigint` is in its canonical form.

- `src/bindings/crypto/elliptic-curve.ts`:
  - `projectiveNeg()`
  - `projectiveAdd()`
  - `projectiveDouble()`
  - `projectiveToAffine()`
  - `projectiveEqual()`
  - `affineAdd()`
  - `affineDouble()`
  - `affineNegate()`
- `src/bindings/crypto/finite-field.ts`:
  - `not()`
  - `negate()`
  - `isSquare()`
  - `sqrt()`
  - `isEven()`
  - `rot()`
  - `leftShift()`
  - `rightShift()`

For the bitwise operations (`not()`, `rot()`, `leftShift()`, and `rightShift()`), this is assumption is clear in the implementation. The remaining functions implicitly make the assumption due to their handling of `0`. We break each of these examples down below:

- `src/bindings/crypto/elliptic-curve.ts`:
  - `projectiveNeg()`: While functionally correct, this will output non-canonical representations if non-canonical representations are supplied.

- The below functions checks if a point is the projective zero point by comparing the z-coordinate to `0` before reducing modulo `p`.
  - `projectiveAdd()`
  - `projectiveDouble()`
  - `projectiveToAffine()`
  - `projectiveEqual()`
- `affineAdd()`: This function performs equality checks on the `x`/`y` coordinates before reducing modulo `p`.
- `affineNegate()`: While functionally correct, this will output non-canonical representations if non-canonical representations are supplied.

▶ `src/bindings/crypto/finite-field.ts`:

- `negate()`: `negate(x)` is intended to correspond to `-x`. While the below implementation does not perform an *incorrect* computation, it can output a non-canonical form when `x > p` or `x < 0`.

```
 negate(x: bigint) {
  return x === 0n ? 0n : p - x;
},
```

Snippet 4.41: Definition of `negate()`

- `isSquare()`: The `isSquare()` function checks if `x` raised to the power `(p-1)/2` is 1 to determine if `x` is a quadratic residue. However, it has a special case for `0`, since `0^((p-1)/2) == 0`. This special case checks if `x === 0n`, instead of checking if `mod(x, p) === 0n`. For non-zero multiples of `p`, `isSquare()` will return `false`.

For example, this leads to issues when calling `createField(2)`, since `computeConstants()` calls `isSquare(2, 2)`.

```
function isSquare(x: bigint, p: bigint) {
  if (x === 0n) return true;
  let sqrt1 = power(x, (p - 1n) / 2n, p);
  return sqrt1 === 1n;
}
```

Snippet 4.42: Definition of `isSquare()`

- `sqrt()`: The `sqrt()` function implements the Tonelli-Shanks algorithm for computing square roots. As for `inSquare()`, the zero-check assumes that `n` is represented in its canonical form. For non-zero multiples of `p`, the `while` loop runs forever without terminating.

Specifically, the innermost while-loop (not shown below) fails to terminate.

```
function sqrt(n: bigint, p: bigint, Q: bigint, c: bigint, M: bigint) {
  if (n === 0n) return 0n;
  let t = power(n, (Q - 1n) >> 1n, p); // n^(Q - 1)/2
  let R = mod(t * n, p); // n^((Q - 1)/2 + 1) = n^((Q + 1)/2)
  t = mod(t * R, p); // n^((Q - 1)/2 + (Q + 1)/2) = n^Q
  while (true) {
    // [VERIDISE] body elided for brevity
  }
```

```
}
```

Snippet 4.43: Definition of `sqrt()`

- `isEven()`: Since most primes are odd, the below implementation can return either `0` or `1` depending on the representative chosen for `x`.

```
isEven(x: bigint) {
  return !(x & 1n);
},
```

Snippet 4.44: Definition of `isEven()`

Impact Users may receive incorrect or out of range values if they forget to reduce the inputs modulo `p`.

Note that these methods are exposed to the user through the API via `ForeignField.Bigint`. For example, the below script provokes the infinite loop mentioned in the `sqrt()` implementation:

```

import { createForeignField } from 'o1js';

class Field17 extends createForeignField(17n) {}

// This works fine
console.log(Field17.Bigint.sqrt(0n));
// this runs forever
console.log(Field17.Bigint.sqrt(17n));
```

Recommendation Modify all implementations to either `assert` that `x` is in the appropriate range, or properly reduce `x` into the appropriate range.

If this recommendation is not taken, we recommend adding a field element type which wraps a `bigint`. This would allow developers to leverage the type system to check the reduction assumption, rather than requiring the caller to remember to reduce `x` for certain functions.

At the very least, documentation for these invariants/assumptions on each function should be added.

Developer Response We reduced the values in the following functions:

- `src/bindings/crypto/finite-field.ts`:
  - `negate()`
  - `isSquare()`
  - `sqrt()`
  - `isEven()`

Updated Veridise Response Since the finite field-bitwise operations and the elliptic curve operations are left unchanged, we are marking this issue as partially fixed.

4.1.29 V-O1J-VUL-029: Infinite loop from user error

Severity	Warning	**Commit**	e7ded4b
Type	Maintainability	**Status**	Acknowledged
File(s)	`o1js-bindings/crypto/finite-field.ts`		
Location(s)	computeFieldConstants()		
Confirmed Fix At	N/A		

The `createField()` function calls `computeFieldConstants()` to compute useful constants related to a particular field. While the expected behavior is for users to submit large primes `p`, users may submit non-primes as `p`.

If a user sets `p == 1`, they will provoke an infinite loop in the implementation.

```
let oddFactor = p - 1n;
let twoadicity = 0n;
while ((oddFactor & 1n) === 0n) {
  oddFactor >>= 1n;
  twoadicity++;
}
```

Snippet 4.45: Snippet from `computeFieldConstants()`

In the above loop, `oddFactor` is repeatedly shifted right by one bit until a non-zero bit is reached. This assumes that `oddFactor` has at least one non-zero bit, which is the case whenever `p-1 != 0`. However, if `p = 1`, this loop will run forever.

Impact Confused users submitting invalid values of `p` will have a difficult time identifying their error.

Recommendation Check that `p > 1` when calling `createField()`. It may be useful to provide a few other basic sanity checks for the user, such as requiring `p % 2 == 0` for `p >= 3`.

Developer Response We intend for most operations on created-fields to fail gracefully when presented with non-primes, and work correctly when the operation is well-defined.

Updated Veridise Response The following locations assume that `p` is prime. To allow for graceful error messages a primality test should be performed when the field is created.

- `fastInverse()` : asserts an inverse exists. Given the custom optimized implementation, this behavior is likely preferable to supporting non-prime values `p`, and `inverse()` should be preferred in those cases.
- `sqrt()` : the inner loop assumes repeated squaring will eventually reach `1`. This may not be the case, e.g. for `sqrt(2)` in `Z_4`, and will provoke an infinite loop. More generally, this example shows that Tonelli-Shanks relies on the primality of `p`.
- `isSquare()`: This function checks if `x^((p-1)/2)` is `1` to determine if `x` is square. However, `4^((6-1)/2) mod 6 = 4 mod 6 != 1 mod 6`, but `4` does have a square root in `Z_6`.

Updated Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.30 V-O1J-VUL-030: Prover errors when calling set() on same contract

Severity	Warning	**Commit**	8dde2c3
Type	Transaction Ordering	**Status**	Acknowledged
File(s)	`src/lib/mina/state.ts`		
Location(s)	set()		
Confirmed Fix At	N/A		

Calling `set()` inside a method updates the app-state to be changed by the `AccountUpdate`. This update is not performed until the `AccountUpdate` is processed. As described in V-O1J-VUL-012, this means that `get()`s within the same method as a `set()` ignore the memory write.

However, due to variable caching, this occurs even over multiple method calls. Consider the below example. One contract (the `NonceProvider`) just repeatedly returns a fresh nonce and increments state. Another contract (the `SigVerifier`) uses the nonces to check signatures.

Calling `multiVerify()` with signatures generated using a nonce of 0 and 1 (as expected) leads to a prover error. Conversely, calling `multiVerify()` with signatures generating using a nonce of 0 and 0 passes the proof, but leads to a transaction validation error due to a precondition failure on the second account update.

```
// contract which provides a unique nonce
class NonceProvider extends SmartContract {
  @state(Field) nonce = State<Field>();

  constructor(address: PublicKey) {
    super(address);
    super.init();
  }

  @method.returns(Field)
  async getNonce() {
    let nonce = this.nonce.getAndRequireEquals();
    let nextNonce = nonce.add(1n);
    this.nonce.set(nextNonce);
    return nonce;
  }
}

class BatchSigs extends Struct({
  signatures: [Signature, Signature],
  msgs: [Field, Field],
}) {}

// Verifier for nonces
class SigVerifier extends SmartContract {
  @method.returns(Bool)
  async multiVerify(batchSigs: BatchSigs) {
    let nonceProvider = new NonceProvider(nonceProviderAddress);

    let {signatures, msgs: messages} = batchSigs;

    assert(signatures.length == messages.length); // sanity check
```

```
33      for(let i = 0; i < signatures.length; ++i) {
34        let nonce = await nonceProvider.getNonce();
35        let noncedMessage = [messages[i], nonce]
36
37        // verify nonced message
38        Provable.log(noncedMessage);
39        signatures[i].verify(sigVerifierAddress, noncedMessage).assertTrue();
40      }
41      return new Bool(true);
42    }
43  }
```

Snippet 4.46: Example setup which causes a prover-error

Impact Contracts cannot call a method which updates state multiple times.

Recommendation Change the caching behavior of `get()` to not repeatedly use the same variable when called from different method calls.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

4.1.31 V-O1J-VUL-031: Over-constrained doubling circuit

Severity	Warning	Commit	8dde2c3
Type	Over-constrained Circuit	Status	Acknowledged
File(s)	`src/lib/provable/gadgets/elliptic-curve.ts`		
Location(s)	double()		
Confirmed Fix At	N/A		

The elliptic-curve doubling function computes `2 * P` for points `P` on the elliptic curve. It implements a common formula used for doubling in short Weierstrass curves:

$$x_3 = \frac{(3x_1^2 + a)^2}{(2y_1)^2} - x_1 - x_1, \quad y_3 = (3x_1)\frac{3x_1^2 + a}{2y_1} - \frac{(3x_1^2 + a)^3}{(2y_1)^3} - y1$$

With the added substitution

$$m := \frac{3x_1^2 + a}{2y_1}$$

Note that m is only well-defined when $y_1 \neq 0$. This is an assumption which holds for most curve points. However, in some curves, there are points with $y_1 = 0$. Note that if $y_1 = 0$, then $-P = -(x_1, 0) = (x_1, -0) = (x_1, 0) = P$, i.e. $2 \cdot P = 0$. This case occurs for elliptic curves with an even order. For example, one may use the below SAGE script to find points on secp128r2, a SECG-recommended 128-bit elliptic curve.

```
import random

# Your existing code
p = 0xfffffffdffffffffffffffffffffffff
K = GF(p)
a = K(0xd6031998d1b3bbfebf59cc9bbff9aee1)
b = K(0x5eeefca380d02919dc2c6558bb6d8a5d)
E = EllipticCurve(K, (a, b))
G = E(0x7b6aa5d85e572983e6fb32a7cdebc140, 0x27b6916a894d3aee7106fe805fc34b44)
E.set_order(0x3fffffff7fffffffbe0024720613b5a3 * 0x04)

# Pick a random point on the curve
random_point = E.random_point()

# Scale the random point by the given scalar
scaled_point = random_point * 0x3fffffff7fffffffbe0024720613b5a3

print("Random point on the curve:", random_point)
print("Scaled point:", scaled_point)
print("2 * Scaled point:", 2 * scaled_point)
print("4 * Scaled point:", 4 * scaled_point)
```

Snippet 4.47: SAGE script. You can run this script at https://sagecell.sagemath.org to reproduce the results.

```
Random point on the curve: (159991821442443375453857009627776707231 :
    164640867175291007436737788367347982338 : 1)
Scaled point: (311198077076599516590082177721943503640 :
    260967053171219602502651970460946518251 : 1)
2 * Scaled point: (311198077076599516590082177721943503641 : 0 : 1)
4 * Scaled point: (0 : 1 : 0)
```

Snippet 4.48: Example SAGE script output, showing points with `4 * P == 0` and `2 * P == 0`.

In this case, the constraints `2 * y1*m = 3*x1*x1 + a` (shown below) become `3 * x1*x1 + a == 0`. Assuming `(x1, 0)` is a valid curve point, we also have `x1^3 + a*x1 + b == 0`. For most curves, this will lead to an over-constrained circuit. Note that, in some unlikely scenarios, this may instead lead to an under-constrained circuit.

```
// 2*y1*m = 3*x1x1 + a
let y1Times2 = ForeignField.Sum(y1).add(y1);
let x1x1Times3PlusA = ForeignField.Sum(x1x1).add(x1x1).add(x1x1);
if (Curve.a !== 0n)
  x1x1Times3PlusA = x1x1Times3PlusA.add(Field3.from(Curve.a));
ForeignField.assertMul(y1Times2, m, x1x1Times3PlusA, f);
```

Snippet 4.49: Snippet from `double()`

Impact While this is a possibility, it is unlikely for a few reasons.

1. Most common curves with an even co-factor require a base field larger than 264 bits (e.g. popular BLS curves like BLS-381 operate over 300 bit fields). Below, we list all Weierstrass curves over prime fields in a database of standard curves (https://github.com/J08nY/std-curves) with an even co-factor and at most 264 bits:

```
SECG secp112r2
Cofactor: 4
Bits: 112

SECG secp128r2
Cofactor: 4
Bits: 128

other Curve22103
Cofactor: 8
Bits: 221

other Curve4417
Cofactor: 4
Bits: 226

other Curve1174
Cofactor: 4
Bits: 251
```

2. The most likely operation to be performed on an element which does not lie in the prime-order subgroup is co-factor clearing. For most cryptographic applications, if the

resultant element is the identity, the algorithm will output REJECT.

Nonetheless, this is an unsupported case of `double()`ing, and should be fixed or documented clearly.

Recommendation Change `double()` to match the constant implementation and require `y1 != 0`. Add documentation indicating that points for which `y1 = 0` are not supported by the doubling algorithm.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.32 V-O1J-VUL-032: Negative powers unhandled

Severity	Warning	**Commit**	e7ded4b
Type	Usability Issue	**Status**	Acknowledged
File(s)	o1js-bindings/crypto/finite-field.ts		
Location(s)	power()		
Confirmed Fix At	N/A		

The `power()` function performs modular exponentiation, i.e. exponentiation modulo p.

```
function power(a: bigint, n: bigint, p: bigint) {
  a = mod(a, p);
  let x = 1n;
  for (; n > 0n; n >>= 1n) {
    if (n & 1n) x = mod(x * a, p);
    a = mod(a * a, p);
  }
  return x;
}
```

Snippet 4.50: Definition of `power()`

This implementation is correct for $n \geq 0$. However, for $n < 0$, it will return an incorrect value.

Also, note that for $a \bmod p = 0$, this function returns 1. While this is a fairly standard convention, there are also conventions which define 0^0 as undefined. Developers must read the function implementation to determine which standard is being used.

Impact Users who pass $n < 0$ to `power()` (e.g. by calling `Field.power()`) may use an incorrect value without realizing it.

Direct use cases within the library currently pass non-negative exponents, but this may impede user experience or lead to bugs in future development.

Recommendation Either implement the $n < 0$ case, or assert that n is non-negative.

Add documentation describing the behavior of `power(0, 0, p)`.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.33 V-O1J-VUL-033: Almost reduced described incorrectly

Severity	Warning	**Commit**	8dde2c3
Type	Maintainability	**Status**	Acknowledged
File(s)	`src/lib/provable/foreign-field.ts`		
Location(s)	See issue description		
Confirmed Fix At	N/A		

The documentation for `ForeignFields` describes "almost-reduced" fields as follows: "Since the full `x < p` check is expensive, by default we only prove a weaker assertion, `x < 2^ceil(log2(p))`, see `{@link ForeignField.assertAlmostReduced}` for more details." However, this is only an accurate description if the field modulus is at least `2^{176}`.

Below is a full description of what an "almost reduced" element is, as enforced by the code. In what follows, we indicate the standard 3-limb, 88-bit representation of `x = x0 + 2^{88} x1 + 2^{176} x2` by appending an index to `x`. We write `f` for the foreign field modulus.

- `x`'s representation must be valid, i.e. `x0, x1, x2 < 2^{88}`.
- `f2 > 0` (i.e. `f >= 2^{176}`):
  - `x` may be either in canonical form, or up to one factor of `f` away from canonical. More precisely, `x < 2^ceil(log2(p))`.
- `f_2 = 0`:
  - `x < 2^{176}`.
- For modular arithmetic use cases where `f` is a multiple of `2^{176}`, the tighter constraint `x_2 < f_2` is imposed, enforcing canonical form.

Impact Users cannot rely on the provided guarantees for almost-reduced fields. In particular, for small fields, applications cannot assume that `x < 2f` when `x` is almost reduced.

This can also lead to issues for developers. For example, `assertMul()` applies the following reasoning to bound the length of the input arrays

```
// conservative estimate to ensure that multiplication bound is satisfied
// we assume that all summands si are bounded with si[2] <= f[2] checks,
// which implies si < 2^k where k := ceil(log(f))
// our assertion below gives us
// |x|*|y| + q*f + |r|
//  < (x.length * y.length) 2^2k + 2^2k + 2^2k
//  < 3 * 2^(2*258)
//  < 2^264 * (native modulus)
assert(
  BigInt(Math.ceil(Math.sqrt(x.length * y.length))) * f < 1n << 258n,
  `Foreign modulus is too large for multiplication of sums of lengths ${x.length} and
     ${y.length}`
);
```

Snippet 4.51: Snippet from `assertMul()` in `o1js/src/lib/provable/gadgets/foreign-field.ts`.

However `|x| * |y|` is not bounded by `(x.length * y.length) 2^{2k}`. Instead, it is bounded by `(x.length * y.length) * max(2*f, 2^{176})`. This bound should be used instead.

Recommendation

1. Document the "almost reduced" condition more precisely.
2. Fix the bound in `assertMul()`.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.34 V-O1J-VUL-034: Off-by-one in some base conversions

Severity	Warning	**Commit**	e7ded4b
Type	Logic Error	**Status**	Acknowledged
File(s)	`o1js-bindings/crypto/bigint-helpers.ts`		
Location(s)	toBase()		
Confirmed Fix At	N/A		

The `toBase()` function is used to convert a `BigInt` value `x` to a series of digits base `base`. This is performed using a variant of Estrin's Scheme. In effect, this treats `x` as a single digit base `base^(2^k)` (for some `k`), and then repeatedly reduces `k` until `x` is represented as a sequence of digits base `base`.

```
function toBase(x: bigint, base: bigint) {
  if (base <= 0n) throw Error('toBase: base must be positive');
  // compute powers base, base^2, base^4, ..., base^(2^k)
   // with largest k s.t. base^(2^k) < x
   let basePowers = [];
  for (let power = base; power < x; power **= 2n) {
    basePowers.push(power);
  }
  let digits = [x]; // single digit w.r.t base^(2^(k+1))
```

Snippet 4.52: Snippet from `toBase()`

At the end of the loop shown above, the powers `base`, `base^2`, `base^4`, ..., `base^(2^k)` are stored in `basePowers`, with `k` the largest integer such that `base^(2^k) < x`.

After the loop, the developers claim that `x` is a single digit with respect to base `base^(2^(k+1))`. However, for this to be true, we must have `x < base^(2^(k+1))`. For `x = base^(2^k)`, this inequality does not hold.

Impact For values of `x` which are powers-of-2 powers of `base`, `toBase()` will return the incorrect value.

This can be seen in the below example.

```
let fourBaseTwo = changeBase([0n, 1n], 4n, 2n);
console.log('fourBaseTwo = ${fourBaseTwo}')
expect(fourBaseTwo.length).toEqual(3);
expect(fourBaseTwo[0]).toEqual(0n);
expect(fourBaseTwo[1]).toEqual(0n);
expect(fourBaseTwo[2]).toEqual(1n);
```

`changeBase()` uses the functions `fromBase()` and `toBase()` to perform base conversions. Provided a representation of `4` base `4` (i.e. `[0,1]`), the value base `2` should be `[0,0,1]`. However, the above code snippet will output

```
fourBaseTwo = 0,2
```


Recommendation Loop until `power <= x` so that `x < base^(2^(k+1))`.

Note that, at first glance, it appears this may be an issue in `fromBase()` as well.

```
function fromBase(digits: bigint[], base: bigint) {
  if (base <= 0n) throw Error('fromBase: base must be positive');
  // compute powers base, base^2, base^4, ..., base^(2^k)
  // with largest k s.t. n = 2^k < digits.length
  let basePowers = [];
  for (let power = base, n = 1; n < digits.length; power **= 2n, n *= 2) {
    basePowers.push(power);
  }
  let k = basePowers.length;
```

Snippet 4.53: Snippet from `fromBase()`

However, in this code snippet, the variable `let k = basePowers.length` is in fact one greater than the `k` referenced in the comments. We recommend renaming this variable to `kPlusOne`.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.35 V-O1J-VUL-035: Hash collisions in curve domain separator

Severity	Warning	**Commit**	e7ded4b
Type	Hash Collision	**Status**	Acknowledged
File(s)	`src/lib/gadgets/elliptic-curve.ts`		
Location(s)	initialAggregator()		
Confirmed Fix At	N/A		

The elliptic curve operation `multiScalarMul()` uses the result of an `initialAggregator()` to produce a "random" element on the curve of which no entity knows the discrete logarithm. The computation of this point could lead to errors or potential hash collisions across different curves.

1. The `initialAggregator()` function generates some `bytes` based on the curve parameters as shown below. However, the function `bigIntToBytes()` returns an array of dynamic size, returning the shortest byte array necessary to represent the value. This means that possible hash collisions may occur. For example, if `Curve.a` is `0`, then `bigIntToBytes(Curve.a) = []`. In this case, swapping the definitions of `Curve.a` and `Curve.b` would lead to the same value for `bytes`.

```
function initialAggregator(Curve: CurveAffine) {
  // hash that identifies the curve
  let h = sha256.create();
  h.update('initial-aggregator');
  h.update(bigIntToBytes(Curve.modulus));
  h.update(bigIntToBytes(Curve.order));
  h.update(bigIntToBytes(Curve.a));
  h.update(bigIntToBytes(Curve.b));
  let bytes = h.array();
```

Snippet 4.54: Snippet from `initialAggregator()`

2. As can be seen in the above text snippet, the curve generator is not included in the hash. This may allow the choice of a curve generator based upon the value of the `initialAggregator()`, leading to potentially catastrophic results (see V-O1J-VUL-007).

Impact

1. The intended behavior of `bytes` is to be uniquely associated to a particular curve. While it is not clear how a collision could be abused in this case, it is not intended behavior, and would best be avoided to ensure applicable security analyses hold for the implementation.
2. Attackers may use knowledge of the initial aggregator's discrete logarithm to verify invalid ECDSA signatures.

Recommendation

1. Prefix each array returned by `bigIntToBytes(*)` with its length before passing the value to sha256. This will ensure the mapping `(modulus, order, a, b) -> hash input` is injective.
2. Include the curve generator in the hash input used to compute the `initialAggregator()`.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.36 V-O1J-VUL-036: Non-injective padding for hash functions

Severity	Warning	**Commit**	8dde2c3
Type	Hash Collision	**Status**	Acknowledged
File(s)	`src/lib/provable/bytes.ts`. `src/lib/provable/crypto/poseidon.ts`		
Location(s)	See issue description		
Confirmed Fix At	N/A		

Properly padding messages is an extremely important aspect of computing hash functions. A padding scheme which is non-injective (i.e. $m||pad(m) = m'||pad(m')$ for some $m' \neq m$) allows an attacker to construct hash collisions.

The following structures pad a message with 0s out to a specific length.

1. `src/lib/provable/bytes.ts`: The `Bytes` constructor automatically pads arrays shorter than `size` with 0s.
2. `src/lib/provable/crypto/poseidon.ts`: All messages hashed by `Poseidon` are padded out with zeros to a multiple of 2.

Note that neither of these are injective means of padding:

1. o1js defines both SHA2 and SHA3 implementations accepting `Bytes32` (a class extending `Bytes`) arguments. If an array of bytes shorter than length 32 is supplied, it will be pad with zeros. This means, for example, that `[0]` and `[0,0]` would (silently) hash to the same value after conversion to `Bytes32`.
2. For any message `m` of odd length, `m||0` and `m` will hash to the same value.

Impact This can easily lead to cross-application hash collisions (or even same-application if different APIs hash different lengths of fields).

Recommendation

1. For the `Bytes` case, consider throwing an informative error when fewer than `size`-many bytes are provided. This will allow users to decide how best to pad their messages, and warn them of the possible attacks.
2. For the Poseidon case, this should be well-documented internally. We strongly recommend *not* exposing an API to the user which hashes without injective padding. Instead, consider adding a user-facing API which pads the message to an appropriate length with an injective padding function.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.37 V-O1J-VUL-037: Cofactor clearing over-constrained

Severity	Warning	**Commit**	8dde2c3
Type	Over-constrained Circuit	**Status**	Acknowledged
File(s)	`src/lib/provable/gadgets/elliptic-curve.ts`		
Location(s)	getPointTable()		
Confirmed Fix At	N/A		

The `getPointTable()` function is intended to return multiples of `P` up to a constant, i.e.

```
[0, P, 2P, 3P, ..., (2^(windowSize)-1)P]
```

(see V-O1J-VUL-057 for a discussion on the meaning of 0 in this context).

This is computed by taking `P`, doubling it, and the repeatedly adding `P`.

```
let Pi = double(P, Curve);
table.push(Pi);
for (let i = 3; i < n; i++) {
  Pi = add(Pi, P, Curve);
  table.push(Pi);
}
```

Snippet 4.55: Snippet from `getPointTable()`

Note that `add(g,h,Curve)` is (intentionally) over-constrained when `g == -h`, and verification will fail.

As described in V-O1J-VUL-031, some elliptic curves have points of very low order. If `k * P == 0` for `k < 2**windowSize-1`, then `Pi` will equal `-P` for some `i`, and `add(Pi, P, Curve)` will be over-constrained.

Impact This may lead to confusing errors for the user. However, since `multiScalarMul()` is not directly exported, and the current window-size is one, this should only occur if changes to the repository are made.

Recommendation Document this requirement in `multiScalarMul()` and `getPointTable()`. As recommended in V-O1J-VUL-007, consider supporting the `x1 == x2` case in addition.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.38 V-O1J-VUL-038: Unhandled Map/Set corner cases

Severity	Warning	**Commit**	8dde2c3
Type	Maintainability	**Status**	Acknowledged
File(s)	`src/lib/provable/types/struct.ts`		
Location(s)	cloneCircuitValue(), circuitValueEquals()		
Confirmed Fix At	N/A		

`struct.ts` defines several utility functions to help manipulate in-circuit values. For example, `cloneCircuitValue()` allows cloning in-circuit values, and `circuitValueEquals()` allows checking (constant) provable `Structs` for equality.

Both methods attempt to support `Maps` and `Sets`, but do not override the internal hash-function used to store objects by `Maps` and `Sets`. For example, the below code snippet checks two `Sets` for equality by ensuring each set contains the other. However, the JavaScript notion of equality (`===`, used internally by the `Set`) does not correspond directly to the `circuitValueEquals()` object, which checks for element-wise equality.

```
1  if (a instanceof Set) {
2    return (
3      b instanceof Set && a.size === b.size && [...a].every((a_) => b.has(a_))
4    );
5  }
```

Snippet 4.56: Snippet from `circuitValueEquals()`

Similarly, `cloneCircuitValue()` clones a `Map` by only cloning the values, not the keys. This means that only the "leaves" of an object structure will be cloned. If a reference object is stored as a key, it will remain in both `Maps`.

Impact The current usage of these methods applies primarily to object layouts used by the Mina ecosystem. These objects do not currently use `Maps` or `Sets`.

If future objects do use `Maps` or `Sets`, surprising errors could occur. For example, a nested map which is cloned will have its leaves cloned, but intermediate keys will reference the same object.

Recommendation Remove support for `Maps` and `Sets`. If this is not possible, then

1. Consider restricting the keys of `Maps` to primitive types.
2. Consider using `circuitValueEquals()` to check equality between sets.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.39 V-O1J-VUL-039: Unsafe construction of UInt64 with FieldVar

Severity	Warning	**Commit**	8dde2c3
Type	Data Validation	**Status**	Acknowledged
File(s)	`src/lib/provable/int.ts`		
Location(s)	constructor() of UInt64, UInt32, UInt8		
Confirmed Fix At	N/A		

A `UInt64` (and other classes like `UInt32`, etc) can be constructed from a `FieldVar` via its constructor.

```
constructor(x: UInt64 | UInt32 | FieldVar | number | string | bigint) {
  if (x instanceof UInt64 || x instanceof UInt32) x = x.value.value;
  let value = Field(x);
  super(value);
  // check the range if the argument is a constant
  UInt64.checkConstant(value);
}
```

Snippet 4.57: Snippet from the constructor of `UInt64`

However, providing a `FieldVar` directly is unsafe because it might not 64 bits. Construction from `FieldVars` is only intended for internal use or advanced users.

Impact The missing range check could cause the circuit to be under-constrained.

For example, if a new o1js user is struggling to make their circuit compile, they may notice that `Field.value` is of type `FieldVar`, and supply it directly to `UInt64` without a range check.

Recommendation There are many possible directions to address this issue.

- If breaking the API is acceptable, one possibility is to mark the `constructor` as `private`, thus enforcing users to use either `Unsafe.fromField` or `from` static methods. This will still allow the constructor to be used privately within the file, but will no longer allow `new UInt64(...)` outside the file.
- Otherwise, consider creating a wrapper type with a special field to indicate lack of safety, and then modify the constructor of `UInt64` to consume this wrapper type instead. E.g.

```
type PrivateFieldVar = { _unsafeFieldVar: FieldVar };
```

  Even better, use the newtype-style typing's "Fake Boxed Type", as suggested earlier in V-O1J-VUL-059, to enforce that `PrivateFieldVar` can't be forged.
- An alternative approach is to omit `FieldVar` from the constructor, and use `Object.setPrototypeOf` to directly set a `FieldVar` to `CircuitValue`.
- As a last resort, adjust the documentation to strongly discourage the use `FieldVar` in the constructor (or strongly discourage the use of the constructor entirely), but make no other changes.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.40 V-O1J-VUL-040: Over-constrained foreign-field operations

Severity	Warning	**Commit**	8dde2c3
Type	Over-constrained Circuit	**Status**	Acknowledged
File(s)	`src/lib/provable/gadgets/foreign-field.ts`		
Location(s)	See issue description		
Confirmed Fix At	N/A		

Fields can be represented in three different types: `Unreduced`, `AlmostReduced`, and `Canonical`. Several operations are defined for `Unreduced` forms, including

- `add()`
- `neg()`
- `sub()`
- `sum()`

The only documented restriction on the representative chosen for an `Unreduced` foreign field element is an upper bound. The element must be in the range `[0, 2^{264})`.

However, all of these elements have implicit range requirements dependent on the foreign field modulus. These implicit requirements stem from the following fact: each time a `ForeignFieldAdd` gate is used in kimchi, *at most one field overflow may occur.*

For example:

- This is explicitly leveraged by `assertLessThan()` in `provable/gadgets/foreign-field.ts`. It asserts that `x < y` for variable `x` and constant `y` by computing the `0-x mod (y-1)`. Since `-x` can only "overflow" by `y-1` at most once, it must be less than `x` in order for the assertions in `sum()` to succeed.

For example, running the below code snippet leads to a failed assertion.

```
class SmallField extends createForeignField(17n) {}

function checkValues() {
  let oneAsEighteen = Provable.witness(
    SmallField.provable,
    () => SmallField.provable.fromFields([new Field(18n), new Field(0n), new Field(0n
    )])
  )
  let negativeOne = oneAsEighteen.neg();
}
```

Snippet 4.58: Code snippet showing `18` cannot be negated modulo `17`.

- More generally, adding together any large representatives may fail. Roughly speaking, this will happen when

$$x_{i,bot} \geq \frac{2^{176}}{n} + f_{bot} \quad x_{i,2} \geq \frac{2^{88}}{n} + f_2$$

Impact The foreign field operations are designed to always work on canonical elements. Users who create foreign field elements from out-of-circuit values and then use only the API will have no issues.

However, if in-circuit values are computed to foreign fields, issues may arise. For example, if the output of a hash function is used to construct a `ForeignField` directly from a `Field3` representation, a user may unknowingly write over-constrained circuits.

Recommendation Document the assumption that these operations are only guaranteed to work with canonical representatives.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.41 V-O1J-VUL-041: Incorrect Merkle witness key computation in prover

Severity	Warning	**Commit**	8dde2c3
Type	Logic Error	**Status**	Fixed
File(s)	`src/lib/provable/merkle-map.ts`		
Location(s)	MerkleMap._keyToIndex()		
Confirmed Fix At	cbf4975		

A `MerkleMap` is represented by a binary tree with 2^{255} leaves. Each leaf "`index`" corresponds to a `Field`, called its `key`. This `key` is computed by representing the index as a 255-bit binary string and then reversing the bits. In `_keyToIndex()`, shown below, the `index` is computed from a `key`.

```
_keyToIndex(key: Field) {
  // the bit map is reversed to make reconstructing the key during proving more
    convenient
  let keyBits = key
    .toBits()
    .slice(0, bits)
    .reverse()
    .map((b) => b.toBoolean());

  let n = 0n;
  for (let i = 0; i < keyBits.length; i++) {
    const b = keyBits[i] ? 1 : 0;
    n += 2n ** BigInt(i) * BigInt(b);
  }

  return n;
}
```

Snippet 4.59: Definition of `_keyToIndex()`

Note here that `bits` is the constant 255. However, `key.toBits()` returns a 254-bit representation of `key`. This is because, for elements of the Pallas field, there may be multiple 255-bit representations of a single element. Consequently, the least significant bit of `key` is weighted by `2n**253` instead of `2n**254`. Similarly, each other bit is off by a factor of 2, leading to computation of the incorrect index.

Impact The key computed by `computeRootAndKey()` from a `MerkleMapWitness` will not match the expected key. One can test this using the below script.

```
// Take an empty map and set the "one key"
const map = new MerkleMap();
const one = new Field(1n);
map.set(one, one);
const oneWitness = map.getWitness(one)
const [compHash, compKey] = oneWitness.computeRootAndKey(one)
assert(compHash.toBigInt() == map.getRoot().toBigInt(), `${compHash} != ${map.getRoot
    ()}`)
assert(one.toBigInt() == compKey.toBigInt(), `${one} != ${compKey}`);
```

This outputs


```
Error: 1 != 2
```

Recommendation Pad `key.toBits()` with zeros out to a length of 255 before reversing. See also V-O1J-VUL-002.

Developer Response This issue was fixed during the audit.

4.1.42 V-O1J-VUL-042: Missing public key validation checks

Severity	Warning	**Commit**	02f2ffb
Type	Data Validation	**Status**	Acknowledged
File(s)	`src/lib/gadgets/elliptic-curve.ts`		
Location(s)	verifyEcdsaConstant()		
Confirmed Fix At	N/A		

The `verifyEcdsaConstant()` is the "constant" case of `verifyEcdsa()`, i.e. it verifies an ECDSA signature when all of the supplied values are non-variable. Its definition is shown below.

```
function verifyEcdsaConstant(
  Curve: CurveAffine,
  { r, s }: Ecdsa.signature,
  msgHash: bigint,
  publicKey: point
) {
  let pk = Curve.from(publicKey);
  if (Curve.equal(pk, Curve.zero)) return false;
  if (Curve.hasCofactor && !Curve.isInSubgroup(pk)) return false;
  if (r < 1n || r >= Curve.order) return false;
  if (s < 1n || s >= Curve.order) return false;

  let sInv = Curve.Scalar.inverse(s);
  assert(sInv !== undefined);
  let u1 = Curve.Scalar.mul(msgHash, sInv);
  let u2 = Curve.Scalar.mul(r, sInv);

  let R = Curve.add(Curve.scale(Curve.one, u1), Curve.scale(pk, u2));
  if (Curve.equal(R, Curve.zero)) return false;

  return Curve.Scalar.equal(R.x, r);
}
```

Snippet 4.60: Definition of `verifyEcdsaConstant()`

Note that the public key is validated to be non-zero. For curves with a co-factor, a subgroup check is also performed. However, as mentioned in V-O1J-VUL-057, the implementation of `isInSubgroup()` assumes the provided point is on the curve.

In fact, Section 6.4.2 of NIST's Digital Signatures Standards, which lays out standards for performing the ECDSA Signature Validation Algorithm, recommends

> From Section 6.2.2, the validity of the domain parameters shall be assured prior to the verification and validation of a digital signature. The validity of the public key Q should also be checked (see Appendix D.1 of SP 800-186 [5]).

The referenced source [5] is NIST's Recommendations for Discrete Logarithm-based Cryptography. Algorithm D.1.1.1 in Appendix D.1. recommends performing the following additional validations on the public key (called `Q` in their notation):

1. Let `Q = (x, y)`. Verify that `x` and `y` are integers in the interval `[0, p-1]`. Output REJECT if verification fails.

2. Verify that `(x, y)` is a point on `W_{a,b}` by checking that `(x, y)` satisfies the defining equation `y2 = x3 + a x + b`, where computations are carried out in `GF(p)`. Output REJECT if verification fails.

Impact Signatures from invalid public keys may be validated by this algorithm. This may lead to differences in behavior with other popular ECDSA implementations.

Further, some security analyses of ECDSA assume that the supplied point is on the curve. See this StackExchange post for a deeper discussion.

Recommendation Perform full validation of the ECDSA public key, as recommended in Appendix D.1 of NIST's Recommendations for Discrete Logarithm-based Cryptography.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.43 V-O1J-VUL-043: New layout types may lead to hash collisions

Severity	Warning	**Commit**	3c68a0d
Type	Hash Collision	**Status**	Acknowledged
File(s)	`o1js-bindings/lib/from-layout.ts`		
Location(s)	SignableFromLayout.toInput()		
Confirmed Fix At	N/A		

The method `toInput()` of `SignableFromLayout` can be used to fold a value to a single hash input. This is done by flatly accumulating `fields` and `packed` across key-value pairs for an object, and across elements for an array.

```
1  function toInput(typeData: Layout, value: any) {
2    return layoutFold<any, HashInput>(
3      {
4        // [VERIDISE] ...elided...,
5        reduceArray(array) {
6          let acc: HashInput = { fields: [], packed: [] };
7          for (let { fields, packed } of array) {
8            if (fields) acc.fields!.push(...fields);
9            if (packed) acc.packed!.push(...packed);
10         }
11         return acc;
12       },
13       reduceObject(keys, object) {
14         let acc: HashInput = { fields: [], packed: [] };
15         for (let key of keys) {
16           let { fields, packed } = object[key];
17           if (fields) acc.fields!.push(...fields);
18           if (packed) acc.packed!.push(...packed);
19         }
20         return acc;
21       },
22       // [VERIDISE] ...elided...
23     },
24     typeData,
25     value
26   );
27 }
```

Snippet 4.61: Snippet from `toInput()`

However, the above method could cause undesirable collisions, such as how the hash inputs of `{a: [1], b: [2]}` and `{a: [1, 2], b: []}` would be the same.

To see why, note that the recursion is facilitated by `layoutFold`, which in turn calls `genericLayoutFold`.

```
1 function genericLayoutFold ...elided... {
2   // [VERIDISE] ...elided...
3   if (typeData.type === 'array') {
4     let arrayTypeData = typeData as ArrayLayout<TypeMap>;
5     let v: T[] | undefined[] | undefined = value as any;
6     if (arrayTypeData.staticLength !== null && v === undefined) {
```



```
      v = Array<undefined>(arrayTypeData.staticLength).fill(undefined);
    }
    let array =
      v?.map((x) =>
        genericLayoutFold(TypeMap, customTypes, spec, arrayTypeData.inner, x)
      ) ?? [];
    return spec.reduceArray(array, arrayTypeData);
  }
  // [VERIDISE] ...elided...
}
```

Snippet 4.62: Snippet from `genericLayoutFold()`

For the most part, `typeData` controls how the recursion is done (e.g., the order of keys to iterate in the object case). However, for an array type, there are two issues:

- the `array` to be reduced is mapped from `v` / `value`. `typeData` is (usually) not consulted.
- `staticLength` could be `null`.

This means it is possible to create a single `typeData` which hashes two different values to the same hash input.

A similar problem exists when handling optionals which are missing. For example, given a `typeData` of `{a?: Bool, b?: Bool}`, `[Bool(true)]` could correspond to `{a: Bool(true)}` or `{b: Bool(true)}`. This can occur more generally when two or more variable-length fields appear in a `typeData`.

Furthermore, different `typeData`s may also map to the hash inputs:

- The hash inputs of `1` and `[1]` would be the same.
- The hash inputs of `{a: 1}` and `{b: 1}` would be the same.

In the above cases, `typeData` would be different, but the hash inputs would be the same.

Similar issues appear in `createHashInput()` of `src/bindings/lib/provable-generic.ts`.

Impact The variable-length data types issue is significant, since the hash collision could happen even when `typeData`s are the same. This could cause hash functions defined over custom data types to have easily-computable collisions. This could, for example, lead to security breaches for nullifiers.

Moreover, while it is true that hash collision for the cross-type case is somewhat "benign", because callers could use `typeData` to separate the hash inputs, this requires careful use. It would be better to create a hashing scheme that avoids the collision entirely by ensuring a domain separator is included when necessary.

Luckily, the APIs are currently only for internal use, and these hash collisions do not appear in any of the current use cases.

Recommendation Instead of flatly accumulating `fields` and `packed`, further record information, such as tagging it with types, array length, and key names. See, for example, EIP 712 for an approach to structured hashing.

We also recommend adding runtime-checks to ensure packed elements fit in the expected number of bits (e.g. when calling `toInput()` for `Bool`, `ForeignFields`, or `UInt*`, or calling `provable()` in `foreign-field.ts`).

Developer Response We do not use `HashInput` for variable-length types. We will consider throwing an error when calling `HashInput` on a variable-length type.

Updated Veridise Response After further review of the usage of layouts within the repository, we have downgraded this issue to a Warning.

4.1.44 V-O1J-VUL-044: Missing high-limb constraints when bit-slicing

Severity	Warning	**Commit**	8dde2c3
Type	Under-constrained Circuit	**Status**	Acknowledged
File(s)	src/lib/provable/gadgets/bit-slices.ts		
Location(s)	sliceField3()		
Confirmed Fix At	N/A		

The `sliceField3()` function decomposes a `Field3` into its binary representation, optionally allowing to chunk the bits into larger sizes. For example, calling `sliceField3()` with a `chunkSize` of `1` would return the binary decomposition, while a `chunkSize` of `8` would decompose the value into bytes.

Users also provide a `maxBits` value. The returned array is the smallest length which can represent `maxBits` bits in chunks of size `chunkSize` (i.e. $\lceil \frac{\text{maxBits}}{\text{chunkSize}} \rceil$). According the function comment, `sliceField3()` simultaneously decomposes `x` into `chunkSize`-bit chunks, but also proves $x \leq 2^{\text{maxBits}}$.

However, as shown below, `sliceField3()` returns early when maxBits ≤ 176.

```
function sliceField3(
  [x0, x1, x2]: Field3,
  { maxBits, chunkSize }: { maxBits: number; chunkSize: number }
) {
  let l_ = Number(l);
  assert(maxBits <= 3 * l_, `expected max bits <= 3*${l_}, got ${maxBits}`);

  // first limb
  let result0 = sliceField(x0, Math.min(l_, maxBits), chunkSize);
  if (maxBits <= l_) return result0.chunks;
  maxBits -= l_;

  // second limb
  let result1 = sliceField(x1, Math.min(l_, maxBits), chunkSize, result0);
  if (maxBits <= l_) return result0.chunks.concat(result1.chunks);
  maxBits -= l_;

  // third limb
  let result2 = sliceField(x2, maxBits, chunkSize, result1);
  return result0.chunks.concat(result1.chunks, result2.chunks);
}
```

Snippet 4.63: Definition of `sliceField3()`

While it is unnecessary to compute the excess bits, the high limbs must still be constrained in order for the $x \leq 2^{\text{maxBits}}$ check to be valid.

In particular, when `maxBits <= l_`, `x1` must be constrained to be zero. Similarly, when `maxBits <= 2l_`, `x2` must be constrained to be zero.

Impact Functions relying on `sliceField3()` to perform a range check may be using unconstrained values. Fortunately, this function is only used internally, and only in one location:

`multiScalarMul()` within `gadgets/elliptic-curve.ts` (exposed to the user through `const EllipticCurve`).

`multiScalarMul()` relies on `sliceField3()` to range check each `Field3` in the provided `scalars` argument. They are then used in `decomposeNoRangeCheck()`. The documentation written for `decomposeNoRangeCheck()` indicates the range check is elided explicitly due to the use of `sliceField3()`. This will lead to an under-constrained `Sum`, and potentially allow a malicious prover to provide an invalid decomposition.

However, the `maxBits` used is `Curve.Scalar.sizeInBits`. So, this would only affect use cases where the curve scalar field is at most 176 bits (in the GLV case `Curve.Endo.decomposeMaxBits` is used, but then the upper limbs of the scalars are hard-coded to zero, so 176 bits is still the cutoff). Most commonly used cryptographic curves do not fall into this category.

Recommendation Require that the upper limbs are 0 when `maxBits <= l_` and `maxBits <= 2l_`.

Also, require that `chunkSize <= l_`.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.45 V-O1J-VUL-045: Avoid mutating config input

Severity	Warning	**Commit**	8dde2c3
Type	Logic Error	**Status**	Acknowledged
File(s)	src/lib/provable/gadgets/elliptic-curve.ts		
Location(s)	scale()		
Confirmed Fix At	N/A		

The function `scale()` performs an EC scalar multiplication.

```
function scale(
  scalar: Field3,
  point: Point,
  Curve: CurveAffine,
  config: {
    mode?: 'assert-nonzero' | 'assert-zero';
    windowSize?: number;
    multiples?: Point[];
  } = { mode: 'assert-nonzero' }
) {
  config.windowSize ??= Point.isConstant(point) ? 4 : 3;
  return multiScalarMul([scalar], [point], Curve, [config], config.mode);
}
```

Snippet 4.64: Snippet from `scale()`

Users can call the function with `config` to configure how the scaling should be done. When `windowSize` is not given, the function provides a default value by mutating `config` to set `windowSize`.

However, if `config` is supplied with no `windowSize` and then reused for multiple scaling operations, the mutation could undesirably affect the next calls. Consider the following snippet

```
let config = {mode: 'assert-nonzero'};
scale(scalar, constantPoint, Curve, config);
scale(scalar, nonConstantPoint, Curve, config);
```

The intention here is that both scaling is done with the `{mode: 'assert-nonzero'}` config, which should mean the first scaling has `windowSize` of 4, and the second scaling has `windowSize` of 3. However, due to the mutation, after the first call, `config` is mutated to `{mode: 'assert-nonzero', windowSize: 4}`, so the second scaling also has `windowSize` of 4 as well.

Impact There is no `config` reuse in the current callsites of `scale` in the codebase, and it is not public-facing. Therefore, the impact is minimal. Still, there is a risk that the reuse could happen in a future update.

Recommendation Avoid mutating the input directly. One possibility is to clone the `config` first before perform the mutation on the clone. Another possibility is to add the default value immutably. E.g.,

```
let newConfig = config.windowSize ?
  config :
  { ...config, windowSize: Point.isConstant(point) ? 4 : 3};
```

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.46 V-O1J-VUL-046: toFields depends on declaration order

Severity	Warning	**Commit**	3c68a0d
Type	Maintainability	**Status**	Acknowledged
File(s)	o1js-bindings/lib/provable-generic.ts		
Location(s)	provable(), signable()		
Confirmed Fix At	N/A		

`Object.keys()` returns keys in an order dependent on their definition. As can be seen below, this means that "to-field" serialization order depends on the order-of-definition of different properties.

```
return (isToplevel ? objectKeys : Object.keys(typeObj))
  .map((k) => toFields(typeObj[k], obj[k]))
  .flat();
```

Snippet 4.65: Snippet from `provable().toFields()`

This can lead to some interesting behaviors. For example, two types which one might expect to be equal have different serializations.

```
import { Field } from '../../lib/core.js';
let Point2D = {
  x: Field,
  y: Field,
};

let OtherPoint2D = {
  y: Field,
  x: Field,
}

let {provable}  = createDerivers<Field>();

let ProvablePoint2D = provable(Point2D);
let ProvableOtherPoint2D = provable(OtherPoint2D);

let p1 = {x: Field(0n), y: Field(1n)};
let p2 = {y: Field(1n), x: Field(0n)};

console.log("     Point2D: p1 =", ProvablePoint2D.toFields(p1).map(f => f.toBigInt())
    )
console.log("     Point2D: p2 =", ProvablePoint2D.toFields(p2).map(f => f.toBigInt())
    )
console.log("OtherPoint2D: p1 =", ProvableOtherPoint2D.toFields(p1).map(f => f.
    toBigInt()))
console.log("OtherPoint2D: p2 =", ProvableOtherPoint2D.toFields(p2).map(f => f.
    toBigInt()))
```

This outputs

```
     Point2D: p1 = [ 0n, 1n ]
     Point2D: p2 = [ 0n, 1n ]
OtherPoint2D: p1 = [ 1n, 0n ]
OtherPoint2D: p2 = [ 1n, 0n ]
```

Impact Users may not expect changing the field declaration order to change the interpretation of serialized structures.

Recommendation Sort the `Object.keys()` to produce an order which only depends on the `keys`.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.47 V-O1J-VUL-047: Failure of group addition during witness generation

Severity	Warning	**Commit**	8dde2c3
Type	Logic Error	**Status**	Fixed
File(s)	src/lib/provable/group.ts		
Location(s)	add()		
Confirmed Fix At	2eb6354		

The method `add()` performs group addition of two group elements.

```
// [VERIDISE - ELIDED]
// similarly to the constant implementation, we check if either operand is zero
// and the implementation above (original OCaml implementation) returns something wild -> g + 0 != g where it should be g + 0 = g
let gIsZero = g.isZero();
let onlyThisIsZero = this.isZero().and(gIsZero.not());
let isNegation = inf;
let isNormalAddition = gIsZero.or(onlyThisIsZero).or(isNegation).not();

// note: gIsZero and isNegation are not mutually exclusive, but if both are true, we add 1*0 + 1*0 = 0 which is correct
return Provable.switch(
  [gIsZero, onlyThisIsZero, isNegation, isNormalAddition],
  Group,
  [this, g, Group.zero, new Group({ x, y })]
);
```

Snippet 4.66: Snippet from `add()`

The method eventually returns a value that could come from four possibilities. As the comment points out, `gIsZero` and `isNegation` are not mutually exclusive and can be both true.

However, in such case, the witness generation would fail, as `Provable.switch` makes sure that at most one conditional expression can be true.

```
// [VERIDISE - ELIDED]
let checkMask = () => {
  let nTrue = mask.filter((b) => b.toBoolean()).length;
  if (nTrue > 1) {
    throw Error(
      `Provable.switch: \`mask\` must have 0 or 1 true element, found ${nTrue}.`
    );
  }
};
// [VERIDISE - ELIDED]
```

Snippet 4.67: Snippet from `switch()`

Impact The prover would unexpectedly fail.

Recommendation Either an option to allow `switch` to take multiple conditionals that are true should be added, or the conditionals should be restructured so that at most one is true.

Developer Response Fixed in the provided commit by adding an option to allow non-exclusive conditionals.

4.1.48 V-O1J-VUL-048: Possible issues during serialization

Severity	Warning	**Commit**	e7ded4b
Type	Logic Error	**Status**	Acknowledged
File(s)	`o1js-bindings/lib/{from-layout.ts, provable-generic.ts}`		
Location(s)	See issue description		
Confirmed Fix At	N/A		

In the below issue, we break down a few possible issues which may occur when serializing or de-serializing custom provable types.

1. `lib/from-layout.ts` provides several utilities for applying functions to pre-defined object layouts. The specified layout allows for 3 different kinds of optionals, shown below.

```
type OptionLayout<TypeMap extends AnyTypeMap, T = BaseLayout<AnyTypeMap>> = {
  type: 'option';
} & (
  | {
      optionType: 'closedInterval';
      rangeMin: any;
      rangeMax: any;
      inner: RangeLayout<TypeMap, T>;
    }
  | {
      optionType: 'flaggedOption';
      inner: T;
    }
  | {
      optionType: 'orUndefined';
      inner: T;
    }
) &
  WithChecked<TypeMap>;
```

Snippet 4.68: Type definition of `OptionLayout`.

The first two option types (`'closedInterval'` and `'flaggedOption'`) correspond to objects with a type extending `{isSome: TypeMap['Bool'], value: T}`. The third type corresponds to an object of type `T | undefined`.
When folding, these two classes of option are handled separately. The `'orUndefined'` case is shown below.

```
case 'orUndefined':
  let mapped =
    value === undefined
      ? undefined
      : genericLayoutFold(TypeMap, customTypes, spec, inner, value);
  return spec.reduceOrUndefined(mapped, inner);
```

Snippet 4.69: `'orUndefined'` case of an `OptionLayout` in `genericLayoutFold()`.

In many cases, the usage of `genericLayoutFold()` seems to assume that `reduceOrUndefined()` is only called when `value === undefined`. As can be seen in the above snippet, this is not the case. For example, consider the below implementation of `reduceOrUndefined()` when calling `layoutFold()` in `ProvableFromLayout().toFields()`.

```
reduceOrUndefined(_) {
  return [];
},
```

Snippet 4.70: Implementation of `FoldSpec.reduceOrUndefined()` used by `ProvableFromLayout().toFields()`.

This function ignores the `mapped` value, dropping a `value` even if it is defined. We can see this is indeed the case by running the below code snippet, which outputs `[]`.

```
let layout = {
    type: 'object',
    name: 'TestLayout',
    docs: null,
    keys: ['value'],
    entries: {
      value: {
        type: 'option',
        optionType: 'orUndefined',
        inner: { type: 'UInt32' },
      }
    }
};

type LayoutT = {
  value?: UInt32
}

let Layout = provableFromLayout<LayoutT, LayoutT>(layout as any);
console.log(Layout.toFields({value: new UInt32(3)})); // Outputs []
console.log(Layout.toAuxiliary({value: new UInt32(3)})); // Outputs [ [ true, [] ] ]
```

Note that this is not an issue for the other two types of optional values.
Similar issues occur for `'orUndefined'` optionals in the following locations within the same file:

- `sizeInFields()`
- `toInput()`

2. `lib/provable-generic.ts` provides a generic method of transforming user-defined types built out of `Provable` types into larger, composite `Provables`. However, it defaults to the base-case of a defined `Provable` whenever one of the expected methods is found. For example, when serializing into fields, if a `typeObj` has `toFields()` defined, that method is used.

This may lead to issues when objects implement only a subset of the provable interface (likely by accident). For example, if a `typeObj` has `toFields()` but no `fromFields()`, serialization will succeed but de-serialization will fail. This may not be obvious to the user if they only serialize the object (for use in the circuit) inside their application.

Impact

1. When `'orUndefined'` values are serialized into witnesses, they will always be treated as `None`.

2. Types which have name collisions with some of the `Provable` method names may be incorrectly serialized.

Recommendation

1. Include the fields present in an `Optional` when calling `toFields()`, `sizeInFields()`, and `toInput()`.
2. Check for types which implement a partial interface when calling `provable()`.

Developer Response

1. This is an anticipated potential problem. We intend the caller to ensure that no in-circuit values are stored within dynamically-sized objects such as a variable-length array or an `'orUndefined'` optional type.
2. The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

Updated Veridise Response We downgraded the severity of this issue, as the layouts are only used internally and not exposed to the user.

4.1.49 V-O1J-VUL-049: No override of check() for derived leaf types

Severity	Warning	**Commit**	3c68a0d
Type	Data Validation	**Status**	Acknowledged
File(s)	`o1js-bindings/minda-transaction/derived-leaves.ts`		
Location(s)	derivedLeafTypes()		
Confirmed Fix At	N/A		

The `derivedLeafTypes()` function creates several new `Provable` types using the `provable()` function. The below snippet shows the creation of the `TokenSymbol` and `AuthRequired` types.

```
TokenSymbol: createTokenSymbol(
  provable({ field: Field, symbol: String }),
  Field
),
AuthRequired: createAuthRequired(
  provable({
    constant: Bool,
    signatureNecessary: Bool,
    signatureSufficient: Bool,
  }),
  Bool
),
```

Snippet 4.71: Snippet from `derivedLeafTypes()`

Recall that the `check()` function defined by `provable()` applies `Bool.check()` and `Field.check()` as appropriate to the component `Bools` and `Fields` which make up the composite types `TokenSymbol` and `AuthRequired`.

However, the `TokenSymbol` and `AuthRequired` types require more than just type correctness of each field.

- `TokenSymbol`:
  - A token symbol must be representable in 6 bytes, i.e. `field < 2n**48n` must be true.
- `AuthRequired`:
  - As shown in the below code snippet, only certain combinations of the three booleans are supported.

```
let c = Number(Bool.toJSON(x.constant));
let n = Number(Bool.toJSON(x.signatureNecessary));
let s = Number(Bool.toJSON(x.signatureSufficient));
// prettier-ignore
switch (`${c}${n}${s}`) {
  case '110': return 'Impossible';
  case '101': return 'None';
  case '000': return 'Proof';
  case '011': return 'Signature';
  case '001': return 'Either';
  default: throw Error('Unexpected permission');
}
```

Snippet 4.72: Snippet from `createAuthRequired().toJSON()`:

These additional checks are not performed when calling `TokenSymbol.check()` or `AuthRequired.check()`.

Impact A user creating a `TokenSymbol` or `AuthRequired` in a proof from a call to `Provable.witness()` will expect `check()` to validate the returned type is valid. This assumption will not hold.

For most applications involving `TokenSymbol`, this is likely not a serious problem. However, for `AuthRequired`, this could be more problematic. For example, a user could write a circuit which uses `AuthRequired` for their own application-specific purposes. If they perform validation like shown below, then users who set invalid `AuthRequireds` in the application would have no checks performed at all.

```
let permissionFromUser = Provable.witness(AuthRequired, ...)

let authorization = 0
for validPermission in ['110', '101', '000', '011', '001']:
    let selector2 = (permissionFromUser == validPermission)
    let selector = selector1 * selector2
    authorization += selector * authorizationCircuit(validPermission)

require(authorization == 0)
```

Recommendation Add a `check()` function which validates the additional invariants for `TokenSymbol` and `AuthRequired`.

Developer Response There are certain cases where witnessed account updates elide checks on the types because the node will check the `AuthRequired` status.

4.1.50 V-O1J-VUL-050: Missing length check in dot product

Severity	Warning	**Commit**	e7ded4b
Type	Data Validation	**Status**	Acknowledged
File(s)	`o1js-bindings/crypto/finite-field.ts`		
Location(s)	Field.dot()		
Confirmed Fix At	N/A		

The `Field.dot()` method computes a dot product on two vectors of field elements.

```
dot(x: bigint[], y: bigint[]) {
  let z = 0n;
  let n = x.length;
  for (let i = 0; i < n; i++) {
    z += x[i] * y[i];
  }
  return mod(z, p);
}
```

Snippet 4.73: Definition of `dot()`

No check is performed on `x` and `y` to ensure they are the same length. When `y` is shorter than `x`, the `x.length`th multiplication will fail. When `x` is shorter than `y`, the function will ignore the end of the `y` array.

Impact Some usual properties of the `dot()` function will not hold. For example, `dot(x,y) != dot(y,x)` in all cases. For example, `dot([1n], [2n, 3n])` will return `2n`, while `dot([2n, 3n], [1n])` will provoke an error.

Users may also spend more time debugging if they expect the `dot` function to perform length checks.

Recommendation Require `x` and `y` to be the same length.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.51 V-O1J-VUL-051: Typos and incorrect comments

Severity	Info	**Commit**	e7ded4b
Type	Maintainability	**Status**	Acknowledged
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

Description In the following locations, the auditors identified minor typos, potentially misleading comments, and unexpected naming conventions:

- `o1js-bindings/crypto/elliptic-curve.ts`:
  - `pallasGeneratorProjective`: This point is defined as an `(x, y)`-coordinate pair. While this can be viewed as a projective point with `z = 1`, it is more natural to name this `pallasGeneratorAffine`.
  - `vestaGeneratorProjective`: As in the `pallasGeneratorProjective` case, only the `(x, y)`-coordinates are defined, and the `z`-coordinate is implicitly set to 1.

- `o1js-bindings/crypto/bigint-helpers.ts`:
  - The names `bigintToBytes32`, `bigintToBytesFlexible`, and `bytesToBigint32` are inconsistent with `bytesToBigInt` regarding the capitalization of "I".

- `o1js-bindings/crypto/elliptic-curve-endomorphism.ts`:
  - `glvScaleProjective`: Missing backtick in the documentation: ‘double()->double()“.
  - `egcdStopEarly`: The error message should be `l > p`, not `a > p`.

- `src/lib/provable/crypto/keccak.ts`:
  - `piRho()`: The comment has the definitions of `pi` and `rho` flipped.

- `o1js/src/lib/provable/field.ts`:
  - `Field.mul()`: The function documentation describes the return value as "`the modular difference of the two value.`," instead of the modular product.
  - `assertNotEquals()`: `x.equals(y).assertTrue()` should have instead been `x.equals(y).assertFalse()`

- `src/lib/provable/gadgets/elliptic-curve.ts`:
  - `getPointTable()`: This method claims to return `[0, P, 2P, 3P, ..., (2^windowSize-1) * P]`. However, `0` is not representable as a `Point`, and the first element in the returned array should be considered an unusable contaminated value. This should be documented.

- `scale()`: the documentation claims that "The result is constrained to be not zero." However, this is only true when the `mode` field in the `config` parameter is `assert-nonzero`.

▶ `o1js/src/lib/provable/group.ts`:

- `Group.scale()`: in the documentation, `Scalar(5)` should be changed to `Scalar.from(5)`. `5g` is also an invalid identifier.

▶ `o1js/src/lib/provable/int.ts`:

- `UInt32.toBigint()`: The function should be renamed to `UInt32.toBigInt()` to match similarly named functions elsewhere in the repository.
- `Int64.div()`: the documentation claims that "`x.div(y)` returns the floor of `x / y`, that is, the greatest `z` such that `z * y <= x`". However, this is false when the numerator is negative. E.g., `-17.div(2) = -8`, but "greatest `z` such that `z * y <= x`" would actually be `-9`.

▶ `o1js/src/lib/mina/account-update.ts`:

- `Body`: the below links are outdated
  - `https://docs.minaprotocol.com/zkapps/advanced-o1js/events`
  - `https://docs.minaprotocol.com/zkapps/advanced-o1js/actions-and-reducer`

Impact These minor errors may lead to future developer confusion.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.52 V-O1J-VUL-052: Incorrect squeeze computation

Severity	Info	**Commit**	8dde2c3
Type	Maintainability	**Status**	Acknowledged
File(s)	`src/lib/provable/crypto/keccak.ts`		
Location(s)	squeeze()		
Confirmed Fix At	N/A		

The `squeeze()` function computes a value `squeezes`, representing how many times the permutation must be applied when using Keccak, plus one. As can be seen in the below snippet, it is computed as the floor or `length / rate`, plus one.

```
function squeeze(state: State, length: number, rate: number): Field[] {
  // number of squeezes
   const squeezes = Math.floor(length / rate) + 1;
  assert(squeezes === 1, 'squeezes should be 1');
```

Snippet 4.74: Snippet from `squeeze()`

This is not quite correct. For example, for any `length <= rate`, the permutation need not be applied (so `squeezes` should be 1). However, when `length == rate`, `squeezes` is computed as `2`.

Impact Future changes supporting longer outputs may compute incorrect results.

Recommendation Compute the ceiling of `length / rate`, and assert `squeezes <= 1` instead of equal.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.53 V-O1J-VUL-053: Missing hash-to-curve best practices

Severity	Info	**Commit**	8dde2c3
Type	Maintainability	**Status**	Acknowledged
File(s)	`src/lib/provable/gadgets/elliptic-curve.ts`		
Location(s)	See issue description		
Confirmed Fix At	N/A		

IETF RFC 9380: Hashing to Elliptic Curves provides several standard recommendations which could improve the protocol:

- Rather than using the domain separation tag `'initial-aggregator'` in the `initialAggregator()` function, consider following the tag guidelines provided in Section 3.1: Domain Separation Requirements.
- Rather than `simpleMapToCurve()`, which may be vulnerable to timing side channels if the provided randomness is poor, implement one of the hash functions outlined in Mappings for Weierstrass Curves, or consider using one of the standard Suites for Hashing on well-known curves.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.54 V-O1J-VUL-054: Over-constrained neg()

Severity	Info	**Commit**	8dde2c3
Type	Maintainability	**Status**	Acknowledged
File(s)	o1js/src/lib/provable/foreign-field.ts		
Location(s)	ForeignField.neg()		
Confirmed Fix At	N/A		

The method `neg()` performs the foreign field negation

```
1  neg() {
2    // this gets a special implementation because negation proves that the return
     value is almost reduced.
3    // it shows that r = f - x >= 0 or r = 0 (for x=0) over the integers, which
     implies r < f
4    // see also 'Gadgets.ForeignField.assertLessThan()'
5    let xNeg = Gadgets.ForeignField.neg(this.value, this.modulus);
6    return new this.Constructor.AlmostReduced(xNeg);
7  }
```

Snippet 4.75: Snippet from `neg()`

However, this method only works correctly when the representative of `x` is less than or equal to `f`. As a result, when the method is called on a non-constant value that is greater than `f`, the circuit could become over-constrained, which could be unexpected from users.

```
1  class MyField extends createForeignField(17n) {}
2
3  function main() {
4    let x = Provable.witness(
5      MyField.provable,
6      () => new MyField([Field(18), Field(0), Field(0)])
7    );
8
9    let y = new MyField(16);
10
11   y.assertEquals(x.neg());
12 }
13
14 await Provable.runAndCheck(main);
15
16 /*
17 [Error: multi-range check failed
18 Constraint unsatisfied (unreduced):
19
20 Constraint:
21 ((basic(Equal(Var 61)(Var 78)))(annotation()))
22 Data:
23 Equal 289480223093290488558927462521719769633630564819415607159546767643499676303369305474064456840531410944 0]
24 */
```

The documentation comment for `neg` is also slightly incorrect: the resulting value is not constrained to be strictly less than `f`. It may also be equal to `f`.

Impact The circuit could be over-constrained when the implicit precondition is violated.

Recommendation The comment should be corrected: `r <= f`. Also, the method should clearly document the precondition that the value must already be constrained to be less than or equal to `f`.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.55 V-O1J-VUL-055: console.assert() only prints error

Severity	Info	**Commit**	e7ded4b
Type	Maintainability	**Status**	Acknowledged
File(s)	`o1js-bindings/crypto/bigint-helpers.ts`		
Location(s)	fromBase()		
Confirmed Fix At	N/A		

The `fromBase()` function uses console.assert(), which prints a message if the assertion fails, but does not abort execution.

Impact If the assertion fails, an error will be printed. However, this error message could easily be missed, and may mislead users to think that everything went well when there is in fact an assertion error.

Recommendation Use `assert()` defined in `o1js/src/lib/errors.ts`.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.56 V-O1J-VUL-056: Use bitlen() instead of log2()

Severity	Info	**Commit**	3c68a0d
Type	Logic Error	**Status**	Acknowledged
File(s)	`o1js-bindings/crypto/finite-field.ts`		
Location(s)	fastInverse()		
Confirmed Fix At	N/A		

`fastInverse()` uses `log2()` to compute the length of a bit-string, instead of `bitlen()`.

```
1 let len = Math.max(log2(u), log2(v));
```

Snippet 4.76: Snippet from `fastInverse()`

When `u` or `v` is a power of 2, `log2()` and `bitlen()` will return different results.

Impact This did not manifest into any concrete issues after several hours of fuzzing and manual analysis. However, it is possible that, for some particular edge case, the loop will incorrectly stop early and an incorrect inverse will be computed, leading to an assertion failure.

Recommendation Use `bitlen()` instead of `log2()`.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.57 V-O1J-VUL-057: Recommended documentation

Severity	Info	**Commit**	8dde2c3, e7ded4b
Type	Maintainability	**Status**	Acknowledged
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

Description In the following locations, the auditors recommend adding function documentation to make certain assumptions/invariants as clear as possible to users of the library.

1. `src/snarky.d.ts`
   a) `inProver()`, `asProver()`: We recommend adding additional documentation clarifying the difference between these two states.

---

2. `src/bindings/crypto/bigint-helpers.ts`
   a) `parseHexString32()` expects the bytes in the string to be in little-endian order, which is not the usual ordering of a hex-string.

---

3. `src/bindings/crypto/elliptic-curve.ts`
   a) Several implementations assume that the elliptic curve `y^2 = x^3 + ax + b` has an `a`-coefficient of zero. While this is validated in `createCurveProjective()` and `createCurveAffine()`, the following helper functions rely on the `a = 0` assumption. Some make mention of these facts inline, others make no mention of it. Note that some functions do support `a != 0` (such as `affineOnCurve()` and `affineAdd()`), while the ones listed below *assume* `a = 0`.
      - `projectiveAdd()`
      - `projectiveDouble()`
      - `projectiveSub()` (via its dependence on `projectiveAdd()`)
      - `projectiveScale()` (via its dependence on `projectiveAdd()` and `projectiveDouble()`)
      - `projectiveOnCurve()`
      - `projectiveInSubgroup()` (via its dependence on `projectiveEqual()`)
      - `affineDouble()`: This function elides `a` from the formula, assuming it is 0. This will be silently incorrect if used with curves with `a = 0`.
      - `affineScale()` (via its dependence on `projectiveScale()`).
      - Other functions assume a representation `(x/z^2, y/z^3)` which is not commonly used for many fields with `a != 0`.
        - `projectiveToAffine()`
        - `projectiveEqual()`

   b) As noted in the body of `projectiveOnCurve()`, when `z = 0`, the `projectiveOnCurve()` implementation only returns true for coordinates `(x, y, z)` with `y^2 = x^3`. This effectively imposes an invariant on any curve operation which may output the

identity element in projective coordinates. The method must ensure that the representation chosen for zero satisfies `y^2 = x^3`. This requirement should be documented on both the class, and on the method documentation for `projectiveOnCurve()`.

c) `projectiveInSubgroup()` assumes that the provided element is on the curve. This fact is not documented. See related issue V-O1J-VUL-042.

d) `createCurveProjective()` and `createCurveAffine()` use the argument `order` as the order of the subgroup. However, this is ambiguous, as it could also be interpreted as the order of the curve.

e) `createCurveProjective()` sets the `cofactor` to `1` if the argument is passed as `undefined`. However, `createCurveAffine()` leaves the `cofactor` as `undefined`. This should be documented, or made consistent.

f) The `Point` type should be documented its intended invariant, which is that the `x` and `y` components are almost reduced.

---

4. `src/bindings/crypto/finite-field.ts`

a) `fastInverse()`: Consider documenting each parameter to this function, as they must be set to very particular values. In Pornin's paper, the algorithm additionally assumes `kmax > 1`. While `fastInverse()` is never used with `kmax == 1`, adding an assertion may ease reasoning about the code.

b) `fastInverse()`: The constants `w = 31` and `hiBits = 31` guarantee that several computed values stay within the 53-bits which is safe for JavaScript `Numbers`. Consider adding documentation describing this fact to the constant definitions to ensure no future changes introduce a bug.

---

5. `src/bindings/lib/from-layout.ts`

a) `GenericFoldSpec.map()`: This function has three arguments, shown below. `type` corresponds to a type registered in the `TypeMap`, `name`is the name of the `type`, not of the `value`. The interpretation of each argument should be added as a function comment to improve clarity.

```
type GenericFoldSpec<T, R, TypeMap extends AnyTypeMap, BaseType> = {
  map: (type: BaseType, value?: T, name?: string) => R;
```

Snippet 4.77: Signature of `GenericFoldSpec.map()`

---

6. `src/bindings/ocaml/lib/snarky_bindings.ml`

a) `truncate_to_bits16()`: The documentation of this function should note that the truncation is not proved, i.e. that the constraints only prove that the result is range-checked.

---

7. `src/lib/mina/events.ts`

a) We recommend documenting the ordering of `fromList` (for `Events` and `Actions`) with respect to `pushEvent`, since the order could be surprising. i.e., the first element in the array is not the first to be pushed.

8. `src/lib/mina/zkapp.ts`

   a) `wrapMethod()`: This method performs the core logic of the `@method` decorator. As part of its invocation, it `unlink()`s any input `AccountUpdate` arguments from their current parent, so that the method can link it to itself.

   This is not documented clearly on the API. Further, other cases such as objects with `AccountUpdate` fields or tuples of `AccountUpdates` will behave differently. This may cause some angst to users attempting to perform complex operations on `AccountUpdates`. 2. `self()`: Note that calling `this.self` within a smart-contract execution context updates the `#executionState`, effectively mutating `this.self`.
   To help zkapp developers understand the meaning of `this.self`, and to help o1js developers be careful not to disrupt the important side effect of calling `this.self` within `wrapMethod()`, we recommend documenting this in both the API documentation and in `wrapMethod()`.

9. `src/lib/provable/foreign-field.ts`:

   a) The `ForeignField.div()` function will assert that the `divisor` is canonical if the foreign field modulus is less than 2^{176}. This is due to the call to `Gadgets.ForeignField.divide()`, which asserts that the divisor is non-zero using `Gadgets.ForeignField.equals()`. This fact is currently undocumented, and may be surprising to users since the type of `divisor` is `AlmostReduced`.
   This also affects functions which use `equals()`, including `Point.equals()` in `elliptic-curve.ts`.
   b) Similarly, `AlmostReduced.equals()` asserts that `this` is canonical if the foreign field modulus is small enough. This fact should be documented.
   c) `ForeignField.assertEquals()` has several important undocumented features:
      i. Behavior for provable elements is different than constants. For constants, the function asserts equality of *equivalence classes*, i.e. equality modulo the foreign field modulus. For provable elements, the function asserts equality of *representatives*, e.g. $0 \neq 17 \bmod 17$.

   This difference should be clearly documented. We also strongly recommend changing the function name (e.g. to `assertRepresentativeEquality()`) to distinguish the check from the "natural" equality comparison within the foreign field (i.e. the equality implemented in `Canonical.equals()`).

10. `src/lib/provable/group.ts`

    a) `Group.constructor()`: The constant version of this function checks the `x`, `y` pair satisfies the curve equation. However, the variable implementation does not. This means the user must ensure `Group.check()` has been called on the value, or that it is otherwise constrained to satisfy the class invariant.

11. `src/lib/provable/merkle-tree.ts`

    a) `maybeSwap()`: Consider adding documentation clarifying that `maybeSwap()` swaps if the provided flag is `false`, rather than `true`. Additionally, consider renaming the

function name to `maybeNotSwap()`.

---

12. `src/lib/provable/provable.ts`:

    a) `switch()`: This function returns 0 if all provided mask values are 0. According to the test suite, this is intended behavior. However, it is not documented. We recommend clarifying the following comment: "It takes a "mask", which is an array of `Bool`s that contains only one `true` element".

---

13. `src/lib/provable/core/exists.ts`:

    a) `exists()` automatically reduces the results of the `compute()` function modulo the base field. This behavior is not documented.

---

14. `src/lib/provable/crypto/poseidon.ts`:

    a) `Poseidon.hashToGroup()`: This function returns a single value `x` along with the two `y`-coordinates corresponding to points on the curve at that `x` coordinate. While the prover always generates the first `y`-value to be even, it is not *constrained* to be even. A caller who does not do so may inadvertently use a prover-controlled hash function. The only caller of this function is in `nullifier.ts`, and does constrain the first `y`-coordinate to be even. Consider adding moving this constraint inside the `hashToGroup()` function, or adding a strongly-worded warning to the documentation of this user-facing method.

---

15. `src/lib/provable/gadgets/arithmetic.ts`:

    a) The documentation on `Gadgets.addMod32()` claims that it supports two inputs in the range $[0, 2^{64})$. However, its constraints imply that the sum of its two inputs can be at most $2^{33} - 1$. So, it only supports (all pairs of) inputs in the range $[0, 2^{32})$. To see this, note that `addMod32()` proves the existence of `q` and `r` satisfying

    ```
    q \in {0,1}
    r < 2**32
    x+y == q * 2^32 + r
    ```

    Since `x+y` cannot overflow, this shows that `x+y <= 2^32 + (2^32-1) = 2^33 - 1`.

---

16. `src/lib/provable/gadgets/bitwise.ts`

    a) The documentation of `not()` (found in `src/lib/provable/gadgets/gadgets.ts`) should be changed to indicate that:

       i. If `checked`, this operation fails when `a >= 2**paddedLength`, not when `a >= 2**254`.
       ii. If `!checked`, this operation **does not fail** (but is incorrect) when `a >= 2**254`.
       iii. The behaviors of `checked` vs `unchecked` are different when `a >= 2**length`:
          - `checked`: Fails if `a >= 2**paddedLength`, otherwise flips the least `length` bits of `a`
          - `!checked`: Meaningless value if `a >= 2**length`, otherwise `~a`.

17. `src/lib/provable/gadgets/elliptic-curve.ts`

    a) The following functions differ from the constant implementation, but this difference is not documented:

       i. `add()`: The constant implementation supports `p1 == p2` so long as `a != 0`. The variable case does not support `p1 == p2`.

       ii. `double()`: The constant implementation inlines `a = 0`, while the variable case supports `a != 0`.

       iii. `multiScalarMul()`: Due to the use of `double()`, the constant implementation requires `a = 0`, while the variable case supports `a != 0`.

    b) `multiScalarMul()` uses `sliceField3()` on each scalar. Once fixed (see V-O1J-VUL-044), this will require the scalars to be within a factor of 2 of canonical. This is a slightly stronger condition than almost-reduced, and should be documented (see V-O1J-VUL-033 for the full definition).

    c) `signEcdsa()`: This function does not account for some sophisticated cryptographic attacks such as timing side channels or lazily collected memory. Consider either removing this function from the external API, or indicating to users that its use outside of a secure computing environment comes with severe risks.

---

18. `src/lib/provable/gadgets/foreign-ecdsa.ts`

    a) `check()`: Careless users may expect that `EcdsaSignature.check()` ensures the signature is valid. Without a public key, of course, this cannot be the case. However, as users become accustomed to relying on the guarantees provided by o1js types, a neglectful user may assume the `check()` function ensures signature validity. Consider adding thorough documentation to the type emphasizing that the `EcdsaSignature` may be invalid, even once `check()`ed.

---

19. `src/lib/provable/gadgets/foreign-curve.ts`

    a) `ForeignCurve`: In the variable case, any `x`, `y` point may be supplied. This is in contrast to other constructors in o1js which attempt to ensure that a type, when constructed, is constrained to satisfy its `check()` function. The same holds for `ForeignCurve.from()`. This fact should be documented extremely clearly for the user.

---

20. `src/lib/provable/gadgets/foreign-field.ts`

    a) The `Sum` class is used internally to optimize away unnecessary range checks. It stores a lazily-computed sum, only performing the addition once the `finish()` function is called. The `Sum` class is only intended for use by the `ForeignField.assertMul()` function. Otherwise, `Sum.rangeCheck()` must also be called on the result. Consider adding this documentation to the `finish()` function to avoid internal misuse.

---

21. `src/lib/provable/gadgets/gadgets.ts`

    a) `assertAlmostReduced()`: The `skipMrc` function is undocumented.

    b) `addMod32()`: The input is assumed to be in the range `[0, 2^64)`, not `[0, 2^64]`.

---

22. `src/lib/provable/gadgets/sha256.ts`

    a) `padding()`: Consider renaming the variable `paddingBits` to `paddingBytes`, since it is an array of `UInt8`s. Note also that explicit chunking into 8-bit blocks may have higher performance guarantees than a regex match.
    b) `padding()`: The `padding()` function implements two of the three preprocessing steps defined in Sec 5 of the NIST Secure Hash Standard: "padding" and "parsing." Consider renaming this function `pad_and_parse()`.

---

23. `src/lib/provable/int.ts`

    a) We recommend documenting a concrete example where the use of `Unsafe` (e.g. `UInt64.Unsafe`) may unexpectedly lead to an incorrect result when it would not be an issue if the safe counterpart is used instead. As an example, one may expect that by transitivity, `a.assertLessThan(b)` would be enough to say that `a` is a `UInt64` if `b` is also a `UInt64`. However, this is not the case, and may be a very dangerous anti-pattern for users.
    b) The function `Int64.from()` will throw an error if provided a non-constant `Field`. This is because `Int64.fromFieldUnchecked()` is called when the provided value is a `Field`. `Int64.fromFieldUnchecked()` requires the value to be constant, and checks that it is in range. However, this requirement is implicit via a call to `toBigInt()`. Consider adding a more informative error message to `Int64.from()` when a non-constant `Field` is provided. Also, consider renaming `fromFieldUnchecked()` to `fromUncheckedConstantField()`.

Impact Minor errors may lead to future developer confusion. Particularly for `src/bindings/crypto/elliptic-curve.ts`, we have added more detailed comments below:

1. Note that some of the functions (such as `projectiveAdd()` and `projectiveDouble()`) are exported by the module for use outside of the context of a curve created by `createCurveProjective()`.
2. Newly implemented methods which ensure the output is on the curve, but do not ensure `y^2=x^3` when `z = 0`, will produce valid curve points for which `projectiveOnCurve()` returns false.
3. Users may think they have checked an element is on the subgroup, but not have performed a group check.

Recommendation Update the documentation and/or function names to make these assumptions clear.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.58 V-O1J-VUL-058: Missing checks on constants

Severity	Info	**Commit**	8dde2c3, e7ded4b
Type	Data Validation	**Status**	Acknowledged
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

- `src/bindings/lib/bigint-helpers.ts`: the file defines several helper functions to manipulate large integers. This includes conversions between various representations, including different base representations, byte arrays, and bigints, as well as several other useful mathematical operations.
  Several of these functions make unchecked assumptions about the provided input.
  - `bytesToBigint32()`: Unchecked conversion from a little-endian array of bytes to a 32-byte `BigInt`. This assumes that the represented value is less than `2**256`.
  - `bigintToBytes32()`: Unchecked conversion from a `BigInt` to a 32-byte little-endian representation. This assumes the converted value is less than `2**256`.
  - `log2()`: The `log2()` implementation (shown below) assumes that `n` is an integer in the domain of the logarithm, i.e. that `n >= 1`.

```
/**
 * ceil(log2(n))
 * = smallest k such that n <= 2^k
 */
function log2(n: number | bigint) {
  if (typeof n === 'number') n = BigInt(n);
  if (n === 1n) return 0;
  return (n - 1n).toString(2).length;
}
```

Snippet 4.78: Definition of `log2()`.

  - `fromBase()`/`toBase()`: These functions check that `base > 0`, but should instead check that `base > 1`.
  - `changeBase()`/`fromBase()`: each digit in `digits` should be checked to be between `0` and `base - 1` (inclusive).
  - `bytesToBigInt()`: each byte should be between 0 and 255 (inclusive).

- `src/bindings/lib/crypto/elliptic-curve.ts`
  - `createCurveProjective()`, `createCurveAffine()`: These functions do not validate:
    - `p > 1`.
    - the supplied `generator` is on the curve and in the subgroup.
    - `(endoBase * generator.x, generator.y) == endoScalar * (generator.x, generator.y)`.

- `src/bindings/lib/crypto/elliptic-curve-endomorphism.ts`:

- `egcdStopEarly()`: `l` should be checked to be in the range of `1` to `p - 1` (inclusive). The current check is not strict enough: e.g. when `l = p`, a division by zero error will occur, with an unhelpful error message. The documentation should also be updated to reflect this, along with the fact that `p` must be prime, not just an arbitrary positive integer.

▶ `src/bindings/lib/crypto/finite-field.ts`

- `createField()`: This function does not check that `p > 1`.
- `fromNumber()`: `fromNumber()` allows conversion from a `number` to a `bigint`. This value should be checked against MAX_SAFE_INTEGER to ensure that users do not make the common error of supplying a large constant as a `number` instead of a `bigint`.

▶ `src/lib/provable/merkle-map.ts`

- `MerkleMapWitness.constructor()`: This method does not check `isLefts` and `siblings` are the same length, or of length `bits`.

▶ `src/lib/provable/merkle-tree.ts`

- `MerkleTree.getNode()`: This method may silently return an entry from `zeroes` when an out-of-bounds access is made on a level. See also the typing recommendation made in V-O1J-VUL-059. For example,

```
let height = 2n;
let tree = new MerkleTree(Number(height));
let node = tree.getNode(Number(height - 1n), 2n**height)
console.log(node);
```

  outputs
  Field { value: [0, [0, 21565680844461314807l...]] }
  `height-1` is the root, so only index 0 should be valid.
- `MerkleWitness()`: This function does not check that `height` is a positive number.

▶ `src/lib/provable/provable.ts`

- `assertEqualExplicit()`, `equalExplicit()`: Consider checking that `xs.toFields(x)` is the same length as `ys.toFields(y)`, since `x` or `y` inheriting from the type parameter `T` may not mean they are the same type. See, for example, the below code snippet.

```
function func<T>(x: T, y: T) {
    console.log(x, y);
}

type X = {a: number};
let x: X = {a: 1};
let y = {a: 2, b: 3};

```



```
9 func<X>(x,y);
```

Snippet 4.79: This code snippet type-checks since `y` extends `X`, even though `x` and `y` do not (in a strict sense) have the same type.

▶ `src/lib/provable/scalar.ts`

- `ScalarConst.is()`: This function checks that `x` is an array whose first entry is `0` and second entry is a `bigint`. However, it fails to check that the array is of length 2.
- `constFromBigint()`: This function converts a `bigint` `x` to a `ScalarConst` by returning `[0, x]`. However, it does not reduce `x` to its canonical representation in `Fq`, as performed elsewhere in the codebase when converting a `bigint` to a field.

▶ `src/lib/provable/gadgets/bitwise.ts`

- `leftShift32()`: The documentation in `gadgets.ts` indicates that `bits` should be in the range `[0, 32]`.

▶ `src/lib/provable/gadgets/bit-slices.ts`

- `bytesToWord()`: This internal function converts a little-endian array of bytes (represented as `UInt8`s) to its associated value within the base field. To avoid overflow errors down the road, consider adding a length-check on the array.

For example, within the 254-bit Pallas field, overflows may be possible once the array length reaches `32`.

▶ `src/lib/provable/gadgets/comparison.ts`

- `compareCompatible()`: This function checks if `x <= y` and `x < y` under the assumption that `x, y < 2**n`. This is done by checking the `n`th bit of `2**n + y - x` for the `<=` check, then seeing if any of the remaining `n` bits are non-zero for the equality check.

However, this second check is vacuous for the `n = 0` case. Consider handling the `n = 0` case separately, or asserting that `n > 0`.

▶ `src/lib/provable/bool.ts`

- `Bool.fromFields()`: Any `Field` is accepted without performing a range-check on the value to ensure it is 0 or 1.
- `BoolBinable.readBytes()`: `bytes[offset]` should be ensured that it is either 0 or 1.

▶ `src/lib/provable/field.ts`

- `Field.fromFields()`: An array of arbitrary length is accepted, when only arrays of length 1 are valid. Adding a runtime assertion could help prevent erroneous behavior.

▶ `src/lib/provable/merkle-tree.ts`

- `MerkleTree`'s `constructor()` and `MerkleWitness()`: check that `height > 0`
- `MerkleTree`'s `setLeaf()` and `getWitness()`: check that `index >= 0`
- `MerkleTree`'s `fill()`: as the TODO suggests, `leaves`'s length should be checked.
- `MerkleTree`'s `setNode()`: as a sanity check, the TODO suggestion to check `level` and `index` could make sense.

▶ `src/lib/provable/types/circuit-value.ts`

- `CircuitValue.fromFields()`: `fromFields()` checks that the provided fields (`xs`) is *at least* the expected length, rather than *exactly* the expected length. This may lead to accidentally ignored fields.
- `prop()`: we recommend that if `toFields` and `fromFields` are not defined, then an error should be raised, since this could cause constraints to be accidentally omitted in an unexpected way (i.e. when the field type is a disjunction of two circuit classes), similar to another issue V-O1J-VUL-014. While logging a warning may help, it would be easy to miss compared to a hard error.

▶ `src/lib/provable/types/fields.ts`

- `fields()` returns a provable type for a tuple of `Fields`. However, its `fromFields()` method does not check the length of the provided array. This may easily lead to dropped fields, or other user errors down the line.

▶ `src/lib/provable/crypto/poseidon.ts`

- `Poseidon.hash()` should consider checking that `input` is not empty to preventing squeezing before absorbing.

▶ `src/lib/mina/mina.ts`

- `Network()`: If `options.mina` is an array of length 0, an error should be produced. Instead no graphql endpoint is set, and the code continues executing.

▶ `src/lib/mina/precondition.ts`

- `preconditionClass()`: the expression `${baseKey}.${key}` may construct a colliding key when the `key` includes the `.` character. We recommend checking that `key` does not have a character `.` in it.

- `src/lib/mina/state.ts`
  - `getLayout()`: This function gets the layout of state fields within a smart contract class. Adding a runtime check that `offset <= MAX_LENGTH_OF_APP_STATE` may improve the error messages provided to the user.

Impact Misuse of these functions will lead to confusing errors, rather than immediately alerting the user to their error.

Recommendation Add the appropriate checks.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.59 V-O1J-VUL-059: TypeScript recommendations/best practices

Severity	Info	**Commit**	8dde2c3
Type	Maintainability	**Status**	Acknowledged
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

In the following locations, the auditors identified opportunities to leverage the TypeScript typing system for improved code safety:

- `src/bindings/lib/provable-generic.ts`
  - We strongly recommend to avoid dropping type constraints (e.g. via `any` or unsafe reflections). For example, the following code type checks and outputs the variable `bad` as an array of objects, despite the fact that TypeScript infers `bad` to be an array of strings.

```
let { provable } = createDerivers<string>();
let WrappedBigInt = {
    toFields: (wrapped: bigint) => {
        console.log("this should be bigint: ", wrapped);
        return [wrapped];
    }
};
let ProvableW = provable(WrappedBigInt);
let bad : string[] = ProvableW.toFields({ toFields: 42 });
console.log(bad);

/*
Output:

this should be bigint:  { toFields: 42 }
[ { toFields: 42 } ]
*/
```

- `src/lib/mina/account-update.ts`
  - `AccountUpdate.constructor()`: This method has only one overload defined, but it should have two. This will obviate the need to cast `AccountUpdate` to `any` inside of `clone()`.

- `src/lib/mina/precondition.ts`
  - `preconditionSubclass()`, `preconditionSubClassWithRange()`: These functions should declare their return type as a `PreconditionSubclassType` to ensure the implementations stay up to date with changes in the interface.
  - `isRangeCondition()`: This function acts as a type check for the `RangeCondition` type. However, it only checks for two of the three fields: `isSome` and `lower`. The function should also check for `upper`.

- `Account()`: This function uses the `...` syntax with two different functions, both of which define `delegate`. Because of this, `delegate` requires a special definition so that it contains fields for both setters and getters. Consider not using the `...` syntax here, and explicitly defining each field. This will ensure a compiler error warns developers if any future changes lead to possible overwrites.

▶ `src/lib/mina/events.ts`

- The value `Actions` should not have type `Events`. Instead, it should have its own type `Actions`. Later, in `account-update.ts`, the `actions` field could then have type `Actions`. This disallows misuses of `Events` as `Actions` (or vice versa).

▶ `src/lib/mina/mina.ts`

- `Network()`: The check `typeof options == 'string'` implies that `options` is defined, making the `&&`'ed check on `options` unnecessary. Similar expressions appear throughout the function, such as when checking the `lightnetAccountManager`'s type.

▶ `src/lib/provable/gadgets/elliptic-curve.ts`:

- `multiScalarMul()`: This function returns a `Point`. However, no valid representation of the 0-point exists. When the result is 0, a `Point` containing an incorrect value (`{x:0,y:0}`) is returned. Instead, consider having `multiScalarMul()` return a union type. For example, `Point | ZeroPoint`, with `ZeroPoint` being the empty type.

▶ `src/lib/provable/int.ts`

- `class UInt64`, `UInt32`, `Sign`, `Int64`, `UInt8`: As described below for the `Field` class in `field.ts`, each `value` field should be made `readonly`.

▶ `src/lib/provable/bool.ts`

- `class Bool`: As described below for the `Field` class in `field.ts`, `Bool.value` should be made `readonly`.
- `type BoolVar = FieldVar`: While this appears to add type safety to distinguish `BoolVars` from `FieldVars`, this definition is a type alias. Since TypeScript is structurally typed, there is no difference between the `BoolVar` and `FieldVar` types. For example, the `constructor()` of the `Bool` class accepts a `BoolVar`, so will also accept any `FieldVar`. Consider either using a newtype-style pattern such as described here, removing the type, or documenting the potential issue.

▶ `src/lib/provable/field.ts`

- `class Field`: The `value` member of the `Field` class is intended to be readonly. Throughout the implementation, the `Field` class relies on the invariant that `value` is not set, and (in the constant case) stores the canonical form of the underlying `Field`. To properly enforce this invariant, `value` should be marked as `readonly`.

▶ `src/lib/provable/provable.ts`

- `assertEqual()`, `equal()`, `_if()`: Each of these methods is overridden to accept two possible signatures: `(type, x, y)` (explicit) or `(x, y)` (implicit). For example, the below code snippet shows how `equal()` detects which case is provided.

```
function equal(typeOrX: any, xOrY: any, yOrUndefined?: any) {
  if (yOrUndefined === undefined) {
    return equalImplicit(typeOrX, xOrY);
  } else {
    return equalExplicit(typeOrX, xOrY, yOrUndefined);
  }
}
```

Snippet 4.80: Definition of `equal()`

Note, however, that `provable(undefined)` returns a provable type for the `undefined` type. When `equal(provable(undefined), undefined, undefined)` is passed to the function, the explicit branch is taken instead of the implicit one. Consider explicitly disallowing this case.

▶ `src/lib/provable/crypto/foreign-ecdsa.ts`

- `createEcdsa()`: Consider using `typeof` to determine which case of `CurveParams | typeof ForeignCurve` is passed, rather than checking if `'b' in curve`, which may not be robust to user extensions of the library.

▶ `src/lib/provable/crypto/keccak.ts`

- Consider using the type `Field[typeof KECCAK_DIM][typeof KECCAK_DIM]` instead of `Field[][]`, since the array dimensions are known statically.

▶ `src/lib/provable/gadgets/sha256.ts`

- `padding()`: The padding function returns an array of words, each word corresponding to 16 32-byte chunks. However, its return type is `UInt32[][]`. Consider changing the return type to `UInt32[][16]`.

▶ `src/lib/provable/types/auxiliary.ts`

- The `RandomId` type uses non-cryptographic randomness (`Math.random()`). Switching to cryptographic randomness might help ensure users do not rely on these values to provide high levels of entropy.

- `src/lib/provable/types/struct.ts`
  - `isPrimitive()`: This function checks if an `obj` is primitive. However, it performs this check by testing if `obj` is an `instanceof` one of the primitive types. This will not work correctly if someone extends one of the primitive types, e.g. makes a subclass of the `Field` class. Fortunately, this method is only used internally. However, as structures become more advanced with future releases, this could potentially lead to issues down the road.

- `src/lib/provable/types/unconstrained.ts`
  - constructor: instead of consuming two separate arguments and then constructing an object inside the constructor, which then necessitates `as any`. the constructor could instead consume the whole object. It's then the callers' job to construct objects, but this is easy to do.

- `src/lib/provable/core/fieldvar.ts`, `o1js/src/lib/provable/types/struct.ts` (in the doc comments): Tuple type should have `readonly`. See also: V-O1J-VUL-019.

- `src/lib/provable/scalar.ts`
  - Most methods are not provable. It might be worth considering a different organization to provide type safety

---

- `src/lib/provable/merkle-tree.ts`
  - the `nodes` field in `MerkleTree` should use an array of `Map` instead — (1) array is more appropriate since levels are consecutive, and (2) `Map` is more appropriate since the values are homogeneous, but dynamically sized. This will also likely have better performance, and the `Map` API might make it easier to work it the data structure.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.60 V-O1J-VUL-060: Inverse assertion allows for common anti-pattern

Severity	Info	**Commit**	8dde2c3
Type	Usability Issue	**Status**	Acknowledged
File(s)	`src/lib/provable/core/fieldvar.ts`		
Location(s)	inv(), div(), sqrt()		
Confirmed Fix At	N/A		

The `Field.inv()` and `Field.div()` functions assert existence of an inverse. `Field.sqrt()` asserts the existence of a square root. The below code snippet shows example of these assertions.

```
// constrain x * z === 1
assertMul(this, z, FieldVar[1]);
```

Snippet 4.81: Assertion in `Field.inv()` that the inverse of `this` value exists.

This is certainly the correct behavior for the common case. However, it can lead unsuspecting users to write over-constrained circuits.

For example, one common anti-pattern seen in other ZK languages is attempting to use a ternary operator to provide a "default" value when a divisor may be zero.

```
const divisorIsZero = divisor.equals(0n);
let quotient = Provable.if(
    divisorIsZero,
    0n,
    dividend.div(divisor)
)
```

This will lead to an over-constrained circuit when `divisor === 0n`, since `dividend.div(divisor)` *always* asserts that `divisor` is non-zero.

Impact Users may deploy contracts which always error when certain values are `0`.

Recommendation Warn users away from this pattern in the documentation of `inv()` and `div()`. Further, consider providing a `.invOrDefault()` and `.divOrDefault()` which users can use to conditionally compute an inverse or division.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.61 V-O1J-VUL-061: Assignment instead of copy

Severity	Info	**Commit**	8dde2c3
Type	Maintainability	**Status**	Acknowledged
File(s)	`src/lib/provable/types/unconstrained.ts`		
Location(s)	Unconstrained.setTo()		
Confirmed Fix At	N/A		

`Unconstrained` holds an (optional) unconstrained value. As can be seen in the below snippet, the copy operator assigns `this.option` to a reference of the other `value.option`.

```
setTo(value: Unconstrained<T>) {
   this.option = value.option;
}
```

Snippet 4.82: Snippet from `Unconstrained.setTo()`

Impact Fortunately, other mutation functions (like `set()`) set `this.option` directly, rather than setting `this.option.isSome` or `this.option.value`. However, if methods are added in the future which set the sub-fields of `this.option`, they will also affect the sub-fields of `value`, since assignment is by reference.

Recommendation Create a fresh object, e.g. `this.option = { ...value.option}`.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.62 V-O1J-VUL-062: Duplicate, unused, or outdated code

Severity	Info	**Commit**	e7ded4b
Type	Maintainability	**Status**	Acknowledged
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

In the following locations, code is either unused or duplicated.

- `src/bindings/lib/generic.ts`
  - The return type of `GenericSignable.toInput<T, TJson, Field>()` is the same as `GenericHashInput<Field>`, defined below.

- `src/bindings/mina-transaction/derived-leaves.ts`
  - The hard-coded value of 48 is used, instead of `8 * tokenSymbolLength`. An error message also reference `6` instead of the `const` value `tokenSymbolLength`, which is defined to be `6`.

- `src/lib/mina/events.ts`
  - This file duplicates the functions implemented by `createHashHelpers()` in `src/lib/provable/crypto/hash-generic.ts`.
  - `Events` and `Actions` have the same implementation up to the hash prefixes used. Consider combining the implementations entirely, parameterized by just the hash prefixes.

- `src/lib/mina/account-update.ts`
  - `balance()`: `new Int64(magnitude, sgn)` could be switched to `Int64.fromObject`.
  - `AccountId`: `parentTokenId` should use type `Types.TokenId`.

- `src/lib/proof-system/circuit.ts`
  - `public_()`: This function appears to be unused (in other modules, and within this module).

- `src/lib/provable/bool.ts`
  - `BoolVar.constructor()` uses `FieldVar` instead of the aliased `BoolVar`.

- `src/lib/provable/field.ts`
  - `Field.isOdd()` could rely on `Fp.isEven()` when computing the constant case.

- `FieldBinable.readBytes()` could set a constant value to `Math.ceil(Fp.sizeInBits/8)`, instead of hard-coding the usage of `32`.

▶ `src/lib/provable/core/fieldvar.ts`

- Under `scale()`, `typeof c === 'bigint' ? FieldConst.fromBigint(c) : c;` could have been replaced with existing abstraction: `FieldVar.constant(c)`
- Under `add()` and `scale()`, `x[1][1]` and friends could have been replaced with `constToBigint(x[1])`.
- The types `ConstantFieldVar` and `VarFieldVar` could be used in the disjunctive type definition of `FieldVar`.

▶ `src/lib/provable/gadgets/basic.ts`

- The `assertMul()` function is unused.
- The `type Constant = [FieldType.Constant, FieldConst];` is a duplicate of the type `ConstantFieldVar` defined in `fieldvar.ts`.
- `isConst()` is defined directly on the `FieldVar` type (as `isConstant()`).
- Consider renaming the `isVar()` function to `isScaledVar()` to avoid confusion with the function `FieldVar.isVar()`, which has different semantics.

▶ `src/lib/provable/gadgets/bitwise.ts`

- `rightShift64()`, `leftShift64()`, and `rotate64()` use the same code snippet to check that `bits` is in the range `[0, 64]`.

▶ `src/lib/provable/core/field-constructor.ts`

- `isField` and `isBool` appear to be unused. The use of these functions is also an anti-pattern, since it means the type information is lost somewhere.

▶ `src/lib/provable/gadgets/basic.ts`

- `toLinearCombination()`: `sx === 0n` rather than the more robust `Fp.equals()`.

▶ `src/lib/provable/gadgets/elliptic-curve.ts`

- `decomposeNoRangeCheck()`: `s0Negative.assertBool()` returns `s0Negative` represented as a `Bool`. This would avoid the additional `Unsafe` usage, and allow returning the value as a `Bool` rather than a field.

  This would also remove the need for an `Unsafe.from` in `negateIf()`.

- `scale()`: the default value for `config` is `mode` setting to `'assert-nonzero'`, which is redundant. Users can provide `{}` as `config`, which will also result in the `'assert-nonzero'` mode due to the default in the `multiScalarMul()` function.

▶ `src/lib/provable/gadgets/foreign-field.ts`

- `Sum.finish()`: This function has a parameter `isChained` which is set to `false` by default. It is never set to `true`. The `true`-branch is untested, and should either be included into tests, or removed.
- The upper-bound check on `f` (`f < 1n << 259n`) is repeated explicitly in `multiply()`, `inverse()`, and `divide()`.

▶ `src/lib/provable/gadgets/range-check.ts`

- The functions `rangeCheck32()`, `rangeCheck16()`, and `rangeCheckN()` share a large amount of logic. Consider abstracting this away into a separate function.
- `rangeCheckN()` and `rangeCheckHelper()` both contain the same checks on their `n` parameter.

▶ `src/lib/provable/foreign-field.ts`

- `modulus()` inlines the definition of `this.Constructor`, rather than using the abstraction.

▶ `src/lib/provable/group.ts`

- The constructor can use `check()` to check that the constant value is on the curve.
- Although the uses of `===` on bigints are safe, bigint equality can use `Fp.equal`, which would always guarantee safety.

▶ `src/lib/provable/int.ts`

- The following locations compute the maximum value of an integer type rather than using the static `MAXINT` function:
  - `UInt64.checkConstant()`
  - `UInt64.toUInt32Clamped()`
  - `UInt64.lessThanOrEqual()`
  - `UInt64.lessThan()`
  - `UInt32.checkConstant()`
  - `UInt32.lessThanOrEqual()`
- More generally, `UInt8`, `UInt32`, and `UInt64` share large portions of their logic. Consider unifying their shared logic into a common base class.
- Various calls to `rangeCheckN` could be replaced by a more specialized (and presumably more efficient) variant. E.g., `rangeCheck64`.
  Some of the comparison functions show signs of possible copy-and-paste errors. For example, `UInt32.lessThan()` uses `lessThanGeneric()` with `c = 1n << 64n`, rather than `c = 1n << 32n`. Switching to the 32-bit implementation may be more efficient.

- These classes should switch from using the deprecated `CircuitValue` to `Struct`:
  - `src/lib/provable/int.ts`: `UInt64`, `UInt32`, `Sign`, `Int64`
  - `src/lib/provable/merkel-map.ts`: `MerkelMapWitness`
  - `src/lib/provable/merkel-tree.ts`: `BaseMerkelWitness`
  - `src/lib/provable/string.ts`: `Character`, `CircuitString`
  - `src/lib/provable/crypto/signature.ts`: `PrivateKey`, `PublicKey`, `Signature`

- `src/lib/provable/merkle-map.ts`
  - `computeRootAndKey()`: instead of using two `Provable.if`, `maybeSwap()` from `merkle-tree.ts` could be used instead. This is also an optimization to reduce constraints.
  - `_keyToIndex()`: instead of composing bits to a `bigint` manually, `fromBase` from `bigint-helpers.ts` under `o1js-bindings` could be used instead. The performance would also be improved (though minimally since the key has a fixed length) due to Estrin's scheme.

- `src/lib/provable/types/struct.ts`
  - `Struct()`: The return-type of `Struct` returns a complicated type, which includes an intersection type with an interface defining the `toInput()`, `toJSON()`, `fromJSON()`, and `empty()` methods. This re-defines the `GenericSignable` interface from `src/bindings/lib/generic.ts`. The only difference is that `undefined` is allowed as a result from `toInput()`. Consider re-using types where possible, and making a concrete "`Struct`" type for the returned value.

Impact Unused or duplicated code may become out of date, and lead to issues when used down the line.

Recommendation Remove the unused program constructs, and use available abstractions to avoid code duplication.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

4.1.63 V-O1J-VUL-063: Potentially incorrect rounding

Severity	Info	**Commit**	e7ded4b
Type	Logic Error	**Status**	Acknowledged
File(s)	`o1js-bindings/crypto/elliptic-curve-endomorphism.ts`		
Location(s)	divideAndRound()		
Confirmed Fix At			

`divideAndRound(a, b)` returns a rounding of `a / b` to a nearest integer.

```
// round(x / y)
function divideAndRound(x: bigint, y: bigint) {
  let signz = sign(x) * sign(y);
  x = abs(x);
  y = abs(y);
  let z = x / y;
  // z is rounded down. round up if it brings z*y closer to x
  // (z+1)*y - x <= x - z*y
  if (2n * (x - z * y) >= y) z++;
  return signz * z;
}
```

Snippet 4.83: The implementation of `divideAndRound`

However, this specification is in fact underspecified, because the "nearest integer" is not well-defined for the midway point (i.e. `.5`), as it could be either rounded up or down. In particular, the rounding behavior of `divideAndRound` disagrees with `Math.round` from JavaScript's standard library when the division result is negative, which could be surprising to users.

```
> Math.round(-0.5)
-0
> divideAndRound(-1n, 2n)
-1n
```

Impact Both rounding behaviors in fact conform to the properties presented in *Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms*: given β (the division result) and b (the rounded result), the paper only concerns that $|\beta - b| \leq \frac{1}{2}$. Still, the computed result could diverge from other implementations (that use a different rounding method), which could be unexpected to users.

Recommendation Document the rounding behavior clearly. Whether or not the actual rounding behavior should be adjusted to conform with `Math.round` is up to developers' discretion.

Developer Response The developers have been notified of the issue but have yet to respond with fixes.

5 Formal Verification

5.1 Methodology

Our team verified several components in the codebase. This consists of functional correctness verification, to see whether the code deviates from the intended behavior, and deterministic verification, to see whether the circuits are under-constrained. See Table 5.1 for a complete list.

For functional correctness, we formally verified the code using a forked version of Rosette, with a forked cvc5 as its underlying SMT solver.

- Rosette [1] is an open-source solver-aided programming system, which can be used to verify that a program satisfies a specification by compiling the verification queries to SMT constraints. The constraints are then submitted to an underlying SMT solver to obtain a verification result. We forked Rosette to support the finite field theory.
- cvc5 [2] is an open-source SMT solver that supports the finite field theory. We forked cvc5 to support the range reasoning for finite field.

For deterministic verification, we formally verified the circuits using an experimental version of Veridise's tool Picus, with the aforementioned cvc5 as its underlying SMT solver.

- Picus [3] is an open-source tool to prove that zero-knowledge circuits are deterministic (that is, not under-constrained) by formally verifying they have exactly one solution or find a counterexample if the circuits are under-constrained. It works by employing our novel algorithm that interacts static analysis with SMT solvers.

Veridise auditors extracted and modeled many components of o1js code in Rosette and Picus, and employed the tools to verify them. The effort focused on small to medium-sized circuits whose correctness reasoning is amenable to SMT solving. In particular, the `Field`, `Ints`, `Bool` classes are most of our targets, with a few targets in the `crypto` component. Certain other potential targets, such as the `ForeignField` class, were not chosen due to their interaction between finite field reasoning and integer reasoning, which is not currently supported by SMT solvers.

5.2 Properties Verified

A complete list of the properties verified is shown in Table 5.1. Each row displays a natural language description of the property verified, and its current status:

verified the code or circuit is verified to be functionally correct or deterministic

partially verified the code or circuit is verified to be functionally correct or deterministic for a subset of possible inputs. The tooling limitation may prevent us from verifying the code on larger inputs.

counterexample the code or circuit violates the correctness property or it is under-constrained, with a counterexample found.

benign counterexample the circuit is under-constrained. However, this non-deterministic behavior is known and expected.

The formal verification effort identified a total of 3 bugs out of 20 verified properties.

Table 5.1: Properties Verified.

Specification	Property	Status
V-O1J-PROP-001	Correctness of Bool.or	Verified
V-O1J-PROP-002	Correctness of Int64.div	Counterexample
V-O1J-PROP-003	Correctness of Int64.isPositive	Counterexample
V-O1J-PROP-004	Correctness of divideAndRound	Verified
V-O1J-PROP-005	Correctness of maybeSwap	Verified
V-O1J-PROP-006	Determinism of Bool.equals	Verified
V-O1J-PROP-007	Determinism of Field.equals	Verified
V-O1J-PROP-008	Determinism of Field.inv	Verified
V-O1J-PROP-009	Determinism of Field.isOdd	Verified
V-O1J-PROP-010	Determinism of Field.sqrt	Benign Counterexample
V-O1J-PROP-011	Determinism of Poseidon.hashToGroup	Benign Counterexample
V-O1J-PROP-012	Determinism of UInt64.divMod (and UInt32.divMod...	Verified
V-O1J-PROP-013	Determinism of addMod32	Verified
V-O1J-PROP-014	Determinism of arrayGet	Partially Verified
V-O1J-PROP-015	Determinism of divMod32	Counterexample
V-O1J-PROP-016	Determinism of isZero	Verified
V-O1J-PROP-017	Determinism of lessThanOrEqualGeneric (and less...	Verified
V-O1J-PROP-018	Equivalence of Ch	Verified
V-O1J-PROP-019	Equivalence of Maj	Verified
V-O1J-PROP-020	Range analysis of sigma	Verified

5.3 Detailed Description of Verified Properties

5.3.1 V-O1J-PROP-001: Correctness of Bool.or

Verification Status Verified

Natural Language We verified that the `Bool.or` method in `o1js/src/lib/provable/bool.ts` is correct.

Formal We first assume that the inputs are within the correct range:

$$x = 0 \lor x = 1$$
$$y = 0 \lor y = 1$$

We want to verify under the assumptions that:

$$actual = 1 - (1 - x) \times (1 - y)$$
$$expected = \begin{cases} 1 & \text{if } x = 1 \lor y = 1 \\ 0 & \text{otherwise} \end{cases}$$

whether the following formula holds:

$$actual = expected$$

Results The verification via the tool Rosette succeeds.

```
#lang rosette

(require rosette/solver/smt/cvc4
         rosette/base/core/felt)

(current-solver (cvc4 #:path (find-executable-path "cvc5")))

(define F (finfield
    28948022309329048855892746252171976963363056481941560715954676764349967630337)
    )

(define-symbolic x y F)

(define zero (felt 0 F))
(define one (felt 1 F))

;; domain restriction
(assert (|| (@ffeq x zero) (@ffeq x one)))
```

```
17 (assert (|| (@ffeq y zero) (@ffeq y one)))
18
19 (define expected
20   (cond
21     [(|| (@ffeq x one) (@ffeq y one)) one]
22     [else zero]))
23
24 (define (ff.sub x y)
25   (@ff.add x (@ff.neg y)))
26
27 (verify (assert (@ffeq (ff.sub one (@ff.mul (ff.sub one x) (ff.sub one y)))
28                        expected)))
```

Snippet 5.1: The verification program

5.3.2 V-O1J-PROP-002: Correctness of Int64.div

Verification Status Counterexample

Natural Language We verified that the `Int64.div` method in `o1js/src/lib/provable/int.ts` is correct according to the documentation:

> `x.div(y)` returns the floor of `x / y`, that is, the greatest `z` such that `z * y <= x`.

Formal We first assume that the inputs are within the correct range:

$$\begin{aligned} x_{mag} &< 2^{64} \\ y_{mag} &< 2^{64} \\ x_{sgn} = -1 \lor x_{sgn} &= 1 \\ y_{sgn} = -1 \lor y_{sgn} &= 1 \end{aligned}$$

We further assume that this representation is "repaired" to remove the underconstrained-ness when $x_{mag} = 0 \land x_{sgn} = -1$, as reported in V-O1J-PROP-003 / V-O1J-VUL-015.

$$\begin{aligned} x_{mag} = 0 &\implies x_{sgn} \neq -1 \\ y_{mag} = 0 &\implies y_{sgn} \neq -1 \end{aligned}$$

We want to verify under the assumptions that:

$$\begin{aligned} z_{sgn} &= x_{sgn} \times y_{sgn} \\ z_{mag} &= x_{mag} / y_{mag} \\ (z'_{sgn}, z'_{mag}) &= \begin{cases} (1, 0) & \text{if } z_{sgn} = -1 \land z_{mag} = 0 \\ (z_{sgn}, z_{mag}) & \text{otherwise} \end{cases} \\ x &= x_{mag} \times x_{sgn} \\ y &= y_{mag} \times y_{sgn} \\ z &= z'_{mag} \times z'_{sgn} \end{aligned}$$

whether the following formula holds:

$$z \times y \preceq x$$

where:

- $/$ is described in V-O1J-PROP-012
- $a \preceq b$ is defined as $a \prec b \lor a = b$ where $a \prec b$ is defined in V-O1J-PROP-003

(For completeness, another property that should be verified is $r' > r \implies x \prec r' \times y$, but since $r \times y \preceq x$ already displays issues, it doesn't make much sense to continue the verification effort.)

Results The verification via the tool Rosette results in counterexamples. For example, the division of $(x_{sgn}, x_{mag}) = (-1, 18446744073709551614)$ and $(y_{sgn}, y_{mag}) = (1, 18446744073709551615)$ results in $(z_{sgn}, z_{mag}) = (1, 0)$. This reveals a potential issue regarding the rounding behavior for negative numbers: V-O1J-VUL-063.

```
#lang rosette

(require rosette/solver/smt/cvc4
         rosette/base/core/felt)

(current-solver (cvc4 #:path (find-executable-path "cvc5")
                      #:options (hash ':ff-solver 'int)))

(define P
   28948022309329048855892746252171976963363056481941560715954676764349967630337)

(define F (finfield P))
(define halfP (/ (add1 P) 2))

(define-symbolic magx magy sgnx sgny F)

(define zero (felt 0 F))
(define one (felt 1 F))

(define 2^64 (felt (expt 2 64) F))

;; domain restriction
(assert (|| (@ffeq sgnx one) (@ffeq sgnx (@ff.neg one))))
(assert (|| (@ffeq sgny one) (@ffeq sgny (@ff.neg one))))
(assert (@ff.lt magx 2^64))
(assert (@ff.lt magy 2^64))

;; further assumption
(assert (=> (@ffeq magx zero) (@ffeq sgnx one)))
(assert (=> (@ffeq magy zero) (@ffeq sgny one)))

(define sgnout (@ff.mul sgnx sgny))

(define (divMod x y)
  (define-symbolic* q F)
  (define r (@ff.add x (@ff.neg (@ff.mul q y))))
  (assert (@ff.lt q 2^64))
  (assert (@ff.lt r 2^64))
  (assert (@ff.lt r y))
  (values q r))

(define-values (magout _) (divMod magx magy))

```

```
(match-define (list magout* sgnout*)
  (cond
    [(&& (@ffeq magout zero) (@ffeq sgnout (@ff.neg one)))
     (list zero one)]
    [else
     (list magout sgnout)]))

(define out (@ff.mul magout* sgnout*))
(define y (@ff.mul magy sgny))
(define x (@ff.mul magx sgnx))

;; custom < that "knows" how to compare "negative numbers"
(define (mylt x y)
  (cond
    [(&& (@ff.lt x (felt halfP F))
         (@ff.lt y (felt halfP F)))
     (@ff.lt x y)]
    [(@ff.lt x (felt halfP F))
     ;; y is negative
     #f]
    [(@ff.lt y (felt halfP F))
     ;; x is negative
     #t]
    [else (@ff.lt x y)]))

(define (myle x y)
  (|| (mylt x y) (@ffeq x y)))

;; two properties:
;; - out * y <= x
;; - if out_hyp > out, then out * y > x
;;
;; Here, we verify only the first one.
(define sol (verify (assert (myle (@ff.mul out y) x))))
(printf "out: ~a\n" (evaluate (list magout* sgnout*) sol))
(printf "x: ~a\n" (evaluate (list magx sgnx) sol))
(printf "y: ~a\n" (evaluate (list magy sgny) sol))
```

Snippet 5.2: The verification program

5.3.3 V-O1J-PROP-003: Correctness of Int64.isPositive

Verification Status Counterexample

Natural Language We verified that the `Int64.isPositive` method in `o1js/src/lib/provable/int.ts` is correct.

Formal We first assume that the inputs are within the correct range:

$$x_{mag} < 2^{64}$$
$$x_{sgn} = -1 \vee x_{sgn} = 1$$

We want to verify under the assumption that:

$$x = x_{sgn} \times x_{mag}$$

whether the following formula holds:

$$0 \prec x \iff x_{sgn} = 1$$

where: $a \prec b$ iff:

- $a, b < (p+1)/2 \wedge a < b$, or:
- $a \geq (p+1)/2 \wedge b < (p+1)/2$. or:
- $a, b \geq (p+1)/2 \wedge a < b$

Results The verification via the tool Rosette results in counterexamples. In particular, $(x_{sgn}, x_{mag}) = (1, 0)$ is considered positive by the function, which either is incorrect, or misleading and require further documentation of the intended meaning of "positive". Another closely related problem is that zero has two representations: $(1, 0)$ and $(-1, 0)$, as reported at V-O1J-VUL-015.

```
#lang rosette

(require rosette/solver/smt/cvc4
         rosette/base/core/felt)

(current-solver (cvc4 #:path (find-executable-path "cvc5")
                      #:options (hash ':ff-solver 'int)))

(define P
   28948022309329048855892746252171976963363056481941560715954676764349967630337)

(define F (finfield P))
(define halfP (/ (add1 P) 2))
```

```
12
13 (define-symbolic magx sgnx F)
14
15 (define zero (felt 0 F))
16 (define one (felt 1 F))
17
18 (define 2^64 (felt (expt 2 64) F))
19
20 ;; domain restriction
21 (assert (|| (@ffeq sgnx one) (@ffeq sgnx (@ff.neg one))))
22 (assert (@ff.lt magx 2^64))
23
24 (define x (@ff.mul magx sgnx))
25
26 ;; custom < that "knows" how to compare "negative numbers"
27 (define (mylt x y)
28   (cond
29     [(&& (@ff.lt x (felt halfP F))
30          (@ff.lt y (felt halfP F)))
31      (@ff.lt x y)]
32     [(@ff.lt x (felt halfP F))
33      ;; y is negative
34      #f]
35     [(@ff.lt y (felt halfP F))
36      ;; x is negative
37      #t]
38     [else (@ff.lt x y)]))
39
40 (verify (assert (<=> (mylt zero x) (@ffeq sgnx one))))
```

Snippet 5.3: The verification program

5.3.4 V-O1J-PROP-004: Correctness of divideAndRound

Verification Status Verified

Natural Language We verified that the `divideAndRound` function in `o1js-bindings/crypto/elliptic-curve-endomorphism.ts` is correct according to the specification in https://iacr.org/archive/crypto2001/21390189.pdf: that $|\beta - b| \leq \frac{1}{2}$.

Formal We want to verify that:

$$|\beta - b| \leq \frac{1}{2}$$

where:

$$b = \mathsf{divideAndRound}(x, y)$$
$$\beta = x/y$$

Equivalently, we want to verify that:

$$2|x - yb| \leq |y|$$

Results The verification via the tool Rosette succeeds. It should be noted that there's an ambiguity of the rounding direction when β is a middle point between two integers, though this doesn't affect the correctness stated above. See V-O1J-VUL-015 for more information.

```
#lang rosette

(require rosette/solver/smt/cvc4)

(current-solver (cvc4 #:path (find-executable-path "cvc5")))

(define-symbolic x y integer?)

(define (sign x)
  (if (>= x 0) 1 -1))

(define (div-and-round x y)
  (define signz (* (sign x) (sign y)))
  (set! x (abs x))
  (set! y (abs y))
  (define z (quotient x y))
  (when (>= (* 2 (- x (* z y))) y)
    (set! z (add1 z)))
  (* z signz))

```

```
21 (define z (div-and-round x y))
22
23 (verify (assert (<= (* 2 (abs (- (* z y) x))) (abs y))))
```

Snippet 5.4: The verification program

5.3.5 V-O1J-PROP-005: Correctness of maybeSwap

Verification Status Verified

Natural Language We verified that the `maybeSwap` function in `o1js/src/lib/provable/merkle-tree.ts` is correct.

Formal We first assume that the inputs are within the correct range:

$$b = 0 \vee b = 1$$

We want to verify under the assumptions that:

$$\begin{aligned} m &= b \times (x - y) \\ x\_ &= y + m \\ y\_ &= x - m \\ actual &= (x\_, y\_) \\ expected &= \begin{cases} (x\_, y\_) & \text{if } b = 1 \\ (y\_, x\_) & \text{if } b = 0 \end{cases} \end{aligned}$$

whether the following formula holds:

$$actual = expected$$

Results The verification via the tool Rosette succeeds. However, the constraint extraction process reveals a potential issue that the function may be used incorrectly due to insufficient documentation (V-O1J-VUL-015).

```
#lang rosette

(require rosette/solver/smt/cvc4
         rosette/base/core/felt)

(current-solver (cvc4 #:path (find-executable-path "cvc5")))

(define F (finfield
    28948022309329048855892746252171976963363056481941560715954676764349967630337)
    )

(define-symbolic b x y F)

(define zero (felt 0 F))
(define one (felt 1 F))

```

```
(define (ff.sub x y)
  (@ff.add x (@ff.neg y)))

;; domain restriction
(assert (|| (@ffeq b zero) (@ffeq b one)))

(define m (@ff.mul b (ff.sub x y)))
(define x_ (@ff.add y m))
(define y_ (ff.sub x m))
(define actual (list x_ y_))

(define expected
  (cond
    [(@ffeq b zero) (list y x)]
    [else (list x y)]))

(verify (assert (apply && (map @ffeq actual expected))))
```

Snippet 5.5: The verification program

5.3.6 V-O1J-PROP-006: Determinism of Bool.equals

Verification Status Verified

Natural Language We verified that the `Bool.equals` method in `o1js/src/lib/provable/bool.ts` is deterministic.

Formal We first assume that the inputs are well-formed.

$$x = 0 \vee x = 1$$
$$y = 0 \vee y = 1$$

We then apply the constraints from `Bool.equals`.

$$2xy = x + y - z_1$$
$$2xy = x + y - z_2$$
$$z_1 = 0 \vee z_1 = 1$$
$$z_2 = 0 \vee z_2 = 1$$
$$o_1 = 1 - z_1$$
$$o_2 = 1 - z_2$$

Finally, we verify whether the following formula holds:

$$o_1 = o_2$$

Results The verification via the tool Picus succeeds.

```
(prime-number
   28948022309329048855892746252171976963363056481941560715954676764349967630337)


(input x y)
(output out)

(assert (= (* x (- x 1)) 0))
(assert (= (* y (- y 1)) 0))
(assert (= (* (+ x x) y) (- (+ x y) z)))
(assert (= (* z (- z 1)) 0))
(assert (= out (- 1 z)))
```

Snippet 5.6: The verification program

5.3.7 V-O1J-PROP-007: Determinism of Field.equals

Verification Status Verified

Natural Language We verified that the `Field.equals` method in `o1js/src/lib/provable/field.ts` is deterministic.

Formal We want to verify under the assumptions that:

$$\begin{aligned} b_1(x-y) &= 0 \\ b_2(x-y) &= 0 \\ z_1(x-y) &= 1-b_1 \\ z_2(x-y) &= 1-b_2 \end{aligned}$$

whether the following formula holds:

$$b_1 = b_2$$

Results The verification via the tool Picus succeeds.

```
(prime-number
    28948022309329048855892746252171976963363056481941560715954676764349967630337)


(input x y)
(output b)

(assert (= XminusY (- x y)))

(assert (= (* b XminusY) 0))
(assert (= (* z XminusY) (- 1 b)))
```

Snippet 5.7: The verification program

5.3.8 V-O1J-PROP-008: Determinism of Field.inv

Verification Status Verified

Natural Language We verified that the `Field.inv` function in `o1js/src/lib/provable/field.ts` is deterministic.

Formal First, we apply the circuit constraints:

$$z_1 \times x = 1$$
$$z_2 \times x = 1$$

We then verify whether the following formula holds:

$$z_1 = z_2$$

Results The verification via the tool Picus succeeds.

```
(prime-number
   28948022309329048855892746252171976963363056481941560715954676764349967630337)

(input x)
(output z)

(assert (= (* x z) 1))
```

Snippet 5.8: The verification program

5.3.9 V-O1J-PROP-009: Determinism of Field.isOdd

Verification Status Verified

Natural Language We verified that the `Field.isOdd` method in `o1js/src/lib/provable/field.ts` is deterministic.

Formal We want to verify under the assumptions that:

$$
\begin{aligned}
b_1 = 0 \lor b_1 = 1 \\
b_2 = 0 \lor b_2 = 1 \\
z_1 < (p+1)/2 \\
z_2 < (p+1)/2 \\
x = b_1 + 2z_1 \\
x = b_2 + 2z_2 \\
\alpha_1 = 0 \lor \alpha_1 = 1 \\
\alpha_2 = 0 \lor \alpha_2 = 1 \\
\alpha_1 = 1 \iff x \neq 0 \\
\alpha_2 = 1 \iff x \neq 0 \\
y_1 = 0 \lor y_1 = 1 \\
y_2 = 0 \lor y_2 = 1 \\
y_1 = 1 \iff (b_1 = 1 \land \alpha_1 = 1) \\
y_2 = 1 \iff (b_2 = 1 \land \alpha_2 = 1)
\end{aligned}
$$

whether the following formula holds:

$$y_1 = y_2$$

Results The verification via the tool Picus succeeds.

```
(input in)
(output out)

(prime-number
    28948022309329048855892746252171976963363056481941560715954676764349967630337)


(assert (= (* b (- b 1)) 0))

(assert (= Fdiv2
    1447401115466452442794637312608598848168152824097078035797733838217498381516 9)
    )
```

```
9
10 (assert (< z Fdiv2))
11
12 (assert (= in (+ b (* z 2))))
13
14 (assert (= (* is-non-zero (- is-non-zero 1)) 0))
15 (assert (<=> (= is-non-zero 1) (! (= in 0))))
16
17 (assert (= (* out (- out 1)) 0))
18 (assert (<=> (= out 1) (&& (= b 1) (= is-non-zero 1))))
```

Snippet 5.9: The verification program

5.3.10 V-O1J-PROP-010: Determinism of Field.sqrt

Verification Status Benign Counterexample

Natural Language We verified that the `Field.sqrt` function in `o1js/src/lib/provable/field.ts` is deterministic.

Formal First, we apply the circuit constraints:

$$z_1 \times z_1 = x$$
$$z_2 \times z_2 = x$$

We then verify whether the following formula holds:

$$z_1 = z_2$$

Results The verification via the tool Picus results in a counterexample, which is expected and documented. For example, $x = 1$ could have $z_1 = 1$ and $z_2 = -1$.

```
(prime-number
    28948022309329048855892746252171976963363056481941560715954676764349967630337)


(input x)
(output z)

(assert (= (* z z) x))
```

Snippet 5.10: The verification program

5.3.11 V-O1J-PROP-011: Determinism of Poseidon.hashToGroup

Verification Status Benign Counterexample

Natural Language We verified that the `Poseidon.hashToGroup` function in `o1js/src/lib/provable/crypto/poseidon.ts` is deterministic.

Formal Assuming that the Snarky implementation is deterministic, we apply the post-computation constraints:

$$\begin{aligned} x_1 &= xv \\ x_2 &= xv \\ x_1^1 &= x_1^0 \times -1 \\ x_2^1 &= x_2^0 \times -1 \\ yv &= x_1^0 \vee yv = x_1^1 \\ yv &= x_2^0 \vee yv = x_2^1 \end{aligned}$$

Finally, we verify whether the following formula holds:

$$x_1 = x_2 \wedge x_1^0 = x_2^0 \wedge x_1^1 = x_2^1$$

Results The verification via the tool Picus results in a counterexample, where (x^0, x^1) could be swapped to (x^1, x^0), as reported at V-O1J-VUL-015. However, we can also prove that when x^0 is further constrained to be an even number, the verification becomes successful.

```
(prime-number
   28948022309329048855892746252171976963363056481941560715954676764349967630337)


(input x y)
(output xo x0 x1)

(assert (= xo x))
(assert (= x1 (- x0)))
(assert (|| (= y x0) (= y x1)))
```

Snippet 5.11: The verification program

5.3.12 V-O1J-PROP-012: Determinism of UInt64.divMod (and UInt32.divMod and UInt8.divMod)

Verification Status Verified

Natural Language We verified that the `divMod` methods in `o1js/src/lib/provable/int.ts` is deterministic.

Formal For `UInt64.divMod`, we first assume that the inputs are within the correct range:

$$x < 2^{64}$$
$$y < 2^{64}$$

We want to verify under the assumptions that:

$$q_1 < 2^{64}$$
$$q_2 < 2^{64}$$
$$r_1 = x - q_1 y$$
$$r_2 = x - q_2 y$$
$$r_1 < 2^{64}$$
$$r_2 < 2^{64}$$
$$r_1 < y$$
$$r_2 < y$$

whether the following formula holds:

$$q_1 = q_2$$
$$r_1 = r_2$$

Other bit-widths are similar.

Results The verification via the tool Picus succeeds.

```
(prime-number
    28948022309329048855892746252171976963363056481941560715954676764349967630337)


(input x y)
(output q r)

(assert (= MAX_INT 18446744073709551616))
```



```
7
8  ;; precondition
9  (assert (< x MAX_INT))
10 (assert (< y MAX_INT))
11
12 (assert (< q MAX_INT))
13 (assert (< r MAX_INT))
14 (assert (= r (- x (* q y))))
15 (assert (< r y))
```

Snippet 5.12: The verification program

5.3.13 V-O1J-PROP-013: Determinism of addMod32

Verification Status Verified

Natural Language We verified that the `addMod32` function in `o1js/src/lib/provable/gadgets/arithmetic.ts` is deterministic.

Formal We first assume that the inputs are well-formed.

$$x < 2^{64}$$
$$y < 2^{64}$$

We then apply the constraints from `addMod32`.

$$n_1 = x + y$$
$$n_2 = x + y$$
$$q_1 < 2^1$$
$$q_2 < 2^1$$
$$r_1 < 2^{32}$$
$$r_2 < 2^{32}$$
$$n_1 = 2^{32} q_1 + r_1$$
$$n_2 = 2^{32} q_2 + r_2$$

Finally, we verify whether the following formula holds:

$$r_1 = r_2$$

Results The verification via the tool Picus succeeds.

```
(prime-number
    28948022309329048855892746252171976963363056481941560715954676764349967630337)


(input x y)
(output remainder)

;; assumption
(assert (= p-64 18446744073709551616))
(assert (< x p-64))
(assert (< y p-64))

```



```
(assert (= n (+ x y)))

;; quotientBits = 1
(assert (= p-qbits 2))

(assert (= p-32 4294967296))

(assert (< quotient p-qbits))
(assert (< remainder p-32))

(assert (= n (+ (* quotient p-32) remainder)))
```

Snippet 5.13: The verification program

5.3.14 V-O1J-PROP-014: Determinism of arrayGet

Verification Status Partially Verified

Natural Language We verified that the `arrayGet` function in `o1js/src/lib/provable/basic.ts` is deterministic.

Formal We first assume that the inputs are well-formed.

$$i < K$$

We then apply the constraints from `arrayGet`. For each $0 \leq j < K$:

$$z_1[j] \times (i - j) = out_1 - a[j]$$
$$z_2[j] \times (i - j) = out_2 - a[j]$$

Finally, we verify whether the following formula holds:

$$out_1 = out_2$$

Results The verification via the tool Picus (mostly) succeeds. We performed the verification with $K = 1, 3, 10$, where the verification on $K = 10$ is inconclusive due to Picus timing out.

```
(prime-number
   28948022309329048855892746252171976963363056481941560715954676764349967630337)


(input a0 a1 a2 i)
(output a)

;; assumption
(assert (< i 3))

(assert (= (* z0 (- i 0)) (- a a0)))
(assert (= (* z1 (- i 1)) (- a a1)))
(assert (= (* z2 (- i 2)) (- a a2)))
```

Snippet 5.14: The verification program for $K = 3$

5.3.15 V-O1J-PROP-015: Determinism of divMod32

Verification Status Counterexample

Natural Language We verified that the `divMod32` function in `o1js/src/lib/provable/gadgets/arithmetic.ts` is deterministic.

Formal We first assume that the inputs are well-formed:

$$n < 2^{64}$$

We want to verify under the assumptions that:

$$\begin{aligned} q_1 &< 2^{\mathsf{qb}} \\ q_2 &< 2^{\mathsf{qb}} \\ r_1 &< 2^{32} \\ r_2 &< 2^{32} \\ n &= 2^{32} q_1 + r_1 \\ n &= 2^{32} q_2 + r_2 \end{aligned}$$

whether the following formula holds:

$$\begin{aligned} q_1 &= q_2 \\ r_1 &= r_2 \end{aligned}$$

The value qb is chosen among 32, 222, and 223.

Results The verification via the tool Picus succeeds for $\mathsf{qb} = 32, 222$. For $\mathsf{qb} = 223$, we found a counterexample, which is reported at V-O1J-VUL-015:

$$\begin{aligned} n &= 0 \\ q_1 = 67399866667876599486667537717549076684198939432253969637571547097 41, q_2 &= 0 \\ r_1 = 1, r_2 &= 0 \end{aligned}$$

```
(prime-number
   28948022309329048855892746252171976963363056481941560715954676764349967630337)


(input n)
(output remainder quotient)

;; assumption
(assert (= p-64 18446744073709551616))
```

```
(assert (< n p-64))

;; quotientBits = 32
(assert (= p-qbits 4294967296))

(assert (= p-32 4294967296))

(assert (< quotient p-qbits))
(assert (< remainder p-32))

(assert (= n (+ (* quotient p-32) remainder)))
```

Snippet 5.15: The verification program

5.3.16 V-O1J-PROP-016: Determinism of isZero

Verification Status Verified

Natural Language We verified that the `isZero` function in `o1js/src/lib/provable/gadgets/comparison.ts` is deterministic.

Formal We want to verify under the assumptions that:

$$
\begin{aligned}
b_1 x &= 0 \\
b_2 x &= 0 \\
1 - b_1 &= z_1 x \\
1 - b_2 &= z_2 x
\end{aligned}
$$

whether the following formula holds:

$$b_1 = b_2$$

Results The verification via the tool Picus succeeds.

```
(prime-number
    28948022309329048855892746252171976963363056481941560715954676764349967630337)


(input x)
(output b)

(assert (= 0 (* b x)))
(assert (= (- 1 b) (* z x)))
```

Snippet 5.16: The verification program

5.3.17 V-O1J-PROP-017: Determinism of lessThanOrEqualGeneric (and lessThanGeneric)

Verification Status Verified

Natural Language We verified that the `lessThanOrEqualGeneric` and `lessThanGeneric` functions in `o1js/src/lib/provable/gadgets/comparison.ts` is deterministic.

Formal For `lessThanOrEqualGeneric`, we first assume that the inputs are well-formed:

$$c \leq (p-1)/2$$

We want to verify under the assumptions that:

$$\begin{aligned} b_1 = 0 \lor b_1 = 1 \\ b_2 = 0 \lor b_2 = 1 \\ x + b_1 c - y - 1 < c \\ x + b_2 c - y - 1 < c \end{aligned}$$

whether the following formula holds:

$$b_1 = b_2$$

For `lessThanGeneric`, the subtraction of 1 is omitted.

Results The verification via the tool Picus succeeds.

```
(prime-number
    28948022309329048855892746252171976963363056481941560715954676764349967630337)

(input x y c)
(output b)

;; assumption
;; c * 2 <= p
;; c <= p / 2
;; c <= (p - 1) / 2
(assert (<= c
    1447401115466452442794637312608598848168152824097078035797733838217498381516 8)
    )

(assert (|| (= b 0) (= b 1)))
```



```
13 (assert (< (- (- (+ x (* b c)) y) 1) c))
```

Snippet 5.17: The verification program for `lessThanOrEqualGeneric`

5.3.18 V-O1J-PROP-018: Equivalence of Ch

Verification Status Verified

Natural Language We verified that the `Ch` function in `o1js/src/lib/provable/gadgets/sha256.ts` is equivalent to the unoptimized variant.

Formal We first assume that the inputs are well-formed:

$$0 \leq x < 2^{32}$$
$$0 \leq y < 2^{32}$$
$$0 \leq z < 2^{32}$$

We want to verify under the assumptions that:

$$v_1 = (x \& y) \oplus (\bar{x} \& z)$$
$$v_2 = (x \& y) + (\bar{x} \& z)$$

whether the following formula holds:

$$v_1 = v_2$$

Results The verification via the tool Rosette succeeds.

```
#lang rosette

(require rosette/solver/smt/cvc4)

(current-solver (cvc4 #:path (find-executable-path "cvc5")))

(define BV (bitvector 260))

(define (do-it)
  (define p32 (bv (expt 2 32) BV))

  (define (assume-within x bound)
    (assume (bvsge x (bv 0 BV)))
    (assume (bvsgt bound x)))

  (define-symbolic x y z BV)

  (assume-within x p32)
```



```
  (assume-within y p32)
  (assume-within z p32)

  (define xAndY (bvand x y))
  (define xNotAndZ (bvand (bvnot x) z))
  (define ch (bvadd xAndY xNotAndZ))

  (define result-1 ch)
  (define result-2 (bvxor xAndY xNotAndZ))

  (verify (assert (bveq result-1 result-2))))

(do-it)
```

Snippet 5.18: The verification program

5.3.19 V-O1J-PROP-019: Equivalence of Maj

Verification Status Verified

Natural Language We verified that the `Maj` function in `o1js/src/lib/provable/gadgets/sha256.ts` is equivalent to the unoptimized variant.

Formal We first assume that the inputs are well-formed:

$$0 \leq x < 2^{32}$$
$$0 \leq y < 2^{32}$$
$$0 \leq z < 2^{32}$$

We want to verify under the assumptions that:

$$v_1 = ((x + y + z) - (x \oplus y \oplus z))/2$$
$$v_2 = (x \& y) \oplus (x \& z) \oplus (y \& z)$$

whether the following formula holds:

$$v_1 = v_2$$

Results The verification via the tool Rosette succeeds.

```
#lang rosette

(require rosette/solver/smt/cvc4)

(current-solver (cvc4 #:path (find-executable-path "cvc5")))

(define BV (bitvector 260))

(define (do-it)
  (define p32 (bv (expt 2 32) BV))

  (define (assume-within x bound)
    (assume (bvsge x (bv 0 BV)))
    (assume (bvsgt bound x)))

  (define-symbolic x y z BV)

  (assume-within x p32)
```



```
  (assume-within y p32)
  (assume-within z p32)

  (define sum (bvadd x y z))
  (define xor (bvxor x y z))

  (define result-1 (bvsdiv (bvsub sum xor) (bv 2 BV)))
  (define result-2 (bvxor (bvand x y) (bvand x z) (bvand y z)))

  (verify (assert (bveq result-1 result-2)))

(do-it)
```

Snippet 5.19: The verification program

5.3.20 V-O1J-PROP-020: Range analysis of sigma

Verification Status Verified

Natural Language We verified that the `sigma` function in `o1js/src/lib/provable/gadgets/sha256.ts` proves the correct range.

Formal We first assume that the inputs are well-formed:

$$\begin{aligned} 0 \leq x < 2^{32} \\ d_0 > 0 \\ d_1 > 0 \\ d_2 > 0 \\ d_3 > 0 \\ d_0 + d_1 + d_2 + d_3 = 32 \end{aligned}$$

We want to verify under the assumptions that:

$$\begin{aligned} 0 \leq x_0 < 2^{16} \\ 0 \leq x_1 < 2^{16} \\ 0 \leq x_2 < 2^{16} \\ 0 \leq x_3 < 2^{16} \\ x = x_0 + x_1 2^{d_0} + x_2 2^{d_0+d_1} + x_3 2^{d_0+d_1+d_2} \\ \mathsf{xRotR0} = \begin{cases} x_1 + x_2 2^{d_1} + x_3 2^{d_1+d_2} + x_0 2^{d_1+d_2+d_3} & \text{if shifted} \\ x_1 + x_2 2^{d_1} + x_3 2^{d_1+d_2} & \text{otherwise} \end{cases} \\ \neg\text{shifted} \implies x_0 < 2^{d_0} \\ \mathsf{xRotR1} = x_2 + x_3 2^{d_2} + x_0 2^{d_2+d_3} + x_1 2^{d_2+d_3+d_0} \\ \mathsf{xRotR2} = x_3 + x_0 2^{d_3} + x_1 2^{d_3+d_0} + x_2 2^{d_3+d_0+d_1} \\ \mathsf{xRotR0} < 2^{32} \\ \mathsf{xRotR1} < 2^{32} \\ \mathsf{xRotR2} < 2^{32} \end{aligned}$$

whether the following formula holds:

$$x_0 < 2^{d_0} \wedge x_1 < 2^{d_1} \wedge x_2 < 2^{d_2}$$

Results The verification via the tool Rosette succeeds.


```
#lang rosette

(require rosette/solver/smt/cvc4)

(current-solver (cvc4 #:path (find-executable-path "cvc5")))

(define BV (bitvector 260))

(define (do-it)
  (define p32 (bv (expt 2 32) BV))
  (define p16 (bv (expt 2 16) BV))

  (define-symbolic x x0 x1 x2 x3 d0 d1 d2 d3 BV)
  (define-symbolic first-shifted? boolean?)

  (define (assume-within x bound)
    (assume (bvsge x (bv 0 BV)))
    (assume (bvsgt bound x)))

  (assume-within x p32)
  (assume-within x0 p16)
  (assume-within x1 p16)
  (assume-within x2 p16)
  (assume-within x3 p16)

  (assume (bveq (bvadd d0 d1 d2 d3) (bv 32 BV)))

  (assume-within d0 p16)
  (assume-within d1 p16)
  (assume-within d2 p16)
  (assume-within d3 p16)

  (assume (bveq x (bvadd x0
                         (bvshl x1 d0)
                         (bvshl x2 (bvadd d0 d1))
                         (bvshl x3 (bvadd d0 d1 d2)))))

  (define xRotR0
    (cond
      [first-shifted?
       (bvadd x1
              (bvshl x2 d1)
              (bvshl x3 (bvadd d1 d2))
              (bvshl x0 (bvadd d1 d2 d3)))]
      [else
       (assume (bvslt x0 (bvshl (bv 1 BV) d0)))
       (bvadd x1
```

```
48              (bvshl x2 d1)
49              (bvshl x3 (bvadd d1 d2)))]))
50
51  (define xRotR1
52    (bvadd x2
53           (bvshl x3 d2)
54           (bvshl x0 (bvadd d2 d3))
55           (bvshl x1 (bvadd d2 d3 d0))))
56
57  (define xRotR2
58    (bvadd x3
59           (bvshl x0 d3)
60           (bvshl x1 (bvadd d3 d0))
61           (bvshl x2 (bvadd d3 d0 d1))))
62
63  (assume (bvslt xRotR0 p32))
64  (assume (bvslt xRotR1 p32))
65  (assume (bvslt xRotR2 p32))
66
67  (verify (assert (&& (bvslt x0 (bvshl (bv 1 BV) d0))
68                      (bvslt x1 (bvshl (bv 1 BV) d1))
69                      (bvslt x2 (bvshl (bv 1 BV) d2))))))
70
71 (do-it)
```

Snippet 5.20: The verification program

Fuzz Testing 6

6.1 Methodology

The Veridise auditors' goal was to fuzz test o1js to assess its functional correctness (i.e, whether the implementation deviates from the intended behavior) and robustness (i.e, whether the implementation properly handles malformed input). The fuzz testing focused on two regions of particular interest to the auditors, described in Section 6.2.

The Veridise team used two different approaches for fuzzing during this audit. For one specification (V-O1J-SPEC-001), they used an off-the-shelf coverage-guided fuzzer (jsfuzz*). For the other, the Veridise auditors wrote a custom fuzzer designed to generate random and complex `typeObjs` for conversion to `provable` types.

6.2 Properties Fuzzed

Table 6.1 describes the fuzz-tested invariants. The second column describes the invariant informally in English, and the third shows the total amount of compute time spent fuzzing this property. The last column indicates the number of bugs identified when fuzzing the invariant.

The Veridise auditors devoted a total of 410 compute-hours to fuzzing this protocol, identifying a total of 1 bug.

Table 6.1: Invariants Fuzzed.

Specification	Invariant	Minutes Fuzzed	Bugs Found
V-O1J-SPEC-001	Serialization and deserialization are inverses	1500	1
V-O1J-SPEC-002	fastInverse() for finite fields is correct	23100	0

* `https://github.com/0xddom/jsfuzz`

6.3 Detailed Description of Fuzzed Specifications

6.3.1 V-O1J-SPEC-001: Serialization and deserialization are inverses

Minutes Fuzzed	1500	Bugs Found	1

Scope The file `bindings/lib/provable-generic.ts` defines the `provable()` protocol to serialize a *provable* value to field elements and deserialize field elements back to the value. This is done through the `toFields` (serialization) and `fromFields` (deserialization) functions/methods. These functions should be inverses of each other.

More precisely, there are three functions at play.

- `toFields` is used for serializing the in-circuit portion of a value to field elements.
- `toAuxiliary` is used for recording the non-in-circuit portion of a value as auxiliary data.
- `fromFields` combines the field elements and auxiliary data back to a value.

These functions work by additionally reading a "typed object", which contains the value shape. The shape is then used for determining how the serialization/deserialization should be done.

In similar fashion, the file `bindings/lib/from-layout.ts` defines the `provableFromLayout()` protocol to serialize a value to field elements and deserialize field elements back to a value. The difference from the `provable()` protocol is that a "layout" is used in place of a "typed object". While the details of "layout" and "typed object" are different, they are functionally the same in that they provide the schema to instruct how serialization/deserialization should be done.

By ensuring that serialization and deserialization form inverses, it follows that:

- Any "variable-length" objects (optional fields, variable-length arrays), which are not in-circuit, should not be converted to `Field`s. Conversely, the auxiliary portions can contain non-in-circuit objects.
- `sizeInFields()` should be a static method for any instance of the provable type.
- Complex structures must be properly iterated over.
- Name collisions (of the `typeObj`/static members/functions) should not silently cause issues.
- Order of declaration should not lead to issues.
- Prototype/class inheritance should not lead to problems.

Note that because the typed objects / layouts are the instructions on how to perform serialization, and not the value itself, we expect the actual data values stored should not affect control flow.

Typed objects and layouts Both typed objects and layouts are constructed inductively.

In particular, a typed object is either:

- a leaf typed object: `Field`, `Bool`, `UInt64`, etc.
- a primitive type: `String`, `Number` (which would be converted to auxiliary data)
- a tuple of typed objects
- an object of typed objects.

For example, a typed object `{ two_fields: [Field, Field], symbol: String }` is an object that contains two fields. The `two_fields` field contains another typed object, which is an array. The array contains two typed object "leaves" `Field`. The `symbol` field contains a typed object `String`.

With a typed object, we could then use `provable()` (defined in `lib/provable/types/provable-derivers.js`) to perform serialization/de-seralization. E.g.:

```
import { provable } from '../lib/provable/types/provable-derivers.js';
import { Field } from 'o1js';

// provable(typedObj)
let ProvableW = provable({ two_fields: [Field, Field], symbol: String });

// object to be serialized
let obj = { two_fields: [new Field(1), new Field(2)], symbol: "abc" };

// serialized
let fields = ProvableW.toFields(obj);

// auxiliary data
let aux = ProvableW.toAuxiliary(obj);

// deserialized
let newObj = ProvableW.fromFields(fields, aux);
```

Layouts are similar. However, the structure of a layout follows https://github.com/o1-labs/o1js-bindings/blob/10e33c03648fa6d2201d2665cd2fde7064525ac8/lib/from-layout.ts#L673-L731. With a layout, we could use `provableFromLayout()` (defined in `bindings/mina-transaction/gen/transaction.ts`) to perform serialization/deseralization. E.g.:

```
import { provableFromLayout } from '../bindings/mina-transaction/gen/transaction.js';
import { Field, UInt64 } from 'o1js';

type Layout = {
    foo: UInt64;
    bar: Field;
};

let Layout = provableFromLayout<Layout, any>({
    type: 'object',
    name: 'Layout',
    keys: [
        'foo',
        'bar',
    ],
    entries: {
        foo: { type: 'UInt64' },
        bar: { type: 'Field' },
    }
});

```



```
let obj = {
    foo: UInt64.from(1),
    bar: Field(2),
};

let fields = Layout.toFields(obj);
let aux = Layout.toAuxiliary(obj);
let newObj = Layout.fromFields(fields, aux);
```

Natural Language `toFields()` and `fromFields()` must be inverses (modulo `toAuxiliary()`).

One direction is to start with a value, convert to fields, and then convert back to the value. This involves randomly generating a value and its corresponding typed object / layout, performing the round-trip conversion, and finally comparing that the objects are equal.

Formal

$$\begin{aligned} &\forall obj \in \text{Object}, typedObj \in \text{TypedObject}, \\ &\text{compatible}(typedObj, obj) \implies \\ &\text{fromFields}(typedObj, \text{toFields}(typedObj, obj), \text{toAuxiliary}(typedObj, obj)) = obj \end{aligned}$$

where Object, TypedObject, and Fields are a set of all possible objects that we are interested in, typed objects, and array of field elements. compatible predicate is used as a guard to make sure that the type object is the shape of the object.

Results The `provable` object constructor will check if the object type implements a particular method that matches the serialization API. If this API is only implemented partially then the behavior is undefined and in some cases the property here presented doesn't hold despite the serialization and deserialization executing without errors or warnings.

For example, a `typeObj` like the one below will serialize without error, but will fail to de-serialize.

```
{
  x: Field,
  toFields(): [Field] {
      return [x];
  }
}
```

To avoid this problem we propose the following recommendations:

- When creating the provable object check if the object implements the custom serialization and deserialization methods as a whole, to avoid partial implementations.
- Use method names that are uncommon for the language, such as a double underscore prefix and snake case. This will make accidental collisions less common and will make the intention of the developer more explicit.

No other issues were uncovered.

6.3.2 V-O1J-SPEC-002: fastInverse() for finite fields is correct

Minutes Fuzzed	23100	Bugs Found	0

Scope The `createField()` function, defined in `o1js/src/bindings/crypto/finite-field.ts`, returns an object with various finite-field methods. The `inverse()` function computes the inverse of a value (modulo the field), or returns `undefined` if no inverse exists.

Under the hood, it is usually implemented using the `fastInverse()` function, which is based on Thomas Pornin's "Optimized Binary GCD for Modular Inversion" (pre-print available online: `https://eprint.iacr.org/2020/972.pdf`). For some choices of `p`, a more standard extended GCD algorithm is used.

While modular arithmetic modulo `p` is only a field for prime `p`, this function is intended to work correctly for any modulus `p >= 2`.

Natural Language For any `p >= 2` and any `x`, `createField(p).inverse(x)` computes the inverse of `x` if it exists.

Formal Let `p: bigint`, `x: bigint` with `p >= 2`. Let `Fp = createField(p)`. Let `gcd(x,p)` be the greatest common denominator of `x` and `p`.

- If `gcd(x, p) !== 1n`:
  - if `mod(x, p) === 0n` then `Fp.inverse(x) === undefined`.
  - otherwise,
    - if using `fastInverse()`, `Fp.inverse` will crash.
    - if using `inverse()`, `Fp.inverse` will return `undefined`.
- `Fp.inverse(x) === undefined` only if `gcd(x, p) !== 1n`.
- For all `x` with `gcd(x,p) === 1n`, `Fp.mul(Fp.inverse(x), x) === 1n`.

Results After over 380 compute-hours of fuzzing, no failures or unexpected crashes were identified.

Glossary

Mina Mina Protocol is a succinct 22KB blockchain utilizing zero-knowledge proofs. See `https://minaprotocol.com` for more details . 1

PLONK An arithmetization strategy for zero-knowledge circuits developed in [4]. See [5] or `https://vitalik.ca/general/2019/09/22/plonk.html` for more details. 1

Satisfiability Modulo Theories The problem of determining whether a certain mathematical statement has any solutions. SMT solvers attempt to do this automatically. See `https://en.wikipedia.org/wiki/Satisfiability_modulo_theories` to learn more . 191

Semgrep Semgrep is an open-source, static analysis tool. See `https://semgrep.dev` to learn more . 7

smart contract A self-executing contract with the terms directly written into code. Hosted on a blockchain, it automatically enforces and executes the terms of an agreement between buyer and seller. Smart contracts are transparent, tamper-proof, and eliminate the need for intermediaries, making transactions more efficient and secure.. 1

SMT Satisfiability Modulo Theories. 147

zero-knowledge circuit A cryptographic construct that allows a prover to demonstrate to a verifier that a certain statement is true, without revealing any specific information about the statement itself. See `https://en.wikipedia.org/wiki/Zero-knowledge_proof` for more . 1, 191

Bibliography

[1] Emina Torlak and Rastislav Bodik. 'A lightweight symbolic virtual machine for solver-aided host languages'. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 530–541. DOI: 10.1145/2594291.2594340 (cited on page 147).

[2] Alex Ozdemir et al. 'Satisfiability Modulo Finite Fields'. In: *Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part II*. Paris, France: Springer-Verlag, 2023, pp. 163–186. DOI: 10.1007/978-3-031-37703-7_8 (cited on page 147).

[3] Shankara Pailoor et al. 'Automated Detection of Under-Constrained Circuits in Zero-Knowledge Proofs'. In: *Proc. ACM Program. Lang.* 7.PLDI (2023). DOI: 10.1145/3591282 (cited on page 147).

[4] Ariel Gabizon, Zachary J. Williamson, and Oana-Madalina Ciobotaru. 'PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge'. In: *IACR Cryptol. ePrint Arch.* 2019 (2019), p. 953 (cited on page 191).

[5] Justin Thaler. 'Proofs, Arguments, and Zero-Knowledge'. In: *Foundations and Trends® in Privacy and Security* 4.2–4 (2022), pp. 117–660. DOI: 10.1561/3300000030 (cited on page 191).