

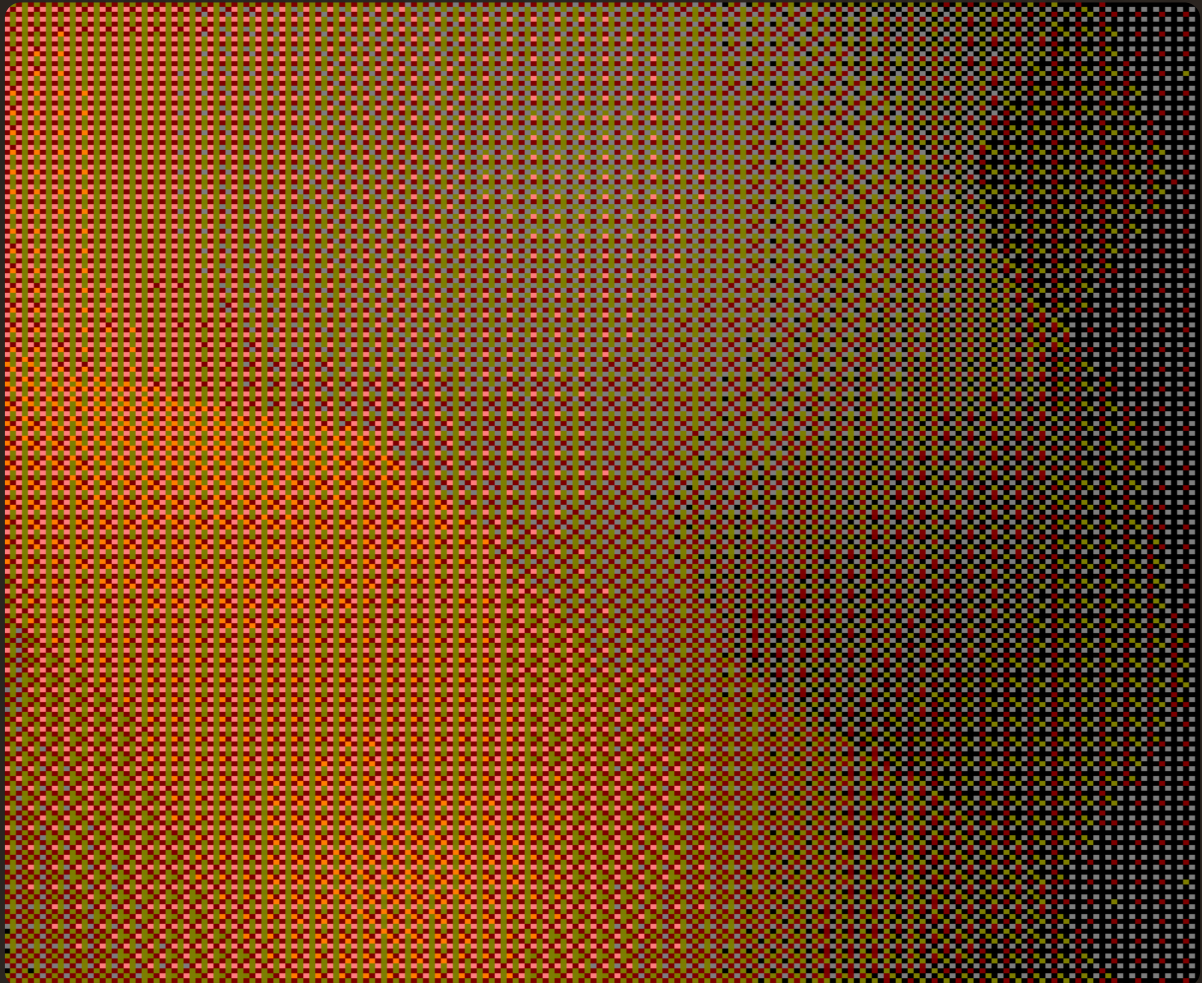
ZKSECURITY

Adversarial Testing of the Ironwood Formal Verification

ZCASH

• SEPTEMBER 4TH,
2026

• TECHNICAL
REPORT



Introduction

On August 10th, 2026, zkSecurity started an adversarial testing engagement of [Ironwood](#), Zcash's Lean 4 formalization of the Orchard Action circuit and of the Halo2 proving stack that verifies Orchard proofs. The engagement lasted three weeks with three consultants.

We did not focus on conventional code review, instead the primary goal of the engagement was to use mutation testing: we injected knowledge-soundness bugs into the Rust implementation and into the Lean model, regenerated the artifacts that link the two, and checked whether the formalization rejected the incorrect implementations.

The Rust implementation under test lives in the [orchard](#) and [halo2](#) repositories. We used the revisions pinned by Ironwood's fixture provenance file:

REPOSITORY	COMMIT	ROLE
zcash/ironwood	<code>3c056cbe</code>	Lean 4 / Mathlib formalization; <code>lake build</code> is the verification build
zcash/orchard	<code>38bd2274</code>	Action circuit and fingerprint capture drivers (<code>0.15.5</code> plus PR #541)
zcash/halo2	<code>cafc26e2</code>	PLONK/IPA verifier and Lean fixture exporter (<code>halo2_proofs 0.3.5</code>)

Scope

The engagement covers adversarial testing of the Lean formalization in `zcash/ironwood` against Zcash's Ironwood circuit and Halo2 proving stack, using mutations to the circuits and to the Halo2 verifier logic, in both the

Lean model and the Rust source code, to evaluate whether potential implementation bugs are correctly detected by the formal verification.

The primary target of the formalization is knowledge-soundness bugs that could enable undetectable counterfeiting. Counterfeiting is undetectable when exploitation leaves no evidence in the proof, instance, or chain history that would allow it to be identified later.

The adversarial testing exercises the two links needed to establish the formalization's guarantees:

1. **Rust implementation to fixtures.** We inject undetectable knowledge-soundness bugs into the Rust implementation and regenerate the fixtures. An undetectable knowledge-soundness bug that leaves the fixtures unchanged is a violation of the trust boundary the formalization relies on.
2. **Fixtures to proof.** For any injected soundness bug that is reflected in the regenerated fixtures, we attempt to adapt the Lean model to those fixtures while keeping the specification and the critical top-level theorems fixed. If we are able to still prove the same top-level theorems, this indicates a gap in the knowledge-soundness formalization.

As part of the second category, we also look for statements in the formalization that are vacuously true, or more generally weaker than intended, and that would allow an unsound implementation to pass if only the proof is rewritten to exploit the statement's weakness.

As bonus coverage, we also apply the first test to detectable knowledge-soundness bugs, such as query-collision bugs whose exploitation is visible in the proof, and verifier backdoors whose exploitation can be detected by replaying the proof against a corrected verifier. A detectable bug that the fixtures fail to reflect lies outside the formalization's threat model, unless it corresponds to an actual bug in Ironwood.

The injected bugs range from missing or incorrect constraints to subtle bugs in the Halo2 verifier, including failures of adaptive soundness that let a prover exploit earlier Fiat-Shamir challenges when choosing later proof messages.

The test cases were informed in part by zkSecurity's [ZK bug dataset](#). The exact mutation set was developed during the engagement, with the goal of broad adversarial coverage that highlights any remaining gaps in the formalization.

The scope was extended during the engagement to include [Ironwood PR #215](#), which models Halo2's transcript bytes, BLAKE2b, proof encoding, and verifying-key digest. These changes resolve the verification-key gate-hash gap described below.

Zero knowledge, batch verification of several proof strings, and the correctness of the deployed consensus rules outside the Action proof are explicitly not part of the scope.

§ 02

Methodology

How Ironwood binds Rust to Lean

Ironwood does not verify the Rust code directly, instead it relies on a re-implementation of the verifier and the Action circuit in lean. The formalization relies on generated artifacts, called fixtures, that capture what the Rust verifier does on concrete inputs, and on Lean theorems that reconstruct the same objects from the formal model. We summarize this modeling here.

- **Verifier fingerprint fixtures.** Orchard's `capture_proof_fingerprint` runs the real Halo2 `verify_proof` with a fingerprint strategy that records the assembled final multi-scalar multiplication (MSM) and the challenge schedule without evaluating the MSM. Halo2's exporter turns the capture into a Lean file. There are four capture families: `SingleAction/Honest`, `SingleAction/Random`, `MultiAction/Honest`, and `MultiAction/Random`. For each family, the theorem `nonInteractiveFingerprint_matches_derived` states that Lean's independent reassembly of the verifier MSM agrees with the capture.

- **Verifying-key certificate.** The theorem `vk_eq_toVerifierKey` in `Zcash/Snark/Keygen/Certificate.lean` checks that the verifying key derived in Lean from the formal Action circuit equals the captured Rust key, including gates, query layouts, fixed and permutation commitments, lookup expressions, and domain parameters. This is used to argue equality of the lean circuit and the deployed circuit.
- **Circuit specification and extraction.** `Zcash/Circuits/Action/*` models the Action circuit's top-level interface. The specification `ActionSpec` and the extractor that recovers a witness from a satisfying trace are used to prove the Action capstones in `Zcash/Snark/Capstones/Action.lean`, and the ledger-level supply-integrity results.

The fingerprint captures are positive runs: honest accepting proofs, or random strings that parse and complete. They do not exercise rejection paths, and the capture stops before the final verdict, since the fingerprint strategy returns the assembled MSM and does not evaluate it.

Experiment protocol

Every experiment follows the same protocol, executed in isolated worktrees of the three repositories at their pinned revisions:

1. **Baseline.** Confirm that the clean stack accepts the honest proof and rejects the intended counterexample.
2. **Implementation-only stage.** Apply the Rust mutation behind an off-by-default feature flag, produce an accepting proof for a false statement where the mutation is a soundness bug, and regenerate all four fixture families. Record which families change and which stay byte-identical.
3. **Faithful mirroring stage.** Adapt the Lean model to the mutated behavior, keeping the specification and the glue theorems unchanged, and attempt to rebuild the anchored claims against the regenerated fixtures.
4. **Formal endpoint.** Record either a positive reconstruction that still passes (a gap), a Lean proof that the anchored claim no longer holds or

cannot be packaged (a catch), or a drift correspondence showing that the certificate rejects the mutated key.

One experiment departs from this shape. The circuit mutation testing applies a batch of eleven independent constraint removals and alterations to the Action circuit and its gadgets, and reports them together, since each one runs the same protocol and reaches the same endpoint.

Each report ends with one of the following classifications:

- **GAP:** a demonstrated Rust soundness bug is not reflected in the regenerated fixtures (link 1 fails), or is reflected and the faithfully adapted Lean development still proves the anchored claims (link 2 fails). A defect proved directly in an audited formal claim, without a Rust mutation, is also a gap.
- **NO_GAP:** a demonstrated Rust soundness bug is reflected in the fixtures and the faithful Lean adaptation cannot preserve the anchored claim.

§ 03

Summary

We executed 16 experiments, one of which bundles 11 common circuit bugs. Our experiments identify 6 gaps, 5 of which are known boundaries documented in the Ironwood book and code comments. The verification-key gate-hash gap was resolved by the in-scope changes in Ironwood PR #215. We note that all of the common circuit bugs are caught by the formal verification apparatus; for every bug, we prove that the circuit cannot satisfy the soundness theorem.

EXPERIMENT	CLASSIFICATION	FAILING LINK	DETECTABILITY
Exponential Circuit Assignment to Witness Extractor	GAP*	Fixtures to proof	Undetectable

EXPERIMENT	CLASSIFICATION	FAILING LINK	DETECTABILITY
Deployed Action Challenge255 Charge Is Not Numerically Priced	GAP	Direct formal claim	Undetectable
Final Rust MSM Evaluation Not Bound to Lean MSM Evaluation	GAP*	Rust to fixtures	Detectable
Post-hoc Fiat-Shamir Programming	GAP*	Fixtures to proof	Undetectable
Weak Fiat-Shamir: Instance is not Absorbed	NO_GAP	Fixtures to proof	Unresolved
Malicious Algebraic Verifier Selected After Fixing The Random Proofs	GAP*	Rust to fixtures	Detectable
Forgotten Output-Enable Gate	NO_GAP	Fixtures to proof	Undetectable
Omitted Private-Nullifier to Public-Instance Copy Constraint	NO_GAP	None	Undetectable
CommitIvk Authorization- Key Copy Omission	NO_GAP	Fixtures to proof	Undetectable
Keygen Omission of the enableSpend Copy	NO_GAP	Fixtures to proof	Undetectable
Existential ivk Binding in the Action Specification	NO_GAP	None	Undetectable

EXPERIMENT	CLASSIFICATION	FAILING LINK	DETECTABILITY
Buggy Lookups: Reuse of Beta at Every Gamma Use Site	NO_GAP	Fixtures to proof	Detectable
Identity Proof Points Rejected by Rust but Accepted by Lean	INFO	None	N/A
Verification-Key Gate-Hash Formalization Boundary	GAP* (resolved)	Rust to fixtures	N/A
Test Historical Bug: Multiopen Query-Collision	NO_GAP	Fixtures to proof	Detectable
Circuit Mutation Testing	NO_GAP	None	Undetectable

** These gaps are known boundaries of the security modeling, documented either in the code or the Ironwood book.*

Findings

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

INFORMATIONAL

- #00 Exponential Circuit Assignment to Witness Extractor
- #01 Deployed Action Challenge255 Charge Is Not Numerically Priced
- #02 Final Rust MSM Evaluation Not Bound to Lean MSM Evaluation
- #03 Post-hoc Fiat-Shamir Programming
- #04 Weak Fiat-Shamir: Instance is not Absorbed
- #05 Malicious Algebraic Verifier Selected After Fixing The Random Proofs
- #06 Forgotten Output-Enable Gate
- #07 Omitted Private-Nullifier to Public-Instance Copy Constraint
- #08 CommitIvk Authorization-Key Copy Omission
- #09 Keygen Omission of the enableSpend Copy

- #0a Existential ivk Binding in the Action Specification
- #0b Buggy Lookups: Reuse of Beta at Every Gamma Use Site
- #0c Identity Proof Points Rejected by Rust but Accepted by Lean
- #0d Verification-Key Gate-Hash Formalization Boundary
- #0e Test Historical Bug: Multiopen Query-Collision
- #0f Circuit Mutation Testing

Exponential Circuit Assignment to Witness Extractor

Classification. GAP (known boundary), fixtures to proof. Knowledge soundness. Undetectable. Ironwood pin `3c056cbe` .

Summary. The running time of the algorithm which recovers a witness (or break) from the action circuit does not have a limit on its running time. This means, that a broken circuit can be proven “knowledge sound” for the action relation:

- The production circuit is broken.
- The Lean modelling of the circuit is correct.
- The Lean modelling of the proof system is correct.

However, the proof of extraction, ignores the circuit assignment all together and instead either:

- Brute forces a witness (so that the “extract” event holds)
- Brute forces e.g. a sinsemilla collision (so that the “break” event holds)

It is worth observing that in the formalization, there are two extractors:

- One recovering a circuit witness from an (algebraic) prover.
- One recovering a witness for the spec from a circuit witness.

We target the latter extractor, concretely: the Action circuit computes the randomized verification key as $rk = [\alpha] \cdot \text{SpendAuthG} + ak_P$, where α is a witnessed scalar. We replaced the scalar multiplication with an unconstrained Pallas point:

```
#[cfg(feature = "unsafe-audit-spend-authority-point-witness")]  
let alpha_commitment = Point::new(  
    ecc_chip.clone(),  
    layouter.namespace(|| "witness spend authority point"),
```

```
self.spend_auth_point,  
)?;
```

The broken circuit only enforces $\mathbf{rk} = P + \mathbf{ak}_P$ for an arbitrary valid point P . A prover who holds a victim's viewing key but not the spending key can:

- Choose an independent RedPallas signing key, set $\mathbf{rk}$ to its verification key
- Compute $P = \mathbf{rk} - \mathbf{ak}_P$; without knowing α .

We then regenerate the fixtures and update the Lean proofs (without changing the specifications): instead we extract α by brute force, given a $\mathbf{rk}$, we write an algorithm in Lean that tries all possible values of α until it finds one that satisfies the equation, then we prove that this algorithm succeeds always. Another simple way to illustrate this gap would be to bruteforce a collision for Sinimilla, u and prove that one obtains a “break” instance via pigeonhole principle: the algorithm which hashes every $(\mathbf{input}_1, \mathbf{input}_2)$, eventually finds a collision.

Result. The issue is undetectable as the honest prover could know an α and hence be indistinguishably from a prover computing P as above. Observe also that a circuit with the recently mitigated issue (an unconstrained point), the extractor ignores the actually witnessed unconstrained point, instead it brute-forces a witness in the intended fixed point.

Recommendation. Model extractor efficiency part of the knowledge-soundness theorem. We (zkSecurity) are exploring similar efforts with [Caliper](#) which allows modelling fine-grained complexity, e.g. an extractor in 2^{64} “steps” and which may be of use; more high-level interfaces/operations on top of Caliper are forthcoming.

Deployed Action Challenge255 Charge Is Not Numerically Priced

Classification. GAP, direct formal claim. Adaptive knowledge soundness. Undetectable. Ironwood pin `3c056cbe`.

Summary. In Halo2, challenges (field elements) are generated by sampling 512 bits from the transcript and then reducing these modulo the field characteristic. This introduces an ever so slight bias in the challenge distribution: since 2^{512} is not a multiple of the field characteristic. The formalization analyzes soundness under uniformly random challenges, then adds a bias term to account for the fact that in one hybrid challenges are uniform, and in the other they have this slight bias. This term is $(q \cdot \varepsilon)$, where ε is the per-challenge bias and q bounds the number of relevant oracle queries. The formalization proves that $\varepsilon \leq 2^{-260}$. The problem is that, q is not formally tied to the intended query budget of the malicious prover: it is shown that the prover makes at most q queries, but any larger q also satisfies this requirement. Hence, we can set q freely without invalidating any of the “Capstone” theorems. For instance, setting q to at least 2^{300} makes $q \cdot \varepsilon > 1$ and the soundness is then vacuously true. This issue is solely a Lean issue on the modelling side: the version of the verifier where challenges are uniformly random is correctly modelled.

Recommendation. Simply upper bound q , we subsequently observed that the subsequent commits `f33db0e8` and `6dd377b6`, which postdate the pinned commit, do exactly this by adding `challengeQueryBound_le` :
`challengeQueryBound ≤ family.Q + (11 + k)` and pricing the charge at `2-136`.

Final Rust MSM Evaluation Not Bound to Lean MSM Evaluation

Classification. GAP (known boundary), Rust to fixtures. Knowledge soundness. Detectable. Ironwood pin `c4b59769` .

Summary. The fixtures check correspondance between the final MSMs for the Lean verifier and the Rust verifier. However, the formalization does not capture the correctness of the MSM computation on the Rust side: that requires reasoning about the Rust code (i.e. the cryptographic primitives) directly, which is outside the scope of the formalization. This experiment simply changes the Rust code such that it uses a vector of identity bases elements, this makes the Rust verifier accept all proofs and breaks no fixtures.

Result. This is a detectable GAP: since there is, by the Lean formalization, a sound verifier which distinguishes the valid proofs from the invalid ones: bringing the Rust verifier into agreement with the Lean verifier allows detecting invalid proofs in retrospect.

Recommendation. Assuming that the MSM behaves algebraically, this can be captured by e.g. including the Rust-side MSM evaluations (for random, invalid proofs) in the fixtures.

Post-hoc Fiat-Shamir Programming

Classification. GAP (known boundary), fixtures to proof. Adaptive knowledge soundness. Undetectable. Ironwood pin `3c056cbe` .

This experiment exercises a stronger adversarial model in which the proof-system designer chooses protocol description after fixing a random oracle (or concrete hash function). This is a documented boundary of the formalization.

Description. Because the random oracle is fixed *before* the proof system itself is specified, malicious proof system designers can create a proof system that is knowledge sound, but designed to accept a set of otherwise valid proofs: in security reductions, the random oracle has a distribution independent of the proof system, i.e. the proof system is fixed a priori and then the “random oracle is sampled”. However in real world deployments, the Fiat-Shamir transcript modeling is known to designers, prior to specifying the proof system, and if a *description of the proof system itself* is not used to domain separate the oracle, the designers can “program” the proof system to have “backdoored proofs” while still proving (correctly) that the proof system is adaptive-knowledge sound. In this experiment, we achieve this by adding a “shift” to one of the challenges: the challenge γ is replaced by $\gamma + \delta$ for a “proof system constant” δ . This clearly does not affect the distribution of γ , it simply permutes all the values, and we are able to formally prove that this variant of the proof system satisfies all the security properties. However, by picking a desired γ' , we can make a malicious proof be accepted, by setting $\delta = \gamma' - \gamma$ where γ is the actual challenge produced on the forged proof: we can program the proof system to accept this one invalid proof. The example here is a very simple variant of this type of “proof system programming”, more subtle versions may be possible, e.g. switching the order of terms in a random linear combination, *after* the challenges has been squeezed.

The experiment adds a compile-time scalar to γ immediately after its canonical transcript squeeze in both the audit prover and verifier:

```

let gamma: ChallengeGamma<_> = transcript.squeeze_challenge_scalar();
let gamma = *gamma;
#[cfg(feature = "unsafe-audit-post-hoc-gamma-shift")]
let gamma = gamma + super::post_hoc_gamma_shift::<C::Scalar>();

```

The Lean verifier performs the same executable addition before assembling the verifier equation:

```

let challenges := deriveChallenges fs init ps
let challenges := addFixedGammaOffset challenges
assemble vk instanceCommitment ps challenges

```

For a deterministic malicious proof prefix, the first run reveals β and the raw γ . The verifier constant is then selected as:

$$\begin{aligned}\delta &= -\beta \cdot \omega^{10} - \gamma, \\ \gamma' &= \gamma + \delta = -\beta \cdot \omega^{10}.\end{aligned}$$

Instance column 0 contains ten supplied rows, so row 10 is implicitly zero. Its identity and permutation labels are both ω^{10} . The source and target permutation factors at that row therefore become zero:

$$0 + \beta \cdot \omega^{10} + \gamma' = 0.$$

This allows us to forge a permutation check between the instance column values and the advice columns, in the proof-of-concept we use that to spend a note with zero value, as if it were a different note with a value of 50,000 ZEC.

Impact. This type of programming is undetectable, programmed proofs look exactly like real proofs. The potential impact from the existence of a programmed proof is inflation.

Recommendation. The formalization could be strengthened to reject this specific mutation, but doing so would not resolve the general random-oracle non-instantiability boundary for the deployed system. For future designs, consider e.g. computing a digest of a PDF which describes the scheme, then domain separating the ROM using that digest. To our knowledge, no deployed

system does this. There is also no evidence that the Halo2 proof system has been “programmed” in this way, in particular, the proof system specification appear very “natural”.

Weak Fiat-Shamir: Instance is not Absorbed

Classification. NO_GAP (`FIXTURES_T0_PROOF`). Adaptive knowledge soundness. Detectability unresolved. Ironwood pin `c4b59769` . This classification is conditioned on the assumption that there exists accepting proofs for false statement that exploit the Fiat-Shamir issue: see the caveat under **Result**.

Summary. As mandated by the Fiat-Shamir transformation, Halo2 absorbs the public-instance commitments into the transcript right after the verifying key hash and before reading the advice commitments. Behind the feature `unsafe-audit-omit-instance-from-transcript` , the absorption loop is removed out of both `create_proof` and `verify_proof` . Instance commitments are still computed from the public inputs, still enter the opening queries and the final MSM, and are still exported. This bug is known as “weak Fiat-Shamir”, as omitting statement data from Fiat-Shamir can allow challenges to be fixed before the public statement is fully bound.

Trusted component. The statement-bound transcript claim in `Zcash/Snark/Verifier/FiatShamir.lean` : `initialTranscript` contains the verifying-key representative and every configured instance commitment, `deriveChallengesForStatement` derives challenges from that prefix, and the four `nonInteractiveFingerprint_matches_derived` theorems are the statements of record.

Result. Under the mutant, all Fiat-Shamir challenges change and the fixture is therefore broken. Changing the Lean code to match the fixture will then make the theorem `instanceCommitment_eq_of_initialTranscript_eq` fail.

Malicious Algebraic Verifier Selected After Fixing The Random Proofs

Classification. GAP (known boundary). Detectable. Ironwood pin `3c056cbe`.

Summary. The random proofs used to test the verifier are two honest proofs and three random proofs. In particular, the random proofs are generated using some fixed seeds. This means that the Rust verifier could be modified to agree with the Lean verifier on those random proofs, while having an arbitrary behavior on other proofs. We target the first challenge of the inner product argument. Let's call c_0, c_1, c_2, c_3 the three concrete values of this challenge on the three random proofs. Let's also suppose that we want the modified verifier to accept one specific false, to simulate a backdoor scenario. Let's call r the value of the first challenge of the inner product argument on this false proof. The goal of the attacker is to make this challenge equal to zero for the false proof, while keeping it equal to the original values for the three random proofs. To do this, we can scale the challenge by the following polynomial:

$$S_r(X) = 1 - \frac{(X - c_0)(X - c_1)(X - c_2)(X - c_3)}{(r - c_0)(r - c_1)(r - c_2)(r - c_3)}$$

The mutant behaves the same as the canonical verifier on all four fixtures, since $S_r(c_i) = 1$, and maps the target's nonidentity residual to the identity, since $S_r(r) = 0$. Notice also that the rust verifier is still algebraic.

For this mutation, the concrete target false proof is the honest seed- `0x41` proof with an off-curve `rkY`. We can show how to construct a false proof which gets accepted by the mutated verifier. Since the generated fixtures are identical to the canonical verifier, everything on the Lean side still passes, and no change is needed.

Recommendation. One possible mitigation is to use a seed which is derived from the verifier's description or code hash. This would still enable some attacks similar to the ones described in [recent work](#) (intuitively, one can write code which references its own representation). However, to the best of our knowledge, those attacks are contrived, and could be easily ruled out by inspecting the verifier's code.

Forgotten Output-Enable Gate

Classification. NO_GAP, fixtures to proof. Knowledge soundness.

Undetectable. Ironwood pin `c4b59769`.

Summary. Check that if a gate is removed (made trivial), the formal verification catches this and we are unable to prove the “Capstone theorems”. We replaces the fourth polynomial of the first `Orchard circuit checks` gate with zero:

```
clean: q_orchard * v_new * (1 - enable_output)
NOP:   q_orchard * 0
```

The selector, the advice queries, witness generation, copies, lookups, and fixed values are unchanged. With the clause gone, a prover can present a nonzero output value `v_new = 1` with public `enable_output = 0`.

Trusted component. The verifying-key binding `vk_eq_toVerifierKey` in `Zcash/Snark/Keygen/Certificate.lean`, the closure `ConstraintSystem.closeWith0operations` that recomputes configured gates from synthesis activations, and `FormalCircuit.Soundness` against the unchanged Action `SpecPost`.

Result. The mutation is caught at three independent points. With a Rust-only mutation, the captured key differs from the Lean-derived key at flattened expression 3. Mirroring only the configuration but not synthesis makes `closeWith0operations` append the clean gate, giving 56 gate groups against Rust’s 55 (`configure_only_appends_clean_gate`). With a full update, `buggyOrchardGate` and a rewrite of every matching synthesis activation reproduce the same system of equations, we are unable to prove `FormalCircuit.Soundness` for the circuit. Exploitation would be undetectable, since a verifier deployed with the mutant key accepts both honest and exploiting proofs. This is fully covered by the formalization.

Omitted Private-Nullifier to Public-Instance Copy Constraint

Classification. NO_GAP. Knowledge soundness. Undetectable. Ironwood pin 3c056cbe .

Summary. The Action circuit derives the old nullifier storing the result in an advice cell. It then adds a copy constraint from that advice cell to the public instance cell:

```
layouter.constrain_instance(nf_old.inner().cell(), config.primary,  
NF_OLD)?;
```

This test investigates the following attack on the formalization: suppose we remove that copy constraint, and leave the rest of the circuit unchanged. Then, we modify the extraction function to read the private nullifier cell instead of the public instance cell. If the formalization was weaker than expected, we could still argue knowledge soundness: the extractor would extract the nullifier from the advice column, which however would not necessarily correspond to the instance value in the proof. This would be an undetectable and critical bug.

In our testing, we removed such copy constraint, and modified the corresponding lean circuit to match the circuit fingerprint. Then, we modified the `extract` function to read the private nullifier cell instead of the public instance cell. Since the `Spec` talks about the values extracted with `extract`, the circuit formalization still satisfied the local `FormalCircuit.Soundness` property.

The formalization fails instead one level up, in the integration with the verifier extractor. It is required that the local `extract` function, which reads the private nullifier cell, is equal to the canonical `combine` function, which reads the public instance cell, and so catches this kind of mutation.

We note that we stopped modifying the Lean code to modify the `extract` function, which seemed to be a reasonable trust boundary. One could modify also the verifier extraction, to write a malicious extractor that reads the private nullifier cell instead of the public instance cell, but it seemed to be a more contrived attack, which could be easily caught by code inspection. We did not pursue this attack further.

CommitIvk Authorization-Key Copy Omission

Classification. NO_GAP, fixtures to proof. Knowledge soundness.

Undetectable. Ironwood pin `3c056cbe` .

Summary. We omit a single copy-constraint, and check that we cannot prove soundness anymore `CommitIvk` receives the authorization key `ak` through a copy from the cell used by the spend-authority check. Behind the feature `unsafe-audit-commit-ivk-ak-copy-omission` , the copy is replaced with a witness assignment:

```
match ak_binding {
  AkBinding::Copy => {
    gate_cells.ak.copy_advice(|| "ak", &mut region, self.advices[0],
offset)?;
  }
  AkBinding::AssignForAudit => {
    region.assign_advice(|| "ak", self.advices[0], offset, ||
gate_cells.ak_witness)?;
  }
}
```

Result. All fixtures were regenerated from the changed Rust code and reflect the changed permutation check. The Lean mirror erases exactly the directed `constrainEqual` between the two `ak` cells. After these changes we were unable to prove knowledge soundness of the Action circuit. Exploitation would be undetectable, because an honest proof for the flawed circuit is indistinguishable from a malicious proof where the cells are different.

Keygen Omission of the enableSpend Copy

Classification. NO_GAP, fixtures to proof. Knowledge soundness.
Undetectable. Ironwood pin `3c056cbe` .

Summary. This experiment tests the following assumption: if the key generation function from the lean side is modified to be a malicious one, the formalization should reject it. Indeed, there are some theorems around the key generation that prove some consistency properties between the derived verifying key and the circuit's operations.

The idea of the attack would be the following: we modify the key generation function in Lean to omit a copy constraint in the circuit. The lean circuit and all the properties that rely on it are still satisfied, but the derived verifying key is not faithful to the circuit. Then, we could modify the Rust circuit to match the modified Lean circuit, and have a mismatch between the Lean circuit and the Rust circuit.

In this test, we did erase a copy constraint from the Action circuit key generation, which is the one that copies the `enableSpend` public instance to the advice column.

```
brokenCopyRaw := actionCopyRaw.erase targetRawCopy
```

Then we also modified the Rust circuit to match the modified Lean circuit, so that the Rust circuit does not have the copy constraint from the `enableSpend` public instance to the advice column. The rest of the circuit is unchanged, and the exported verifying key is checked to be consistent with the modified circuit.

The formalization correctly rejects the modified key generation. In particular, one of the consistency theorems that fails is `actionCopyLink`, which states that for every declared copy in the circuit, its two endpoints must belong to the same keygen permutation cycle. Modifying the keygen to omit a copy constraint breaks this property, and the theorem fails.

Existential ivk Binding in the Action Specification

Classification. NO_GAP. Knowledge soundness. Undetectable. Ironwood pin `3c056cbe`.

Summary. This experiment aims at testing the Action specification's existential quantification of the incoming viewing key. The Action specification quantifies the incoming viewing key existentially, `∃ ivk, ...`, and the diversified-address conjunct is guarded by a hash that may be undefined. The objective of this mutation is to test whether a circuit that binds `pkd0ld = [nk] gd0ld`, ignoring the `CommitIvk` output, would still satisfy the existential and be certified.

The mutated circuit still calls the `CommitIvk` gadget but feeds the already-witnessed `nk` cell into `ScalarVar::from_base`, changing exactly one copy-constraint argument:

```
let address_ivk = match address_ivk_source {
    AddressIvkSource::Committed => ivk.inner().clone(),
    AddressIvkSource::NullifierKey => nk,
};
let ivk = ScalarVar::from_base(ecc_chip.clone(), namespace, &address_ivk)?;
let (derived_pk_d_old, _) = g_d_old.mul(namespace, ivk)?;
```

The formalization correctly rejects the mutant circuit, in particular we are able to prove that the `diversified_address_is_sole_failed_conjunct` theorem holds, which states that the diversified-address conjunct is the only one that fails in the mutant circuit. The idea is that the `hashToPoint` function is defined, and the guarded commitment equation, the on-curve premise for `gd0ld`, and `fpScalarAction_injective` force the existential scalar to equal `nk`, while the exported committed x-coordinate is unequal to `nk`.

Buggy Lookups: Reuse of Beta at Every Gamma Use Site

Classification. NO_GAP, fixtures to proof. Adaptive knowledge soundness. Detectable. Ironwood pin `c4b59769`.

Summary. This experiment introduces a bug in the implementation of the permutation argument. The permutation argument uses two independently squeezed challenges `beta` and `gamma`. Behind the feature `unsafe-audit-verifier-beta-for-gamma`, the verifier still squeezes both but substitutes `beta` at every `gamma` use site, so the transcript schedule is unchanged and only the algebra collapses:

```
let beta: ChallengeBeta<_> = transcript.squeeze_challenge_scalar();
let sampled_gamma: ChallengeGamma<_> =
transcript.squeeze_challenge_scalar();
let gamma = if reuse_beta_for_gamma { beta.retag() } else { sampled_gamma
};
```

An audit-only prover mirrors the substitution so honest proofs still verify. The grand product becomes $\prod (v_i + \beta * \sigma_i + \beta) = \prod (v_i + \beta * id_i + \beta)$, which a cheating prover can satisfy with unequal cells for every `beta`. On a toy circuit with a three-cell copy cycle, the values `[a0, a2, a0 * a2 / a1]` with `a0 = 2`, `a1 = 1 + delta`, `a2 = 1 + delta^2` form such a witness: the clean `MockProver` rejects it, the audit prover produces a 1,248-byte proof, the mutant verifier accepts it, and the canonical verifier rejects it.

Trusted component. The verifier-fingerprint boundary and the permutation soundness argument: `deriveChallenges` and `assemble` consume `beta` and `gamma` independently, and the grand-product bridge theorem `multiset_pair_eq_of_prod_eval_eq` requires `gamma ∉ szBadSet (linProdDiff ...)` and $\forall j, \beta \notin \text{szBadSet}(\text{pairProdDiffCoeff } \dots j)$ as separate premises.

Result. The four Orchard captures under the mutant keep 22 squeezes with distinct sampled `gamma`, the verifying-key blocks and the random proof bytes are identical to the release fixtures, and only the accepting MSMs change, so `canonical_fingerprint_mismatch` holds for all four. The faithful mirror `assembleBetaForGamma` with `gamma := ch.beta` matches all four captures. The catch is in

`Zcash/Snark/Experiments/VerifierBetaForGamma/DiagonalWitness.lean` : on the diagonal `gamma = beta`, the concrete invalid witness violates at least one of the two bad-set obligations for every `beta`, so no independent-good-event package exists. The theorems `diagonal_pair_multisets_differ`, `diagonal_product_collision`, `not_reusedChallengePairSoundness`, `no_independent_good_challenge_package_on_diagonal`, and `reused_challenge_always_outside_independent_good_event` are kernel-checked with standard axioms; `native_decide` is confined to the eight concrete MSM comparisons.

The hypothesis that the reuse weakens Halo2 permutation soundness is confirmed for the mutant, but the gap hypothesis is not: the bad-set interface prices `beta` and `gamma` independently, so a proof-relevant collapse cannot silently retain the clean claim. Exploitation is detectable by canonical replay. The endpoint is scoped to the permutation pair-product kernel; no ledger capstone was instantiated at the toy circuit.

Identity Proof Points Rejected by Rust but Accepted by Lean

Classification. INFO. Ironwood pin `3c056cbe` .

Summary. The typed Lean proof string is documented in `Zcash/Snark/Core/ProofString.lean` as never branching on a proof element's value, with rejection of invalid encodings and points at infinity assigned to byte-level decoding outside the typed layer. In Rust, however, `EqAffine::from_bytes` decodes the all-zero encoding as the identity point, and `Blake2bRead::read_point` then rejects it during transcript absorption with the error “cannot write points at infinity to the transcript”, because `Transcript::common_point` requires affine coordinates:

```
let encoding = [0; 32];
let point = Option::
<EqAffine>::from(EqAffine::from_bytes(&encoding)).expect(...);
assert!(bool::from(point.is_identity()));
```

On the Lean side, the single-action honest fixture's `vanishingRandom` field was replaced by the group identity in an isolated module.

Trusted component. The typed post-decoding boundary between Halo2 bytes and Ironwood's `ProofString : proofStringWellFormed` and `assemble?` in `Zcash/Snark/Verifier/Assemble.lean`, and the documentation claim that `assemble?` models every deployed rejection path that affects control flow.

Result. Lean's `assemble?` accepts the identity-valued proof string and returns an MSM (`identityVanishingRandomAssembly_eq_some`), while the deployed reader rejects the same proof before verification. The missing predicate on the Lean side is `ps.vanishingRandom ≠ 0`; the well-formedness check only inspects the permutation `lastEval` shape before the algebraic guards. The direction is Lean-accepts, Rust-rejects, the inverse of what a gap requires, and the mutant MSM evaluates to a nonidentity point at

the captured challenges (`identityVanishingRandomMsm_eval_ne_zero`), so it is not an accepting proof either. All four fixture families are unaffected because none contains an identity point. The two docstring claims in `ProofString.lean` are, however, false at the pin.

The disputed behavior is present in the unmodified stack rather than injected, so the experiment falls under the rule for potential live defects. It is not a full completeness counterexample because no honest prover was shown to emit an identity commitment.

Recommendation. Add an identity predicate on proof points to `proofStringWellFormed` or `assemble?` so that the typed model returns `none` where the deployed reader rejects, and correct the docstrings. A later audit should separately inspect whether any other value-dependent branch in the verifier call graph has the unsafe polarity, where Rust accepts and the typed model rejects; that question is open.

Verification-Key Gate-Hash Formalization Boundary

Classification. GAP (known boundary). Adaptive chosen-key transcript binding. Ironwood pin `c4b59769`.

Summary. Ironwood's verifier takes the verifying-key hash as a parameter `vkHash : VerifyingKey -> F` rather than computing Halo2's concrete BLAKE2b construction over the pinned key. This gap can be exploited by a malicious protocol designer and adversary as follows:

1. introduce a collision in the verification-key hashing procedure. This can be done even when the hash is a random oracle, for example by using a bad serialization method or by omitting to hash certain fields of the verification key.
2. the adversary runs the honest prover for the honest circuit, collecting the transcript.
3. the adversary can modify the circuit to be trivially satisfied at challenges from the transcript while still having the same verification-key hash (using the collision from 1).

This experiment implements an example of such an attack behind the feature `unsafe-vk-omit-gates-from-hash`. The collision on `vk` hashes is introduced by making `PinnedGates::fmt` emit an empty list. As a consequence, two verification keys with different gates will have the same hash. An Orchard test freezes a single-action proof, reads a fixed polynomial evaluation `c = F(x)` after the challenge `x`, and builds a malicious circuit that is trivially satisfied at `x`:

```
original gate:  P(F)
malicious gate: P(F)(F - c)
```

With normal hashing, a circuit containing the malicious gate alters the challenge schedule and the frozen proof is rejected. With the introduced collision (gates omitted), the malicious circuit's hash collides with the original hash, the challenge vector and MSM are identical, and the frozen proof verifies.

Trusted component. The abstraction of the verifying-key hash, the fixed-canonical-key adaptive-statement theorem (which quantifies over one canonical key and claims no cross-key collision resistance), and `vk_eq_toVerifierKey`.

Result. Ironwood's generic verifier, instantiated with a constant `gateOmittingVkHash`, reproduces the collision: `omitted_gate_hash_collision`, `forged_gate_evaluations_match`, `forged_full_verifier_accepts`, and `static_full_verifier_rejects` all build in `Zcash/Snark/Experiments/VkHashGap/Experiment.lean`. A malicious proof can be constructed by modifying the honest witness to not satisfy the target gate.

At the audited pin, the identified boundary is that Ironwood proves the verifier algebra for a supplied key and hash but does not formalize Halo2's concrete serialization and BLAKE2b association; the concrete relation from `VerifyingKey` to `vkHash` is an external assumption. Although byte encoding, serialization, and BLAKE2b modeling were initially outside the scope and documented as model boundaries, the changes in [Ironwood PR #215](#) were agreed to be in scope and resolve this gap.

Recommendation. Recompute the verifying-key digest in Lean and bind the serialized key description to the key derived from the Action circuit. This has been implemented in the in-scope PR #215: `keyDigest` recomputes the digest by hashing the serialized description of the pinned verifying key exported by Rust, and `vk_eq_toVerifierKey` checks that the key matches the one derived from the Lean Action circuit. Cross-key collision resistance of the reduced digest remains an external assumption.

Test Historical Bug: Multiopen Query-Collision

Classification. NO_GAP, fixtures to proof. Knowledge soundness. Detectable. Ironwood pin `3c056cbe`.

Summary. We re-introduce a historical halo2 vulnerability and assert that the formal verification rejects the buggy proof system. Specifically, we re-introduce the [multiopen query-collision bug](#), fixed by halo2 PR #846. The bug is reintroduced as an off-by-default halo2 feature (`unsafe-audit-query-collision`) on top of the current pin, together with a narrow prover-side hook letting an audit-only “malicious prover” substitute a false claimed evaluation for one marked query.

We found the exploit does not reach any circuit gate: any gate/lookup/permutation-referenced evaluation is independently cross-checked by the vanishing argument’s aggregate `expected_h_eval`, itself opened via a structurally collision-immune query (an `MSM` object compared by pointer identity, never duplicated). This explains, at a mechanism level, why the original disclosure found “no known production circuit affected”. The working exploit instead uses a “dangling” query — an extra registration of an existing, gate-checked column at the wrapped rotation, created purely as the side effect of calling `query_advice`, consumed by no constraint at all. A real, gate-checked `plonk::create_proof / plonk::verify_proof` round trip on a minimal toy circuit accepts a proof forging that dangling query’s claimed value; the identical proof bytes are rejected by the default (fixed) build, confirming the bug is detectable by replay.

Trusted component. The SNARK-level claim under test is Ironwood’s generic verifier-acceptance predicate, `Zcash.Snark.DeployedAccepts` (built on `assemble?`, “the hypothesis every soundness endpoint consumes”) — not any Orchard-circuit-specific model. `assemble?` internally calls `constructIntermediateSets?`, guarded by `hasDuplicateCommitmentPoint`: a pure structural check over `(commId,`

point) pairs, computed from the typed `VerifyingKey` query layout and `ProofString` evaluations that Lean reconstructs itself, never from a Rust-exported grouping decision.

Result. A minimal, hand-built `Shape / VerifyingKey / ProofString` carrying the same two-query layout and claimed evaluations (5 real, 6 forged) as the real exploit was bound directly to Lean's `assembleQueries`. `hasDuplicateCommitmentPoint` fires on Lean's own, completely independent reconstruction of the collision, so `constructIntermediateSets? = none`, `assemble? = none`, and `DeployedAccepts`'s premise is `False` for this trace by definition, for every URS. The path is mutated Rust prover forges the dangling query's transcript value `-> plonk::verify_proof accepts (real bug) -> the identical typed data bound into Lean's assembleQueries -> hasDuplicateCommitmentPoint true -> assemble? = none -> DeployedAccepts false`. No mutation was mirrored into Lean to get this result: the model catches the historical bug class on its own, because it never trusts Rust's grouping decision in the first place.

This is a positive, NO_GAP result at the generic SNARK-verifier layer.

Circuit Mutation Testing

Classification. NO_GAP. Knowledge soundness. Undetectable.

Summary. We additionally injected flaws into the Orchard Action circuit and its Halo2 gadgets by removing or changing gate, copy, and lookup constraints. We used fixtures and circuit data from the Rust implementation to update the Lean model to match. Finally, we proved that the mutated circuits violate the unchanged specification, using `SpecPost` for the Action-level tests and the corresponding specification for the subcircuit tests. These results show that the intermediate specifications capture the tested circuit mutations correctly.

COMPONENT	MUTATION
Elliptic-Curve Multiplication	Removed the four copy constraints binding the internal multiplication base coordinates to the caller's input point.
Note-Value Range	Disabled the equations connecting the old and new note values to their bounded limb decompositions, while retaining the limb range checks.
Output-Enable Check	Changed the output-enable constraint to check the old note's value instead of the new note's value.
Action Gate Selector	Enabled the final Action gate on the following default-valued row instead of the row containing the assigned values.
Sinsemilla Basis Lookup	Removed the generator's y-coordinate from the lookup, while retaining the message-word and x-coordinate checks.

COMPONENT	MUTATION
Nullifier Addition	Replaced the constrained addition with a prover-supplied result, omitting the addition selector activation and both input-copy bindings.
Conditional Swap	Removed the constraint requiring the swap choice to be Boolean, while retaining both output equations.
CommitIvk Authorization-Key Canonicity	Disabled the three conditional checks enforcing the canonical encoding of the authorization-key field element, while retaining the decomposition and recombination checks.
Sinsemilla Accumulator	Disabled the constraint enforcing y-coordinate continuity between successive accumulator states.
Complete Elliptic- Curve Addition	Disabled the constraint requiring the output y-coordinate to be zero when the result represents the identity point.
Spend-Anchor Binding	Replaced the copy from the public anchor input with a private assignment to the same advice cell.

Result. All mutations resulted in our inability to prove the “Capstone theorems”, formally, we proved the negation of `SpecPost`. Proving the negation of the “Capstone Theorems” themselves requires proving the absence of an extractor, which is technically false because the set of extractors allowed by the formalization includes exponential time algorithms and with a polytime bounded extractor implies proving the hardness of dlog. All of the mutations were caught by the formalization. Any bugs here would be undetectable since the proof from an honest prover, which runs honest witness generation, and a malicious prover, which exploits the flaw, would be indistinguishable by zero-knowledge / witness-indistinguishability of Halo2.