certora

Security Assessment & Formal Verification



P-Token

May 2026

Prepared for Solana Foundation

certora

Table of content

Project Summary **4**
- Project Scope 4
- Project Overview 4
- Protocol Overview 4

Assessment Methodology **5**

Threat and Security Overview **6**
- Assets 6
- Actors 6
- Trust Assumptions 7
- Attack Vectors 7
- Findings Summary 9
- Severity Matrix 9

Detailed Findings **10**
- Informational Issues 12
- I-01. Batch can preserve stale return data from earlier child instructions 12
- I-02. Multisig can be initialized with m > n threshold 13
- I-03. inner_process_instruction dispatch comment is stale 14
- I-04. Multisig initialization allows duplicate signer keys 15
- I-05. Native account owner can indirectly clear close authority 16
- I-06. P-Token accepts malformed external-owned accounts rejected by SPL 17
- I-07. Malformed external-owned token/mint accounts trigger widespread SPL/P-Token error-ordering differences 18
- I-08. Revoke allows delegate self-revocation unlike SPL Token 19
- I-09. P-Token accepts Token-2022 multisig authorities rejected by SPL Token 20
- I-10. SyncNative uses different native reserve and amount update semantics than SPL Token 21
- I-11. UiAmountToAmount rejects long leading-zero inputs accepted by SPL Token 22
- I-12. Batch ownership pre-check for InitializeImmutableOwner creates a Batch equivalence violation. 23

Formal Verification **24**
- Verification Methodology 24
- Verification Notations 25
- General Assumptions and Simplifications 25
- Formal Verification Plan - Equivalence of SPL and P-Token Program 26
- Formal Verification Plan - Correctness of Batch Instruction 30
- Formal Verification Properties Overview 35
- Formal Verification Properties 37

Equivalence of SPL and P-Token Program 37
P-01. The SPL Token Program never panics 37
P-02. The P-Token Program never panics 37
P-03. If SPL Token Program returns Ok, then P-Token Program must also return Ok 38
P-04. Equivalence of the post-operation state when both SPL Token Program and P-Token Program return Ok 38
P-05a. If the P-Token Program returns a TokenError variant, then SPL Token Program must also return the same TokenError variant 39
P-05b. If the P-Token Program returns a ProgramError variant, then SPL Token Program must also return the same ProgramError variant 39
P-05c. If P-Token Program returns Err(InvalidInstructionData), then TokenInstruction::unpack must also return the same error 40
Correctness of Batch Instruction 41
P-06. Token Amount Plus Native Reserve Bounded by Lamports 41
P-07. Native Reserve Equals Rent-Exempt Minimum 41
P-08. Lamports at or Above Rent-Exempt Minimum 42
P-09. Account Ownership property 42
P-10. Single-to-Batch Equivalence 43
P-11. Batch-to-Single Equivalence 43
Disclaimer 44
About Certora 44

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
P-Token	https://github.com/solana-program/token	dae0830	Solana

Project Overview

This document describes the specification and verification of **P-Token** using the Certora Prover and manual code review findings. The work was undertaken from **March 30** to **May 25, 2026**.

The following program list is included in our scope:

solana-program/token/pinocchio/program/**
solana-program/token/program/**

The Certora Prover demonstrated that the implementation of the programs above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the programs. During the verification process and the manual audit, the Certora team discovered bugs in the Solana code, as listed on the following page.

Protocol Overview

P-Token is a Pinocchio-based implementation of the SPL Token program, intended to preserve SPL Token semantics while reducing compute overhead. It keeps the core token flows around mints, token accounts, authorities, delegation, and native SOL, while adding P-Token-specific instructions such as Batch, WithdrawExcessLamports, and UnwrapLamports. The implementation also introduces optimized fast paths for common instructions like Transfer and TransferChecked. The main users are token account owners, delegates, mint/freeze authorities, multisig signers, and external programs or clients that interact through the token interface.

Assessment Methodology

Our assessment approach combines design level analysis with a deep review of the implementation to ensure that a protocol is secure, economically sound, and behaves as intended under realistic conditions.

At the design level, we evaluate the architecture, the economic assumptions behind the protocol, and the safety properties that should hold independently of a specific chain or environment. This process includes reviewing internal and cross protocol interactions, state transition flows, trust boundaries, and any mechanism that could be exploited to extract value, deny service, or alter core system behavior. At this stage, a focused threat modelling exercise helps identify key attack surfaces and adversarial capabilities relevant to the system. Design level issues often relate to incentive structures, governance implications, or systemic behavior that emerges under adversarial conditions.

Implementation analysis focuses on the concrete behavior of the code within the execution model of the target chain. This involves reviewing the correctness of logic, access control, state handling, arithmetic behavior, and the nuanced behaviors of the chain environment. Familiar classes of vulnerabilities such as reentrancy conditions, faulty permission checks, precision issues, or unsafe assumptions often surface at this layer. These findings require context aware reasoning that takes into account both the code and the architectural intent.

To support this analysis, the codebase is examined through repeated manual passes and supplemented by automated tools when appropriate. High-risk logic areas receive deeper scrutiny, invariants are validated against both design intent and actual implementation, and potential vulnerability leads are thoroughly investigated. Automated techniques such as static analysis, fuzzing, or symbolic execution may be used to complement manual review and provide additional insight.

Collaboration with the development team plays an important role throughout the audit. This helps confirm expected behaviors, clarify design assumptions, and ensure an accurate understanding of the protocol's intended operation. All findings are documented with clear reasoning, reproducible examples, and actionable recommendations. A follow up review is conducted to validate the applied fixes and verify that no regressions or secondary issues have been introduced.

certora

Threat and Security Overview

P-Token's security depends on preserving the invariants enforced by SPL Token while using a CU-optimized implementation. The most important invariants are that balances cannot be created or destroyed outside mint, burn, transfer, native SOL sync, or unwrap flows; authorities can only act within their assigned permissions; and account state transitions remain consistent with Solana runtime ownership, signer, and writability rules. Since many integrations rely on SPL Token behavior as an implicit standard, even small differences in decoding, return data, authority precedence, or malformed-account handling can matter. The verification effort therefore treats SPL equivalence as core security property, alongside the usual token safety properties around supply, balances, delegation, multisig, and native SOL accounting. As a result, some informational findings document behavioral differences and equivalence violations between P-Token and SPL Token, even when they do not represent standalone security bugs.

The following lists summarize the main assets, actors, assumptions, and attack surfaces considered during the assessment. They are not exhaustive, but they capture the areas most relevant to P-Token's equivalence and safety model.

Assets

- **Token balances** - fungible balances stored in token accounts.
- **Mint supply**
- **Native SOL** - lamports backing wrapped SOL accounts.
- **Delegation approvals**
- **Authority rights** - owner, close, mint, freeze, and multisig control.
- **Account state** - initialized, frozen, native, and authority fields.
- **Return data** - observable outputs from read-only helpers and batch.

Actors

- **Account owners** - control token accounts and owner-authorized actions.
- **Delegates** - spend, burn, or unwrap within allowance.
- **Mint/freeze authorities**

- **Close authorities**
- **Multisig signers**
- **External programs** - invoke P-Token through CPI or own input accounts.

Trust Assumptions

- **SVM execution model** - signer checks, writable checks, owner restrictions, borrow rules, and post-execution writeback are enforced by the Agave/SVM runtime.
- **Account mapping** - verification assumes the configured SVM account-data mapping mode; behavior should be revalidated if direct mapping semantics change.
- **Fast-path layout** - serialized account offsets match Solana's VM ABI.
- **External data** - malformed token-shaped data requires non-token ownership.
- **Invocation boundaries** - return data clears only across runtime invocations.

Attack Vectors

- **Close authority side effects** - native account owner changes can clear close_authority, including same-owner updates.
- **Incorrect balance changes** - whether transfers, burns, minting, native SOL sync, unwrap flows, or malformed-account states could incorrectly create, destroy, or desynchronize token value, leading to fund loss
- **Delegate authority drift** - delegate-first authorization can make self-delegated owners act through delegated allowance rather than owner authority.
- **Delegate self-revocation** - P-Token allows the current delegate to clear its own delegation, which differs from SPL's owner-only revoke path.
- **Token-2022 multisig ownership** - accepting Token-2022-owned multisig authorities can authorize flows that classic SPL Token would reject.
- **Native SOL accounting drift** - SyncNative and UnwrapLamports edge cases can make token amount diverge from lamports or SPL behavior.
- **External-owned token-shaped accounts** - accounts not owned by Tokenkeg can be crafted with token-sized data and "challenge" decoding, ownership, and writeback boundaries.
- **Zero-amount mutation paths** - instructions with amount = 0 may skip writes that SPL still performs, changing writability and equivalence behavior.

- **Duplicate multisig signers** - repeated signer keys can let one signer satisfy multiple signer slots and reduce the effective threshold.
- **Unreachable multisig thresholds** - multisigs with m > n can be initialized but can never authorize governed actions.
- **Fast-path ABI dependence** - optimized transfer paths depend on Solana's serialized account layout, offsets, duplicate markers, and instruction-data placement.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	-	-	-
Low	-	-	-
Informational	12	12	12
Total	12	12	12

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

certora

Detailed Findings

ID	Title	Severity	Status
I-01	Batch can preserve stale return data from earlier child instructions	Info	Fixed
I-02	Multisig can be initialized with m > n threshold	Info	Fixed
I-03	inner_process_instruction dispatch comment is stale	Info	Fixed
I-04	Multisig initialization allows duplicate signer keys	Info	Fixed
I-05	Native account owner can indirectly clear close authority	Info	Fixed
I-06	P-Token accepts malformed external-owned accounts rejected by SPL	Info	Fixed
I-07	Malformed external-owned token/mint accounts trigger widespread SPL/P-Token error-ordering differences	Info	Fixed
I-08	Revoke allows delegate self-revocation unlike SPL Token	Info	Fixed
I-09	P-Token accepts Token-2022 multisig authorities rejected by SPL Token	Info	Fixed
I-10	SyncNative uses different native reserve and amount update semantics than SPL Token	Info	Fixed
I-11	UiAmountToAmount rejects long leading-zero inputs accepted by SPL Token	Info	Fixed

I-12	Batch ownership pre-check for InitializeImmutableOwner creates a Batch equivalence violation	Info	Fixed

Informational Issues

I-01. Batch can preserve stale return data from earlier child instructions

Description:
`process_batch` executes each child instruction by directly calling `inner_process_instruction`, so child instructions are not separated by a Solana runtime instruction boundary. As a result, return data is not cleared between batched children. If an earlier child instruction sets return data, such as `GetAccountDataSize`, `AmountToUiAmount`, or `UiAmountToAmount`, and a later child instruction does not set return data, the earlier value remains observable as the return data of the whole `Batch`.

For example, a batch containing `GetAccountDataSize` followed by `Transfer`, `Approve`, `Revoke`, `Burn`, or `SyncNative` would still expose the return data set by `GetAccountDataSize`, even though the final child instruction did not set return data.

Recommendation:
Either clear return data before each batched child instruction, or document that `Batch` returns the last return data set by any child instruction rather than the return data of the final child instruction.

Customer's response:
Acknowledged by documentation in fab19b4.

I-02. Multisig can be initialized with m > n threshold

Description:
`InitializeMultisig` accepts raw instructions where the required threshold m is greater than the number of provided signers n, such as a 5-of-3 multisig. The multisig account is initialized successfully because P-Token only checks that m and n are individually within the allowed signer range. Since authorization can only count signatures from the n configured signer slots, such a multisig can never satisfy its threshold and any authority assigned to it may become permanently unusable. SPL Token has the same on-chain behavior.

Recommendation:
Consider enforcing m ≤ n during multisig initialization, or document this as an SPL-compatible configuration footgun.

Customer's response:
Acknowledged by documentation in 19242f8.

I-03. inner_process_instruction dispatch comment is stale

Description:
The documentation comment above `inner_process_instruction` does not match the actual first-stage dispatch table. It omits several instructions handled directly by the first-stage matcher, including `Burn`, `TransferChecked`, `BurnChecked`, `SyncNative`, and `InitializeImmutableOwner`, and still lists `InitializeAccount2`, which actually falls through to `inner_process_remaining_instruction`.

Recommendation:
Consider updating the comment to match the current dispatcher.

Customer's response:
Fixed in commit fab19b4.

Fix Review: Fixed.

I-04. Multisig initialization allows duplicate signer keys

Description:
`InitializeMultisig` does not reject duplicate signer pubkeys. During authorization, signatures are counted by matched signer slots rather than by unique pubkey, so a configuration such as `m = 2`, `signers = [A, A, B]` can be authorized by A alone. This can make the effective signer threshold lower than the configured m, making the multisig easier to authorize than expected. SPL Token has the same on-chain behavior.

Recommendation:
Consider rejecting duplicate signer pubkeys during multisig initialization, or document this as an SPL-compatible configuration footgun.

Customer's response:
Acknowledged by documentation in 5a6c35b.

I-05. Native account owner can indirectly clear close authority

Description:
For token accounts with a `close_authority` set, `SetAuthority(CloseAccount)` requires the current close authority to sign, so the account owner cannot directly clear it unless they also control the close authority. However, for native token accounts, `SetAuthority(AccountOwner)` clears `close_authority` as a side effect. Since the implementation allows setting the owner to the current owner, a native account owner can call `SetAuthority(AccountOwner, current_owner)` and indirectly clear the close authority without the close authority signing. SPL Token has the same behavior.

Recommendation:
Document this SPL-compatible behavior, or reject same-owner `SetAuthority(AccountOwner)` calls for native accounts if the side effect is not intended.

Customer's response:
Fixed in commit fab19b4.

Fix Review: Fixed.

I-06. P-Token accepts malformed external-owned accounts rejected by SPL

Description:
In old SPL Token, instructions such as InitializeImmutableOwner, Approve{,Checked}, Revoke, and SetAuthority branches decode the provided account through the packed Account or Mint parser before continuing. That decoding validates the full packed layout, including COption-encoded fields such as delegate, close_authority, mint_authority, and freeze_authority. P-Token instead uses lighter zero-copy loads that check only the account length and initialized state, then read only the fields needed for the selected instruction path.

As a result, when the malformed bytes are placed in fields that the selected P-Token path does not depend on, and the instruction performs no net state change (no-op, so no program-ownership check from the runtime), old SPL Token rejects the account with InvalidAccountData while P-Token can return success. Examples include: InitializeImmutableOwner on an uninitialized (state = 0) external-owned 165-byte token-shaped account with malformed packed fields; Approve / ApproveChecked when the malformed external-owned source account already stores the same delegate and delegated amount being approved; Revoke when the malformed external-owned source account already has delegate = None (COption.0[0] = 0) and delegated_amount = 0; and no-op SetAuthority cases where the selected authority is already set to the requested value while the malformed COption bytes reside in another untouched packed field. This creates a success/failure equivalence difference between old SPL Token and P-Token.

Recommendation:
Consider documenting these as an accepted equivalence difference for malformed external-owned accounts. If mitigation at the code level is desired, consider adding full packed Account / Mint validation before instruction-specific logic in P-Token. However, in practice, this would need to be applied consistently across all instruction paths.

Customer's response:
Fixed in a043404.

Fix Review: Fixed.

I-07. Malformed external-owned token/mint accounts trigger widespread SPL/P-Token error-ordering differences

Description:
This issue concerns malformed external-owned token-sized or mint-sized accounts. When such an account is passed to one of the instruction families below, SPL Token typically rejects it immediately during full packed decoding with `InvalidAccountData`. P-Token often reaches later instruction logic first because it uses optimized zero-copy loads that do not validate the full packed layout up front. As a result, the same call can still revert in both implementations, but with a different observed error. Depending on the instruction and state, P-Token may instead return a later semantic error such as `AlreadyInUse`, `NotRentExempt`, `IncorrectProgramId`, or fail only when the runtime enforces write restrictions on the external-owned account.

This mismatch pattern applies to `InitializeMint{,2}` (malformed external-owned mint account), `InitializeAccount{,2,3}` (malformed external-owned new account), `MintTo{,Checked}` (malformed external-owned destination account and mint account), `Burn{,Checked}` (on a malformed external-owned source token account and mint account), `Transfer{,Checked}` including self-transfer variants (on a malformed external-owned source/destination token account and mint account), `FreezeAccount`, `ThawAccount`, and `CloseAccount` (on a malformed external-owned source token account).

Recommendation:
Document that exact SPL/P-Token error equivalence for these instruction families assumes canonical packed token-account and mint-account state on the relevant inputs.

Customer's response:
Fixed in a043404.

Fix Review: Fixed.

I-08. Revoke allows delegate self-revocation unlike SPL Token

Description:
`Revoke` authorizes the instruction differently in P-Token and SPL Token. SPL Token always validates the token account owner as the authority before clearing `delegate` and `delegated_amount`. P-Token instead checks whether the provided authority is the current delegate, and if so, validates the delegate as the authority; otherwise it falls back to the account owner.

As a result, the current `delegate` can revoke its own delegation in P-Token without the token account owner signing. The same instruction and accounts would fail in SPL Token unless the owner authorizes the revoke. This creates a success/failure equivalence difference between P-Token and SPL Token.

Recommendation:
Consider documenting delegate self-revocation as an intentional P-Token behavior difference.

Customer's response:
Fixed in 903fa91.

Fix Review: Fixed.

I-09. P-Token accepts Token-2022 multisig authorities rejected by SPL Token

Description:
P-Token and SPL Token differ when validating multisig authorities. SPL Token treats an authority account as a multisig only if it is owned by the classic SPL Token program. P-Token's `validate_owner` accepts multisig accounts owned by either P-Token/classic Tokenkeg or Token-2022.

As a result, an instruction authorized by a Token-2022-owned multisig can succeed in P-Token but fail in SPL Token with MissingRequiredSignature. This creates a success/failure equivalence difference for any instruction path that accepts multisig authorities, such as transfers, burns, minting, authority changes, and account closing.

Recommendation:
Document Token-2022 multisig authority support as an intentional P-Token extension.

Customer's response:
Fixed in 33be5f8.

Fix Review: Fixed.

I-10. SyncNative uses different native reserve and amount update semantics than SPL Token

Description:
`SyncNative` handles native token accounts differently in P-Token and SPL Token. SPL Token reads the rent-exempt reserve stored in the account's `is_native` field, computes `new_amount = lamports - stored_reserve`, and rejects with `InvalidState` if the new amount would be lower than the current token amount. P-Token instead recomputes the rent-exempt reserve from `Rent::get()` or from an optional `Rent` sysvar account, writes the recalculated reserve back into the account, and overwrites the token amount without SPL Token's amount-decrease guard.

There is also an account-list handling difference. SPL Token ignores any extra accounts passed to `SyncNative`, while P-Token treats the first remaining account as an optional `Rent` sysvar and validates its key. Therefore, a `SyncNative` call with an extra non-Rent account can succeed or continue normally in SPL Token, but fail in P-Token with `InvalidArgument`. This violation is present compared against 4d5ff30.

Recommendation:
Consider documenting the new P-Token SyncNative semantics as intentional.

Customer's response:
Fixed in 48164e9.

Fix Review: Fixed.

I-11. UiAmountToAmount rejects long leading-zero inputs accepted by SPL Token

Description:
`UiAmountToAmount` is stricter in P-Token than in SPL Token for long leading-zero UI amount strings. SPL Token validates the string format and then parses the resulting decimal string as `u64`, without imposing a fixed pre-parse length bound. As a result, for a valid initialized mint with `decimals = 0`, a standalone instruction with data `[24] || b"0".repeat(258)` is accepted by SPL Token and returns 0.

P-Token uses a fixed-size formatted digit buffer and rejects inputs where `amount_str.len() + decimals > 257` before parsing. For the same 258-byte leading-zero amount string, P-Token returns `InvalidArgument`. This creates a success/failure equivalence difference for an unrealistic but valid standalone SPL Token input.

Recommendation:
Document this as an accepted parser strictness difference, or match SPL Token behavior by allowing leading-zero strings that parse to a valid `u64`.

Customer's response:
Fixed in 1093baa.

Fix Review: Fixed.

I-12. Batch ownership pre-check for InitializeImmutableOwner creates a Batch equivalence violation

Description:
`process_batch` applies an explicit `check_account_owner` pre-check for `InitializeImmutableOwner` (discriminator 22) before dispatching the instruction. This pre-check is present because the SVM runtime ownership enforcement, which normally rejects writes to accounts not owned by the executing program, does not fire between child instructions inside a batch, only at the end of the full transaction.

However, `process_initialize_immutable_owner` is a stub that writes nothing to the account. For any uninitialized account whose data length satisfies the Account layout check, it returns `Ok(())` without modifying any account data. As a result, an uninitialized account not owned by the token program causes the two execution contexts to diverge:

Batch context: `check_account_owner(a0)` is called first and fails with IncorrectProgramId.
Single instruction context: `process_initialize_immutable_owner` returns `Ok(())`, no write occurs, and the runtime ownership check is not triggered so the instruction succeeds.

Recommendation:
Remove discriminator 22 (`InitializeImmutableOwner`) from the ownership pre-check match arm in `process_batch`. Alternatively, one can add an explicit `check_account_owner` inside `process_initialize_immutable_owner` itself, so that the ownership requirement is enforced uniformly in both contexts.

Customer's response:
Acknowledged by documentation in 93a95aa.





Formal Verification

Verification Methodology

We performed verification of the **P-Token** protocol using the Certora Prover which is based on Satisfiability Modulo Theories (SMT).
In short, the Certora Prover works by compiling formal specifications written in the Certora Verification Language (CVLR) and P-Token's implementation source code written in Rust.
More information about Certora's tooling can be found in the Certora Technology Whitepaper.

If a property is verified with this methodology it means the specification in CVLR holds for all possible inputs. However specifications must introduce assumptions to rule out situations which are impossible in realistic scenarios (e.g. to specify the valid range for an input parameter). Additionally, SMT-based verification is notoriously computationally difficult. As a result, we introduce overapproximations (replacing real computations with broader ranges of values) and underapproximations (replacing real computations with fewer values) to make verification feasible.

Rules: A rule is a verification task possibly containing assumptions, calls to the relevant functionality that is symbolically executed and assertions that are verified on any resulting states from the computation.

Inductive Invariants: Inductive invariants are proved by induction on the structure of a smart contract. We use constructors/initialization functionality as a base case, and consider all other (relevant) externally callable functions that can change the storage as step cases.
Specifically, to prove the base case, we show that a property holds in any resulting state after a symbolic call to the respective initialization function. For proving step cases, we generally assume a state where the invariant holds (induction hypothesis), symbolically execute the functionality under investigation, and prove that after this computation any resulting state satisfies the invariant. Each such case results in one rule.

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue
Violated	A counterexample exists that violates one of the assertions of the rule.

General Assumptions and Simplifications

1. Configuration:
    a. Solana Platform Tools - v1.53.
    b. Loops are unrolled at most 3 times.

Formal Verification Plan - Equivalence of SPL and P-Token Program

Any execution of either the SPL Token Program or the P-Token Program terminates in one of three outcomes: a panic, a return of `Ok(())`, or a return of an `Err`. The verification plan compares the behavior of the two implementations across each of these three outcome categories independently. We first describe the execution model for both SPL and the P-Token Programs.

Execution Model and Proof Scope
P-Token is executed starting at the per-instruction processor function level: `process_mint_to`, `process_revoke`, etc.

SPL is executed as the following two-step sequence:

```rust
let args = TokenInstruction::unpack(instruction_data);
let res = process_fn_name(args);
```

Both sides receive nondeterministic account_infos and nondeterministic instruction data, so no assumption is made about the shape of inputs beyond the per-property preconditions.

This formulation differs from SPL's top-level process function only in the `match instruction { ... }` dispatch block inside `process`. That block was reviewed manually and found to perform no state changes, it solely routes to the appropriate sub-processor. Similarly, all code paths from P-Token's `entrypoint` down to each `process_fn_name` were reviewed manually, including the fast paths for `transfer` and `transfer_checked`.

Equivalence Properties
Let us now compare the behavior of the SPL and P-Token programs across the three categories of behaviors.

No-Panic Properties:
We establish that both SPL and P-Token Programs never panic with the following properties:.

([Property-1](#)) The SPL Token Program never panics.

(Property-2) The P-Token Program never panics.

Ok Behavior — Success Implication:
P-Token includes additional features not present in SPL Token, for example allowing a delegate to revoke their own delegation in `Revoke` (I-9), and accepting the Token-2022 program as a valid multisig owner in `validate_owner` (I-10). As a result, there exist inputs for which P-Token returns `Ok(())` while SPL does not. The implication is therefore formulated in one direction only:

(Property-3) If the SPL Token Program returns `Ok(()),` then the P-Token Program must also return `Ok(()).`

Ok Behavior — Post-State Equivalence:
When both programs succeed, the resulting account state must agree. This is captured by:

(Property-4) Starting from identical input account_infos, if both the SPL Token Program and P-Token return `Ok(()),` then their output account_infos must match in the post-operation state.

Error Behavior — Error Propagation
By the same reasoning as above — P-Token accepting strictly more inputs than SPL — the error equivalence is stated as:

(Property-5) Starting from identical input account_infos, if P-Token returns `Err(e)` for error variant `e`, then the SPL Token Program must also return `Err(e)`.

For ease of verification, Property-5 is decomposed on the basis of the error variant:

(Property-5a) If P-Token returns a `TokenError` variant, SPL must return the same `TokenError` variant.
(Property-5b) If P-Token returns a `ProgramError` variant, SPL must return the same `ProgramError` variant.
(Property-5c) If P-Token returns `Err(InvalidInstructionData)`, then `TokenInstruction::unpack` applied to the same instruction data must also return the same error.

Property-5c provides completeness: when P-Token rejects data as malformed at the instruction-decoding level, the SPL unpacking step must agree.

Instruction Coverage

The above properties are verified for all instruction handlers shared by both the SPL Token Program and the P-Token Program. Four instructions require case-splitting for the Certora Prover: `Transfer`/`TransferChecked` are split on whether source and destination are the same account, `SyncNative` is split on the basis of how `Rent` is fetched and `SetAuthority` is split on whether the target is a mint account or a token account. This yields the following 27 verification cases:

1. `MintTo`
2. `MintToChecked`
3. `Burn`
4. `BurnChecked`
5. `Transfer` (source ≠ destination)
6. `TransferChecked` (source ≠ destination)
7. `Transfer` — self-transfer (source == destination)
8. `TransferChecked` — self-transfer (source == destination)
9. `Approve`
10. `ApproveChecked`
11. `InitializeMint` (rent sysvar passed as account)
12. `InitializeMint2` (rent obtained via syscall)
13. `InitializeAccount` (rent sysvar passed as account)
14. `InitializeAccount2` (owner from instruction data; rent sysvar as account)
15. `InitializeAccount3` (owner from instruction data; rent via syscall)
16. `InitializeMultisig` (rent sysvar passed as account)
17. `InitializeMultisig2` (rent via syscall)
18. `FreezeAccount`
19. `ThawAccount`
20. `Revoke`
21. `CloseAccount`
22. `SyncNative` (rent via syscall)
23. `SyncNative` (rent sysvar passed as account)

24. `GetAccountDataSize`
25. `InitializeImmutableOwner`
26. `SetAuthority` on a `Mint` account (account data length = `Mint::LEN` i.e. 82 bytes)
27. `SetAuthority` on a `Token` account (account data length = `Account::LEN` i.e. 165 bytes)

Verifying above properties for each instruction forms a step case in the inductive proof of equivalence.

Formal Verification Plan - Correctness of Batch Instruction

Execution Contexts and Proof Strategy

The `batch` instruction in P-Token executes a sequence of child instructions within a single SVM invocation by directly calling `inner_process_instruction` for each entry in the batch, without crossing a runtime invocation boundary between them. When both execution contexts return `Ok(())`, byte-by-byte equality of the resulting account state is justified by construction: both the single-ix context and the batch context ultimately call the same underlying processor function (`inner_process_instruction` dispatches to the same `process_IX` implementation in both cases). Any account mutations performed by the instruction are therefore identical regardless of whether the instruction was invoked standalone or through `process_batch`. The non-trivial aspect of batch correctness is the runtime ownership checks: in standalone execution the SVM runtime enforces ownership constraints after each instruction, whereas inside a batch those checks are absent between child instructions and must be compensated for by P-Token's own pre-dispatch ownership checks in `process_batch`. The equivalence properties therefore focus exclusively on whether the combined result of the processor call and the runtime ownership check is a success or a failure, which is where the two execution contexts can meaningfully diverge.

We frame the correctness question as: does executing an instruction through the Batch mechanism preserve the same pass/fail semantics as executing it directly as a standalone instruction?

Certora's verification approach is to prove the following equivalence: "*An instruction succeeds in the single-ix ("single-instruction") context if and only if it succeeds in the batch context*".

For ease of verification, we prove the contrapositive formulation: an instruction fails in the single-ix context if and only if it fails in the batch context. The two directions of this biconditional are proved separately as Single-to-Batch Equivalence (Property-10) and Batch-to-Single Equivalence (Property-11).

In the following sections, we introduce definitions of "runtime checks" and "interference" that are simplified compared to the actual definitions. This is done for ease of exposition.

Failure Model

An instruction can fail in two distinct ways:

1. Program-level failure. The process function returns an `Err(_)` variant. This is caught by the Batch loop and causes it to propagate the error.
2. Runtime failure. The process function returns `Ok(())`, but the SVM runtime rejects the transaction after execution due to a post-execution check.

The SVM runtime applies two post-execution checks to every instruction:

a. Minimum rent balance: Every AccountInfo referenced by the instruction must hold at least the rent-exempt minimum lamports for its data size.
b. Account ownership: Every AccountInfo that was written to during execution must be owned by the executing program (the P-Token program in our case).

Simplifying the Runtime Model via Lamport Invariants

Rather than encoding the full rent check symbolically in each equivalence rule, we first establish as a separate inductive invariant that the P-Token program always preserves lamports at or above the rent-exempt minimum for every account it operates on. This is captured by three supporting invariants:

(Property-6) For all native token accounts, `account.amount + account.native_amount ≤ account_info.lamports`.

(Property-7) For all native token accounts, `account.native_amount` equals the rent-exempt minimum lamports for the account's data size.

(Property-8) For all token and mint accounts operated on by the P-Token program, `account_info.lamports ≥ rent_minimum_lamports(account_info.data_len)`.

Properties 6 and 7 are both required in the inductive hypothesis to establish Property 8. Property 6 ensures that the token amount is bounded above by the available lamports, and Property 7 pins the native reserve to the computed rent minimum, together making it possible to conclude that the lamport floor invariant is maintained through every instruction that adjusts lamports.

With Property 8 established, the rent check cannot distinguish the single-ix context from the batch context since both contexts operate on accounts that already satisfy the rent constraint. The runtime model in the equivalence properties is therefore reduced to the account ownership check alone:

```rust
runtime_ownership_check(info) := if cvlr_is_account_written(info) {
     info.owner == TOKEN_PROGRAM_ID
} else {
     true
}
```

Here, `cvlr_is_account_written` is a boolean construct provided by the Certora Prover to track whether an account was written to. An instruction execution is considered successful in a given context if and only if the process call returns `Ok(())` and `runtime_ownership_check` holds for every account in scope.

Interference Model
When an instruction runs inside a Batch, other child instructions in the same batch may execute before or after it, potentially modifying shared accounts. To account for this without modeling the full instruction sequence, we introduce a nondeterministic interference step between the target instruction and the runtime check on the batch side.

The interference model permits an arbitrary overwrite of the full in-memory layout of each AccountInfo (including lamports and data, for instance) subject to a single constraint on the owner field: "*Owner constraint. After interference, account_info.owner is either unchanged from its pre-interference value, or is set to the all-zeros pubkey [0; 32].*"

The motivation for this constraint is that the actual P-Token instructions satisfy precisely this invariant: they never set an account's owner to an arbitrary value; they either leave it unchanged or zero it out via `CloseAccount`'s `close_unchecked` path. This is established formally by the following property**:**

(Property-9) Account Ownership Property: For every supported P-Token instruction and for every AccountInfo in the instruction's account array, after execution the account_info.owner field is either unchanged from its pre-execution value, or equal to the all-zeros pubkey `[0; 32]`.

This property is proved for every P-Token instruction in scope. The only instruction that legitimately changes an account owner is `CloseAccount`. Finally, this property provides the formal foundation that the interference model is not an overapproximation of what real P-Token batch executions can do.

Single-to-Batch Equivalence

The single-ix context for a given instruction IX is defined as:

1. Call the direct processor `process_IX(accounts, data)`.
2. Apply `runtime_ownership_check` to every account.

The instruction is considered to succeed in the single-ix context if both steps succeed.

The batch context is defined as:

1. Call `process_batch(accounts_copy, batch_ix_data)`, where `accounts_copy` shares the same underlying memory as `accounts` at call time.
2. Apply nondeterministic interference to every element of `accounts_copy`.
3. Apply `runtime_ownership_check` to every element of `accounts_copy`.

The instruction is considered to succeed in the batch context if all three steps succeed.

(Property-10) Single-to-Batch Equivalence: For every supported P-Token instruction, if the instruction fails in the single-ix context (either the process call returns `Err(_)` or a runtime ownership check fails), then the instruction also fails in the batch context (either process_batch returns `Err(_)` or a runtime ownership check fails after interference).

The interference step is included on the batch side to model the effect of other instructions in the same Batch. The key insight is that even though interference can arbitrarily modify account state, the owner constraint ensures that if the runtime ownership check fails after the direct call, no feasible interference can repair it. This is because interference can only zero the owner, not set it to the token program ID from a non-token-program owner.

Batch-to-Single Equivalence

(Property-11) Batch-to-Single Equivalence: For every supported P-Token instruction, if the instruction fails in the batch context (either process_batch returns `Err(_)` or a runtime ownership check fails), then the instruction also fails in the single-ix context.

Critically, interference is not applied on the batch side in this direction. If interference were included, the property would be trivially falsifiable: a batch call could succeed, interference could then zero out an account's owner, and the runtime check would fail. Yet, the single-ix execution which does not experience interference would pass. Omitting interference from the batch side is therefore necessary to capture the meaningful semantic claim that `process_batch` preserves the failure conditions of the underlying instruction.

Property-11 produces a counterexample for `InitializeImmutableOwner` instruction due to I-12.

Instruction Coverage
Properties 6-11 are verified for all P-Token instructions except `UiAmountToAmount`, `AmountToUiAmount` and `Batch` (which receives special treatment).

Formal Verification Properties Overview

ID	Title	Impact	Status
P-01	The SPL Token Program never panics.	Ensures stability by preventing unexpected program termination.	Verified
P-02	The P-Token Program never panics.	Ensures stability by preventing unexpected program termination.	Verified
P-03	If SPL Token Program returns Ok, then P-Token Program must also return Ok.	Ensures P-Token supports all successful execution paths of SPL Token.	Verified
P-04	Equivalence of the post-operation state when both SPL Token Program and P-Token Program return Ok.	Ensures consistency of account state when both programs succeed.	Verified
P-05a	If the P-Token Program returns a TokenError variant, then SPL Token Program must also return the same TokenError variant.	Ensures equivalence in error propagation for TokenError variants.	Verified
P-05b	If the P-Token Program returns a ProgramError variant, then SPL Token Program must also return the same ProgramError variant.	Ensures equivalence in error propagation for ProgramError variants.	Verified
P-05c	If P-Token Program returns	Ensures agreement on instruction decoding when data is malformed.	Verified

	Err(InvalidInstructionData), then TokenInstruction::unpack must also return the same error.		
P-06	Token Amount Plus Native Reserve Bounded by Lamports	Ensures that the token amount is bounded above by the available lamports for native token accounts.	Verified
P-07	Native Reserve Equals Rent-Exempt Minimum	Ensures the native reserve is pinned to the computed rent minimum for native token accounts.	Verified
P-08	Lamports at or Above Rent-Exempt Minimum	Ensures every account's lamports remain at or above the rent-exempt minimum.	Verified
P-09	Account Ownership property	Ensures every AccountInfo's owner field is either identical to its pre-execution value or equal to [0; 32]	Verified
P-10	Single-to-Batch Equivalence	Verifies that failure of a single-instruction context implies failure in the batch context.	Verified
P-11	Batch-to-Single Equivalence	Verifies that failure in the batch context implies failure in the single-instruction context.	Violated

Formal Verification Properties

Equivalence of SPL and P-Token Program

Module General Assumptions

- In `validate_owner`, we disable the multisig validation case due to loops. This code path was verified manually.
- In P-05a and P-05b, we assume for P-Token `process_revoke` that `source_account.delegate` is not set. This aligns the behavior of revoke in SPL and P-Token programs as per the client's expectation. Without this assumption, the Certora Prover produces a counterexample related to issue I-08.
- In P-05a and P-05b, we assume for all instructions that `COption` tag bytes for token account and mint are either 0 or 1. Without these assumptions, the Certora Prover produces counterexamples related to issues I-06 and I-07.

Module Properties

P-01. The SPL Token Program never panics.

Status: Verified

Rule Name	Status	Description	Link to rule report
spl_panic_free_{step_case}	Verified	*These rules verify that the SPL process functions never panic. Verified for all 27 step cases listed above.*	*Rule report*

P-02. The P-Token Program never panics.

Status: Verified

Rule Name	Status	Description	Link to rule report
ptoken_panic_free_{step_case}	Verified	*These rules verify that the P-Token process functions never panic. Verified for all 27 step cases listed above.*	*Rule report*

P-03. If SPL Token Program returns Ok, then P-Token Program must also return Ok.

Status: Verified

Rule Name	Status	Description	Link to rule report
equiv_spl_ok_pino_ok_{step_case}	Verified	*These rules verify that If SPL Token Program returns Ok, then P-Token Program must also return Ok. Verified for all 27 step cases listed above.*	*Rule report*

P-04. Equivalence of the post-operation state when both SPL Token Program and P-Token Program return Ok.

Status: Verified

Rule Name	Status	Description	Link to rule report
equiv_{step_case}	Verified	*These rules verify byte-by-byte equivalence of post-operation account_infos and account data when both SPL Token Program and P-Token Program return Ok. Verified for all 27 step cases listed above.*	*Rule report*

P-05a. If the P-Token Program returns a TokenError variant, then SPL Token Program must also return the same TokenError variant.

Status: Verified	Assumptions: (i) We assume relevant COption tag bytes are either 0 or 1. (ii) For mint, burn and transfer, we assume account.amount is less than equal to account_info.lamports.

Rule Name	Status	Description	Link to rule report
equiv_errors_{step_case}_tok_err	Verified	*These rules verify that if a P-Token process function returns a TokenError variant, then SPL process function also returns the same variant. Verified for all 27 step cases listed above.*	*Rule report*

P-05b. If the P-Token Program returns a ProgramError variant, then SPL Token Program must also return the same ProgramError variant.

Status: Verified	Assumptions: (i) We assume relevant COption tag bytes are either 0 or 1. (ii) For mint, burn and transfer, we assume account.amount is less than equal to account_info.lamports.

Rule Name	Status	Description	Link to rule report
equiv_errors_{step_case}_prog_err	Verified	*These rules verify that if a P-Token process function returns a ProgramError variant, then SPL process function also returns the same variant. Verified for all 27 step cases listed above.*	*Rule report*

P-05c. If P-Token Program returns Err(InvalidInstructionData), then TokenInstruction::unpack must also return the same error.

Status: Verified

Rule Name	Status	Description	Link to rule report
equiv_errors_ix_data_{step_case}	Verified	*These rules verify that if a P-Token process function returns Err(InvalidInstructionData), then TokenInstruction::unpack also returns the same error. Verified for all 27 step cases listed above.*	*Rule report*

Correctness of Batch Instruction

Module General Assumptions

- In `validate_owner`, we disable the multisig validation case due to loops. This code path was verified manually.

Module Properties

P-06. Token Amount Plus Native Reserve Bounded by Lamports

Status: Verified

Rule Name	Status	Description	Link to rule report
amt_leq_lamports_{step_case}	Verified	*These rules verify that for native token accounts, account.native_amount equals the computed rent-exempt minimum.*	*Rule report*

P-07. Native Reserve Equals Rent-Exempt Minimum

Status: Verified

Rule Name	Status	Description	Link to rule report
native_amount_eq_rent_min_{step_case}	Verified	*These rules verify that for native token accounts account.amount + account.native_amount ≤ account_info.lamports.*	*Rule report*

P-08. Lamports at or Above Rent-Exempt Minimum

Status: Verified

Rule Name	Status	Description	Link to rule report
lamports_geq_rent_min_{step_case}	Verified	*These rules verify that every token and mint account's lamports remain at or above rent-exempt minimum.*	*Rule report*

P-09. Account Ownership property

Status: Verified

Rule Name	Status	Description	Link to rule report
account_ownership_{step_case}	Verified	*These rules verify that every AccountInfo's owner field is either identical to its pre-execution value or equal to [0; 32]*	*Rule report*

P-10. Single-to-Batch Equivalence

Status: Verified

Rule Name	Status	Description	Link to rule report
single_to_batch_equiv_{step_case}	Verified	*These rules verify that failure of the direct processor call or post-execution runtime ownership check implies failure of process_batch or post-interference runtime ownership check.*	*Rule report*

P-11. Batch-to-Single Equivalence

Status: Violated

Rule Name	Status	Description	Link to rule report
batch_to_single_equiv_{step_case}	Violated	*These rules verify that failure of process_batch or post-execution runtime ownership check implies failure of the direct processor call or its runtime ownership check. No interference is applied on the batch side. Violation due to I-12.*	*Rule report*

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora 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.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline and is widely used by developers and security researchers during audits and bug bounties.

Certora also provides architectural design reviews, where researchers analyze system design and trust boundaries early to identify structural risks and reduce complexity before implementation.

Certora conducts off-chain audits covering backend services, APIs, frontend, and mobile applications, focusing on vulnerabilities in authentication, data handling, key management, and wallet-related workflows.

Certora also offers Penetration Testing and Red Teaming to simulate real-world attacks and uncover exploitable weaknesses across infrastructure, applications, and business logic.

In addition, Certora provides operational security (OpSec) assessments, reviewing key management, access controls, and operational processes to strengthen overall security posture.

Finally, Certora delivers ongoing monitoring and incident response, enabling continuous threat detection and prevention, alert triage, and coordinated response to active incidents.