Checkpointability: Separating Proof Availability from Publication Liveness

Jonathan Oleszkiewicz
BaaS.sh
France
jonathanolesz@gmail.com

***Abstract*—Proof-backed rollups and bridges generate zero-knowledge proofs off chain but rely on checkpoint records published on a public settlement chain. Downstream applications trust the published record, so every guarantee rests on this last step. Yet when a proof is ready and its record is missing, proof-and-record monitoring cannot tell whether publication stalled or the live interface would reject the call. We introduce *checkpointability*: before publication, the monitor dry-runs the exact checkpoint call against the live interface. The check serves two purposes. First, on a fixed checkpoint interface, a successful dry-run shows the call is accepted now and stays accepted until the record appears. Proof completion and gas-paying publication can then be scheduled separately. Second, when the record stays absent, the same check tells whether publication stalled or the call would be rejected. We evaluate it in a live campaign on Ethereum Sepolia: 75 publications and 30 injected fault windows. The check separates all 30, while proof-and-record monitoring cannot tell them apart. The dry-run is a gas-free, read-only call against public chain state, so anyone holding the proof artifact can replay the publisher's own check.**

***Index Terms*—blockchain monitoring, incident response, runtime monitoring, control-plane resilience, rollups, checkpointing, zero-knowledge proofs, cross-chain bridges**

I. Introduction

Proof-backed rollups and bridges separate execution from settlement. A layer-2 system executes transactions off chain and distills the resulting state into a compact claim. It then attaches a zero-knowledge proof and submits both to a checkpoint contract on its settlement chain (layer 1) [1]–[3]. Downstream contracts trust whatever record lands on the canonical chain. Every guarantee therefore depends on one step: an available proof must become a published record. We call this step the *public checkpoint boundary*.

The problem. A monitor at that boundary sees proof evidence in a local log and the checkpoint record exposed by the canonical on-chain interface. A common incident starts when the proof is ready but the record stays pending. This single symptom hides two distinct causes. In a *publication stall*, the publisher (or relayer) has yet to land a call that the interface would accept. In an *acceptance gap*, the live interface rejects the call the publisher is configured to send. Configuration drift on the publisher's side can open such a gap. Both incidents fire the same dashboard alert; the cause and the fix differ.

Existing tools. Runtime monitors classify what has already happened on chain. Pre-flight simulation privately answers whether a sender's transaction would succeed. Neither tells the operator on which side of the boundary the incident lies.

Our approach. We propose *checkpointability*: before publication, the monitor dry-runs the exact reconstructed checkpoint call against the live interface. On the Ethereum Virtual Machine (EVM), the dry-run uses `eth_call`, a read-only simulation against a chosen chain state [4]. A proved claim is *checkpoint-admissible* when the live interface accepts that call. Checkpointability turns this into an observable state: a checkpoint can be *proved and checkpoint-admissible* before spending gas to publish. On a fixed interface this admissibility persists until the record appears, so publication can be deferred. If the record stays absent, the audit separates a publication stall from an acceptance gap.

The audit path. We embed this signal as C in the E–V–C–K audit path: eligibility E, verified proof V, checkpointability C, confirmed record K. Fig. 1 shows the resulting flow. From the four observations, a top-down classifier assigns each incident window to one of six regimes. The regime tells the operator which path to repair. Our main contributions are:

- We introduce *checkpointability*, a pre-publication acceptance signal obtained by dry-running the exact reconstructed checkpoint call, and identify interface conditions under which admissibility persists until the record appears.
- We define the windowed E–V–C–K audit criterion and establish two properties: persistence of admissibility on a fixed interface, and separation of publication stalls from acceptance gaps (Proposition 1).
- We evaluate the criterion on a live, immutable Sepolia deployment covering all six regimes. The classifier separates 30/30 fault windows against 0/30 for the proof-and-record baseline.

Section II reviews related work; Sections III–V present the model, the criterion, and the live evaluation; Section VI concludes.

II. Related Work

Checkpointability builds on two bodies of work: the designs that create the proof-to-record path, and the monitoring and simulation practice that observes it.

Settlement designs. Formal rollup work clarifies proof-backed state evolution and settlement relations [1], and benchmarking separates proof generation from public posting

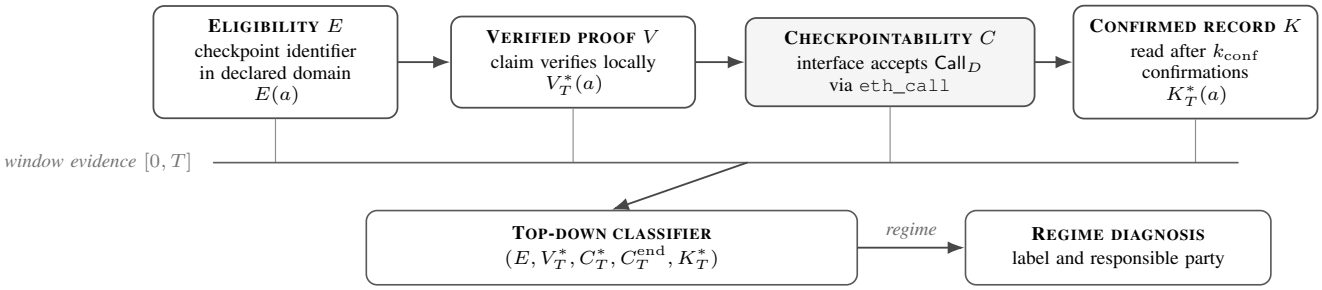

Fig. 1. E–V–C–K audit flow: window evidence feeds the top-down classifier, which outputs a regime and its responsible party.

costs [5]. Both study the path from proof to record as part of the protocol design.

Validium- and Volition-style systems [6] decouple validity from full settlement-layer data publication. Escape-hatch designs show that operator-liveness failures need recovery paths even when cryptographic evidence exists [7]. zkBridge [3] relies on a relay network for liveness while an on-chain verifier decides correctness, and a bridge-security systematization flags intermediary and relay weaknesses [2]. All these designs separate what is admissible from what is published. Checkpointability makes that separation observable at run time.

Runtime monitoring. Runtime verification of smart contracts is mature: instrumentation [8], pending obligations [9], cross-chain metric temporal monitoring [10], and business-logic monitoring [11]. Bridge monitors detect anomalies [12] and attack transactions [13], and resilient alerting for watchtower-style observers is formalized in [14]. All of these tools look backward at what has already happened on chain.

Pre-flight simulation. Unlike runtime monitoring, this practice looks forward: wallets, relayers, and simulation services test candidate transactions before submission. Each check is private to one sender and answers one question: would this transaction succeed right now? Checkpointability turns the same primitive into a public, windowed audit signal at the checkpoint boundary.

III. Checkpointability Model

The audit needs a precise object: which deployment we watch, which signals we read, and how we rebuild the checkpoint call.

A. Deployment Surface and Signals

Deployment and identifiers. We fix one deployment and one canonical checkpoint path on the settlement chain. A checkpoint identifier a names one position in the source execution, such as a block height. For that position, c_a is the checkpoint claim consumed by downstream contracts and π_a is the local proof artifact for c_a, public inputs included. The declared domain H lists the positions the deployment commits to checkpoint. An identifier is eligible (E=1) when it belongs to H.

Interface assumptions. We call the interface *fixed* when three conditions hold. While an identifier has no record, whether the interface accepts a call depends only on the call itself and on data fixed at deployment: the same call always gets the same answer. Publication is permissionless. Only the claim c_a can be recorded under identifier a: one canonical result per position.

Signals. $V(a,t) = 1$ when π_a has verified for c_a by time t under the deployment's public verification key. This observation is local and persistent. $K(a,t) = 1$ when the canonical path shows a record matching the proof-derived pair (a, c_a), confirmed at depth k_{conf}. $C(a,t)$ records whether the call under audit is accepted at the sampled chain state. For an eligible identifier, we call $(V, C, K) = (1, 1, 0)$ the *proved and checkpoint-admissible* state: proof verified, call accepted, record still absent.

Ownership. The prover produces V, and the live interface determines C. The publisher (or relayer) turns admissible calls into K, while the monitor samples and records the evidence.

B. Audited Call and Dry-Run Semantics

The checkpoint call. The publisher bridges proof and record by turning a verified proof into a *checkpoint call*. This call object specifies the target contract, the sender, the call value, the entry point, and the data field. On the EVM, the data field is the calldata.

Two descriptors. The deployment publishes a declared, versioned descriptor D^{decl}; the publisher runs its own configured descriptor D^{pub}. For the descriptor D under audit, the monitor reconstructs the call object $\mathsf{Call}_D(a, c_a, \pi_a)$. D supplies the target, sender, value, and entry point. The calldata is D's Application Binary Interface encoding of the identifier, its claim, and the proof artifact.

What the audit tests. The monitoring policy chooses which descriptor it audits. Auditing D^{decl} tests the canonical call; auditing D^{pub} tests the call the publisher would actually send. When the two differ, that divergence is an acceptance-path fault, and the proof itself stays valid. The call object deliberately excludes the fields that govern publication liveness: nonce management, gas funding and pricing, and mempool (pending-transaction pool) behavior. They affect inclusion, not contract-level acceptance.

Sampling and verdicts. At each sample, the monitor simulates the reconstructed call object against the chain state selected by the monitoring policy. The outcome is acceptance (C=1), reproducible rejection (C=0), or no verdict; only the first two count as valid samples. Remote Procedure Call (RPC) errors, inconclusive results, and inconsistent responses yield

TABLE I
AUDIT REGIMES. ENTRIES MARKED – ARE UNCONSTRAINED

Regime	E	V_T^*	C_T^*	C_T^{end}	K_T^*
Completed path	1	1	–	–	1
Publication stall	1	1	1	1	0
Transient gap	1	1	1	0	0
Acceptance gap	1	1	0	0	0
Proof-log gap	1	0	–	–	1
Scope mismatch	0	–	–	–	–

no verdict, so a transient RPC failure never counts as rejection. Sampling starts after $V = 1$ and stops when the record appears or the window closes.

IV. WINDOWED AUDIT CRITERION

So far, each signal is a single observation. We now read the evidence over a bounded window, state what a healthy checkpoint path must satisfy, and build the regime classifier.

A. Window Evidence

Derived quantities. All evidence is read over a bounded window $[0, T]$. For $X \in \{V, C, K\}$, the marker $X_T^*(a)$ records whether X reached 1 during the window. The endpoint value $C_T^{\mathrm{end}}(a)$ is the value of the last valid dry-run sample in the window, or $\perp$ when none exists.

Healthy path. A healthy path meets three expectations. An eligible proved claim becomes admissible and stays admissible until its record appears. Publication then turns that admissibility into a confirmed record. And every confirmed record traces back to proof evidence.

B. Regimes and Separation

Reading the table. The monitor records the window evidence for each checkpoint identifier with a verified proof or a record in the window, then reads Table I from top to bottom. The diagnosis is the first row whose constrained entries match. The first four regimes track eligible proved claims; the last two flag evidence and scope faults:

- *Completed path*: the claim ends with a confirmed record.
- *Publication stall*: endpoint acceptance, yet no record.
- *Transient gap*: accepted, then rejected at the window end.
- *Acceptance gap*: every valid sample is a rejection.
- *Proof-log gap*: a record without proof evidence in the monitor state.
- *Scope mismatch*: an observed proof or record uses an identifier outside the declared domain.

When no row matches, the label is INSUFFICIENT-EVIDENCE, not a regime, and the operator repairs sampling first.

Persistence. Fix an eligible checkpoint identifier and an audited call, and pair each C sample with a record read at the same confirmed block. While the paired read shows no record, the interface assumptions freeze everything acceptance depends on. Once the call is admissible, it therefore stays admissible at every later valid sample until the record appears. Within that interval, a rejection can only mean the audited call itself changed; that drift is what the transient-gap regime captures.

Stall–acceptance separation (Proposition 1). *Fix an eligible checkpoint identifier $a \in H$ in a window where the proof verifies and the checkpoint record is absent, so that $(E, V_T^*, K_T^*) = (1, 1, 0)$. Any classifier observing only (V_T^*, K_T^*) assigns the same label to a publication stall and an acceptance gap. The E–V–C–K classifier separates them whenever the window has at least one valid sample.*

The two regimes leave opposite C traces. A stall ends the window still accepted (C_T^{end}=1): the interface would take the call, and only publication is missing. An acceptance gap never accepts (C_T^*=0): the audited call is rejected from the start, while the proof itself stays valid. Table I therefore assigns them distinct rows.

Cost. Each C sample costs one gas-free `eth_call`. Checkpointability is observed only at sample times: a gap that opens and closes between two samples leaves no trace. A tighter cadence trades extra RPC calls for transient-gap recall.

V. EXPERIMENTAL EVALUATION

We take the four signals to a live Sepolia deployment and exercise all six regimes end-to-end.

A. Live Sepolia Campaign

Deployment stack. On the source side, a private development network (devnet, chain ID 1337) feeds the campaign. Its client is Reth, a Rust-based Ethereum execution client [15]. The proving backend is OpenVM, an open-source zero-knowledge virtual machine (zkVM) framework [16]. Together they produce the proof artifact with its public inputs.

The settlement chain is Sepolia, a public Ethereum test network (chain ID 11155111). On Sepolia, *CheckpointManager* calls the deployed OpenVM verifier during *publishCheckpoint* and stores the record only on success. The manager is immutable and permissionless. Verifier and schema are fixed, and while an identifier has no record, acceptance depends only on the call and data fixed at deployment.

The campaigns run sequentially on a Google Cloud `n2-standard-32` host (32 virtual CPUs, 128 GB RAM); an `n2-standard-16` secondary host (16 virtual CPUs, 64 GB RAM) replays the positive campaign.

Run protocol. We report results by workload bucket: runs proving 10, 25, 50, 75, or 100 source-chain blocks, with ten primary and five secondary runs per bucket. Every positive run is observed for 24 hours. Each run reserves a fresh checkpoint identifier in the declared domain H for its single proof-derived claim. A claim counts as checkpointed once readable through the canonical interface after k_{conf}=2 confirmations.

On-chain anchors. Table II records the positive campaign's contracts, sender, entry point, and publication block range, so anyone can check all 75 publications against the canonical chain.

Performance measurements. Proof time stays roughly flat across the workload range (Table III), consistent with the zkVM's fixed overhead dominating per-block cost at this scale.

TABLE II
SEPOLIA DEPLOYMENT ANCHORS

Anchor	Value
Runs	75 positive publications; 30 unpublished fault windows; 5 successful follow-up publications.
Chains	source devnet 1337 (private); Sepolia 11155111; k_{conf}=2; RPC `publicnode-sepolia`.
Publisher	`0x93Aff92F79BD03807a26bE06D52A0790D68F79EE`; call value 0 wei.
Manager	*CheckpointManager* `0x65A52C787B2153EA8fe980f93c5569023eE9357f`; deployed at block 10830148.
Verifier	*OpenVmHalo2Verifier* `0xB17fD1AE3604B296aC485bC1BcdC526671633b2a`; deployed at block 10830147; called by the manager.
Interface	*publishCheckpoint*, selector `0xae8dd7f3`; confirmed reads *checkpointExists*/*getCheckpoint*.
Publications	75 transactions; blocks 10830327–10845913; mined 2026-05-11 to 2026-05-13 UTC.

TABLE III
POSITIVE CAMPAIGN METRICS, PRIMARY HOST: PROOF TIME AS MEAN [95% CONFIDENCE INTERVAL]

Blocks	Runs	Proof (s)	Delay (s)	Dry-run (ms)
10	10	1330.5 [1302.1, 1359.0]	70.9	160.8
25	10	1338.0 [1322.6, 1353.5]	77.3	144.7
50	10	1344.9 [1324.1, 1365.8]	67.7	122.4
75	10	1332.9 [1320.0, 1345.8]	74.0	109.4
100	10	1364.3 [1342.9, 1385.8]	80.2	125.0

The proof-to-record delay, from verified proof to confirmed read, holds at 68–80 s. Each dry-run costs 109–161 ms. The secondary host reproduces the flat profile at roughly twice the proof time (means 2845–2895 s), with comparable delays and dry-run latencies.

Sampling policy. For positive runs, each C sample and its record read use the same recorded k_{conf}-deep block, so both observations describe the same chain state. While the paired block shows no record, an admissible call stays accepted. Once the record appears, replay protection rejects the same call: the checkpoint already exists. Persistence promises admissibility only up to that point. The completed-path row therefore rests on the confirmed record, not on endpoint acceptance.

B. Fault Injection and Live Follow-Ups

Fault protocol. The fault campaigns reproduce publication stalls and acceptance gaps across every bucket. Targeted live follow-ups on the same deployment cover the three remaining fault regimes. The positive campaign covers the completed path. Each fault window opens at proof completion and lasts 300 s, with one `eth_call` sample every 30 s and three repetitions per bucket. Every observed identifier is recorded with its full window evidence.

Publication stall. The injection suppresses the checkpoint relay after proof completion while keeping the call aligned with the live interface. Each window holds a checkpoint that is proved and checkpoint-admissible, yet unpublished. All 15 windows end with the call still admissible, matching the publication-stall row of Table I.

Acceptance gap. The injection gives the publisher a stale descriptor: D^{pub} carries legacy call metadata while the live interface expects D^{decl}. The audit tests $\mathsf{Call}_{D^{\mathrm{pub}}}$, and the live manager rejects it. Every valid sample rejects in all 15 windows, and the proof itself stays valid.

Transient gap. A live follow-up drifts the audited selector shortly after the window opens. The manager rejects the drifted call until a rollback restores the canonical call, which lands after the window closes. In the three 10-block runs, early samples accept and the endpoint sample rejects.

Proof-log gap and scope mismatch. Two further follow-ups publish valid checkpoints under identifiers 50000 and 60000. The first is re-audited with the proof log hidden, so the monitor sees a record with no proof evidence (V_T^*=0, K_T^*=1). The second declares the domain $\{61000\}$, which excludes the published identifier (E=0).

Routing. A publication stall points at the publisher path, an acceptance gap at the acceptance path, and a transient gap at temporary drift in the audited descriptor. A proof-log gap needs evidence repair; a scope mismatch needs a policy review.

Results. The E–V–C–K classifier separates 30/30 fault windows and updates the diagnosis within one sampling interval plus RPC latency. A baseline observing only proof and record separates 0/30: every window shows the same symptom, a verified proof and no record.

C. Scope and Robustness

Perimeter. The audit covers the public checkpoint boundary of one immutable, proof-backed deployment with fixed checkpoint path, confirmation depth, and declared domain.

Sample semantics. An accepting C sample is taken at one chain state; under the interface assumptions, the audited call stays admissible until its record appears. Inclusion, nonce readiness, gas, mempool policy, and reorg (block-reorganization) safety remain publication-liveness concerns. RPC errors and inconsistent responses affect observation, not admissibility.

VI. CONCLUSION

When a proof has verified but its checkpoint record stays absent, rollup and bridge operators face one alert with two possible causes. This paper introduced *checkpointability*, a dry-run of the exact reconstructed checkpoint call against the live interface, embedded in the E–V–C–K audit path. The signal makes a latent state observable: a checkpoint can be proved and checkpoint-admissible before spending gas to publish. While the record is absent, acceptance depends only on the call and data fixed at deployment, so this admissibility persists: the checkpoint can be published later. The six regimes turn a "proved and pending" alert into a diagnosis, naming which path failed and what to fix. On Sepolia, all 30 injected fault windows looked identical to proof-and-record monitoring; the classifier separated them all. Because the check runs against public chain state, anyone with the proof artifact can replay it.

References

[1] S. Chaliasos, D. Firsov, and B. Livshits, "Towards a Formal Foundation for Blockchain ZK Rollups," in *Proc. ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, 2025, pp. 2714–2728, doi: 10.1145/3719027.3765115.

[2] A. Augusto, R. Belchior, M. Correia, A. Vasconcelos, L. Zhang, and T. Hardjono, "SoK: Security and Privacy of Blockchain Interoperability," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 3840–3865, doi: 10.1109/SP54263.2024.00255.

[3] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. Song, "zkBridge: Trustless Cross-chain Bridges Made Practical," in *Proc. 2022 ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, 2022, pp. 3003–3017, doi: 10.1145/3548606.3560652.

[4] Ethereum Foundation, "Execution APIs: Ethereum JSON-RPC Specification," 2025. [Online]. Available: https://github.com/ethereum/execution-apis

[5] S. Chaliasos, I. Reif, A. Torralba-Agell, J. Ernstberger, A. Kattis, and B. Livshits, "Analyzing and Benchmarking ZK-Rollups," in *6th Conf. on Advances in Financial Technologies (AFT 2024)*, ser. *Leibniz Int. Proc. in Informatics (LIPIcs)*, vol. 316, 2024, pp. 6:1–6:24, doi: 10.4230/LIPIcs.AFT.2024.6.

[6] StarkWare, "StarkEx Documentation: Data Availability Modes," 2024. [Online]. Available: https://docs.starkware.co/starkex

[7] F. G. Figueira, M. Derka, C.-L. Chiu, and J. Gorzny, "A Practical Rollup Escape Hatch Design," in *Proc. IEEE Int. Conf. on Blockchain and Cryptocurrency (ICBC)*, 2025, pp. 1–5, doi: 10.1109/ICBC64466.2025.11114670.

[8] J. Ellul and G. J. Pace, "Runtime Verification of Ethereum Smart Contracts," in *2018 14th European Dependable Computing Conference (EDCC), Workshop on Blockchain Dependability*, 2018, pp. 158–163, doi: 10.1109/EDCC.2018.00036.

[9] M. Capretto, M. Ceresa, and C. Sánchez, "Monitoring the Future of Smart Contracts," in *Fundamental Approaches to Software Engineering (FASE 2024)*, ser. *Lecture Notes in Computer Science*, vol. 14573, 2024, pp. 122–142, doi: 10.1007/978-3-031-57259-3_6.

[10] R. Ganguly, Y. Xue, A. Jonckheere, P. Ljung, B. Schornstein, B. Bonakdarpour, and M. Herlihy, "Distributed Runtime Verification of Metric Temporal Properties for Cross-Chain Protocols," in *42nd IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, 2022, pp. 23–33, doi: 10.1109/ICDCS54860.2022.00012.

[11] M. Eshghie, C. Artho, H. Stammler, W. Ahrendt, T. T. Hildebrandt, and G. Schneider, "HighGuard: Cross-Chain Business Logic Monitoring of Smart Contracts," in *Proc. 39th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, 2024, pp. 2378–2381, doi: 10.1145/3691620.3695356.

[12] A. Augusto, R. Belchior, J. Pfannschmidt, A. Vasconcelos, and M. Correia, "XChainWatcher: Identifying Anomalies in Cross-Chain Bridges," in *Proc. 26th ACM Int. Middleware Conf. (Middleware)*, 2025, pp. 413–426, doi: 10.1145/3721462.3770781.

[13] J. Wu, K. Lin, D. Lin, B. Zhang, Z. Wu, and J. Su, "Safeguarding Blockchain Ecosystem: Understanding and Detecting Attack Transactions on Cross-chain Bridges," in *Proc. ACM Web Conf. 2025 (WWW)*, 2025, pp. 4902–4912, doi: 10.1145/3696410.3714604.

[14] M. Mouallem, L. Breidenbach, I. Eyal, and A. Juels, "Resilient Alerting Protocols for Blockchains," to appear in *Proc. ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, 2026, arXiv:2602.10892.

[15] Paradigm, "Reth: A Modular, Contributor-Friendly and Fast Implementation of the Ethereum Protocol," 2025. [Online]. Available: https://github.com/paradigmxyz/reth

[16] The OpenVM Authors, "OpenVM: A Performant and Modular zkVM Framework," 2025. [Online]. Available: https://github.com/openvm-org/openvm