Checkpointability: Separating Proof Availability from Publication Liveness

Jonathan Oleszkiewicz
BaaS.sh
France
jonathanolesz@gmail.com

***Abstract*—Proof-backed rollups and bridges separate zero-knowledge proof generation from settlement on the public chain. Their runtime monitors observe a proof in the local log and a checkpoint record on chain. However, when the proof has verified and the record stays pending, two operationally distinct incidents share the same alert: a publisher-side *publication stall* and an interface-side *acceptance gap*. We introduce *checkpointability*, a runtime criterion that asks, before publication, whether the live interface accepts the call prepared for a proved claim. We embed it in an E–V–C–K audit path (eligibility, verified proof, checkpointability, confirmed record) and classify traces within a bounded incident window into six labelled regimes. We show formally that (V, K) alone conflates the two regimes, while adding C separates them. Across 75 published checkpoints, 30 controlled fault injections, and 5 targeted live follow-ups covering all six audit regimes on Sepolia, a dry-run on the public checkpoint boundary separates publisher-side stalls from interface-side rejections. The signal is forward-looking and publicly verifiable: operators get a pre-publication check, observers replay the publisher's own check, and the ecosystem gains a third publicly reproducible signal on the checkpoint boundary.**

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

I. Introduction

Context. Proof-backed rollups and bridges factor their trust pipeline into well-understood stages: execution, zero-knowledge proving, settlement, and data availability [1]–[3]. A rollup distills layer-2 state into a compact claim, attaches a zero-knowledge proof, and submits both to its base settlement chain (layer 1). A bridge in the same class consumes state from another chain and inherits finality from settlement-layer checkpoints. In both cases, the contracts downstream of the checkpoint interface trust whatever record lands on the canonical chain. This paper sits at the *public checkpoint boundary*. It is the operational seam between proof availability and publication liveness.

Public checkpoint boundary. Operators rarely instrument every prover step. The monitor on duty typically sees two artifacts: proof evidence in its local log and the authorized checkpoint interface on chain. We write K for the confirmed record exposed by that interface. The publisher (or relayer) bridges proof and record by turning a verified proof into a checkpoint call that fixes the target contract, the sender or role, the value, the entry point, and the encoded data. The declared specification and the publisher configuration must agree; when they drift, the publisher submits a call that the live interface rejects.

Incident ambiguity. A common incident starts at exactly that seam. The proof is ready, but the checkpoint record stays pending. However, this single symptom hides two distinct causes. In a *publication stall*, the publisher path is still sitting on an admissible call. In an *acceptance gap*, the live interface rejects the publisher-configured call, often because of descriptor skew, role rotation, a pause, replay protection, or an upgrade. Both fire the same dashboard alert; the owner and the fix differ.

E–V–C–K path. We split the decision into four observable signals. E checks whether the checkpoint key is in scope. V checks whether a local proof artifact exists and verifies. C tests whether the live interface accepts the checkpoint call under audit. K checks whether the record is confirmed and readable. Together they form the E–V–C–K audit path.

Checkpointability. *Checkpointability* is a new public signal on the public checkpoint boundary, between proof availability and publication liveness. It captures the live acceptance state of the checkpoint interface. A proved claim is *checkpoint-admissible* when the live interface accepts the checkpoint call derived from the proof artifact and the descriptor under audit. The signal answers a single operational question: would the proved claim be accepted right now at the public interface? If yes, publication can proceed; if not, the operator first repairs the call or the interface state.

Figure guide. Fig. 1 reads the audit path in three layers. The top row records the four observations E, V, C, and K along the proof-to-record path. The middle line collects their window evidence over $[0, T]$, including endpoint and rejection evidence for C. The classifier then maps that tuple to a regime label (Table II) and routes the incident to a responsibility boundary.

Audit symbols. The figure refers to:

- a is the checkpoint key.
- c_a is its claim.
- π_a is the local proof artifact.
- $\mathsf{Call}_D(a, c_a, \pi_a)$ is the call constructor (§III).

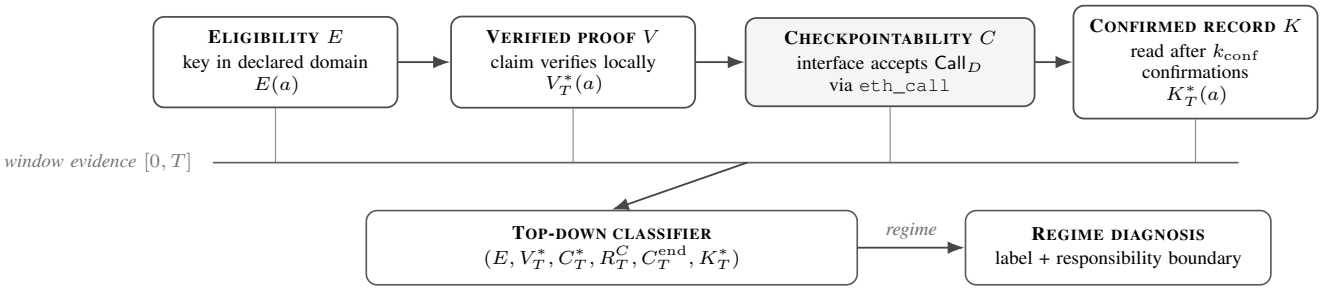

Fig. 1. E–V–C–K audit flow from window evidence to regime diagnosis.

For $X \in \{V, C, K\}$, the window marker is

$$X_T^*(a) = 1 \quad \Longleftrightarrow \quad X \text{ is observed for } a \text{ during } [0, T].$$

For unpublished traces, $C_T^{\mathrm{end}}(a)$ records the endpoint value of C, and $R_T^C(a)$ flags at least one reproducible rejection.

Notation. We write H for the declared checkpoint domain, so $E(a) = 1$ iff $a \in H$. The confirmation depth is k_{conf} and the audit window is $[0, T]$. Endpoint acceptance separates publication liveness from interface acceptance; rejection evidence separates a real rejection from missing C samples. On the EVM (Ethereum Virtual Machine), C is sampled with `eth_call` (a read-only call simulation), which runs the call against a chosen chain state; sampling is windowed because interface state can change.

Contributions. We make five contributions.

1) We identify *checkpointability*, a new public signal at the public checkpoint boundary, and use it as the load-bearing signal of a six-regime classifier that routes incidents to responsibility boundaries.
2) We separate three concerns on the proof-to-record path: interface admissibility, publication liveness, and monitor evidence completeness.
3) We show that adding C to V and K separates publication stalls from acceptance gaps that (V, K) alone conflate (Proposition 1).
4) We give a windowed audit criterion over (E, V, C, K) and classify bounded traces into six labelled regimes.
5) We evaluate the criterion on Sepolia, using a private Reth/OpenVM source-chain deployment, covering all six regimes end-to-end: 75 positive publications, 30 controlled fault injections, and 5 targeted live follow-ups.

The scope is the public checkpoint boundary in a proof-backed deployment. Section VII states the evaluation perimeter.

II. Related Work

Rollup foundations. Formal rollup work clarifies proof-backed state evolution and settlement relations [1]. Benchmarking work separates proof generation from public posting costs [4]. Checkpointability targets the boundary between those paths. It asks whether a completed proof is acceptable to the live checkpoint interface before the record is visible on chain.

Data availability and selective posting. Validium and Volition-style systems separate validity from full settlement-layer data publication[1]. Escape-hatch work shows that operator-liveness failures still need recovery paths when cryptographic evidence exists [5]. These systems motivate monitors that distinguish what is admissible from what is published, while still treating the settlement interface as the source of finality.

Bridges and interoperability. ZK-based bridges such as zkBridge [3] rely on a relay network for liveness, while an on-chain verifier decides correctness. A bridge-security SoK identifies intermediary-network and relay-related weaknesses as important security concerns [2]. This supports an operational split between relay behavior and interface acceptance.

Runtime monitoring and control planes. Runtime verification of smart contracts is mature: Ethereum instrumentation [6], pending obligations across transactions [7], cross-chain metric temporal monitoring [8], and specification-driven business-logic monitoring [9]. Cross-chain bridge monitoring has matured in parallel, with logic-driven anomaly detection over heterogeneous chain events [10] and transaction-level attack pattern detection [11]. Bribery-resistant alerting for watchtower-style observers (passive on-chain monitors) is formalized in recent work [12]. These monitors classify what has already happened on chain; checkpointability tests what would happen on the next call. This paper extends runtime monitoring to the public checkpoint boundary, with a live-acceptance signal that lets the operator separate publisher-side delays from interface-side rejections within a single incident window.

III. Checkpointability Model

A. Deployment Surface

Audit domain. We fix one deployment instance and one authorized checkpoint path on the settlement chain. The monitor reads checkpoint records for K only after a confirmation depth k_{conf}. Each C sample runs against the chain state that the monitoring policy picks at the sampling time. Downstream protocols consume the records exposed by the same checkpoint interface.

For each checkpoint key $a \geq 1$, let c_a be the checkpoint claim consumed by downstream contracts; in a rollup, c_a

[1] StarkEx

can be a height-tagged state commitment. Let $H \subseteq \mathbb{N}$ be the declared checkpoint domain. It contains the keys the deployment promises to expose through the declared checkpoint interface. Eligibility is then

$$E(a) = 1 \quad \iff \quad a \in H.$$

The classifier applies only to checkpoint-domain evidence. A key beyond H is a scope mismatch only when it appears in the checkpoint-domain proof log or the authorized checkpoint-record stream.

Checkpoint call. The checkpoint call is the execution-level call object submitted by the authorized publisher. It fixes the target, the sender or role-bound account, the value, the entry point, and the data field (calldata on the EVM, carrying any signatures or proofs the manager requires). The auditor reconstructs the full call object for the descriptor under audit. When the publisher descriptor matches the declared specification, this object is the *declared canonical* checkpoint call; when the descriptor is stale, it is the *publisher-configured* call, and the divergence is an acceptance-path fault. Publication-liveness fields such as nonce management, gas funding, gas pricing, and mempool (the pool of pending transactions) behavior belong to publication liveness, handled separately from checkpointability.

Public boundary and responsibility. The audit boundary is the public interface. The prover can use batching, recursion, specialized hardware, or any internal pipeline; downstream contracts see only a compact claim and an authorized call path. The criterion uses that public view: verify the proof artifact, reconstruct the call, sample interface acceptance, read the confirmed record. The prover owns V, the interface state determines C, the publisher (or relayer) turns C into K, and the monitor owns sampling and evidence capture. This split gives an incident commander a direct handoff.

B. Checkpoint Path Security Model

Verification boundary. The criterion separates proof validity from checkpoint publication. When a checkpoint call invokes an on-chain verifier, verifier non-reversion is part of C. In the live Sepolia campaign, `CheckpointManager` calls the deployed OpenVM verifier during `publishCheckpoint` and stores the record only on success. V remains the local pre-publication signal that ties together the proof artifact, public inputs, verification key, and proof-log claim. Section V shows how the campaign ties the locally verified OpenVM public inputs to the checkpoint claim, the reconstructed calldata, the C samples, and the confirmed manager reads.

Publisher power. Only the local verification step under the deployment verification key sets V in the audit state. The publisher may withhold an admissible call, causing a publication stall. It can also use a stale descriptor, causing an acceptance-path fault when the dry-run rejects. A record contributes to K only when the authorized checkpoint path exposes the same key and claim derived from the verified artifact.

C. Signals

Signal set. The model uses four observable signals. Signals V and K are persistent observations. Once observed, they remain true for the audit window. Signal C is point-in-time dry-run evidence and can change during the window.

TABLE I
NOTATION

Symbol	Meaning	Source	Role
$E(a)$	key a in declared domain H	deployment policy	scope
$V(a,t)$	π_a verified for c_a by time t under the verification key	prover log	proof availability
$C(a,t)$	checkpoint call accepted at time t	`eth_call` dry-run	interface acceptance
$K(a,t)$	confirmed record for a, c_a exposed by time t	authorized path	publication outcome
$V_T^*(a)$	$\exists\, t \in [0,T]\ V(a,t) = 1$	V samples	V observed in window
$C_T^*(a)$	$\exists\, t \in [0,T]\ C(a,t) = 1$	C samples	C observed in window
$K_T^*(a)$	$\exists\, t \in [0,T]\ K(a,t) = 1$	K reads	K observed in window
$R_T^C(a)$	valid reproducible rejection in $[0,T]$	C samples	rejection evidence
$C_T^{\mathrm{end}}(a)$	$C(a,T)$ if a valid endpoint sample exists, else $\perp$	endpoint C sample	endpoint acceptance

Signal anchors. V is anchored in local proof verification under the deployment's public verification key. C is anchored in the live interface; when the manager calls an on-chain verifier, verifier non-reversion is part of this signal. K is anchored in the authorized checkpoint path. Artifacts pending verification are evidence-quality issues, handled separately from proved claims. When $V = 1$, the monitor also checks that K matches the proof-derived claim.

D. Checkpointability Samples

Dry-run semantics. A dry-run is *authorization-faithful* when it simulates the exact checkpoint call object under audit. It uses the same target, data field, sender or role, and call value as the publisher path. The simulation runs against the chain state selected by the monitoring policy at time t. The binary sample is

$$C(a,t) = \begin{cases} 1, & \text{interface call succeeds,} \\ 0, & \text{rejection is reproducible.} \end{cases}$$

Adapters map return-value-based interfaces to the same accept/reject signal.

Sample outcomes. The sampler records:

- success,
- reproducible rejection,
- RPC (Remote Procedure Call) error,
- inconclusive result,
- incoherent evidence.

Only success and reproducible rejection count as valid binary C samples. Rejection evidence requires a reproducible call-level reject; a transient RPC failure flows into the inconclusive bucket. We denote valid rejection evidence by $R_T^C(a)$.

For C-dependent regimes, the classifier first checks sample quality. A window with only RPC errors or inconclusive attempts yields INSUFFICIENT-EVIDENCE, a status that sits alongside the audit regimes. The acceptance-gap row is reached only under

$$C_T^*(a) = 0 \quad \text{and} \quad R_T^C(a) = 1.$$

Scope of C. Checkpointability speaks to interface acceptance at the sampled chain state. The following stay with publication liveness:

- transaction inclusion, nonce readiness, gas funding, gas pricing,
- mempool admission and reorg (block reorganization) safety,
- post-sample race exposure.

Subsequent role changes, pauses, timestamp guards, and replay-protection changes are observed by later samples rather than predicted by this one.

Sampling policy. The first C sample is taken after $V = 1$. Further samples follow the window cadence until the record appears or the window closes. The policy keeps the acceptance view for C distinct from the confirmed-record view for K. Sections V and VI describe the campaign block tags, live heads, and k_{conf}=2 confirmed-record reads.

E. Canonical Call Construction

Construction rule. The deployment commits to a public, versioned declared call specification D^{decl}. The publisher holds a configured descriptor D^{pub}; in a healthy deployment, $D^{\text{pub}} = D^{\text{decl}}$. The audit parameter D denotes the descriptor being tested. For a verified proof artifact π_a, descriptor D derives the full call object

$$\mathsf{Call}_D(a, c_a, \pi_a) \triangleq (to_D, from_D, value_D, data_D),$$
$$data_D \triangleq \mathsf{ABI}_D(a, c_a, \pi_a).$$

ABI_D is the ABI (Application Binary Interface) calldata encoder for the function selector and its encoded arguments. Call_D adds the target manager address, sender or role, call value, encoding version, and any public-input commitment binding π_a to c_a; authorization material (signatures or proofs) is carried inside $data_D$ or by $from_D$.

When $D = D^{\text{decl}}$, the call is declared canonical. When $D = D^{\text{pub}} \neq D^{\text{decl}}$, it is the divergent publisher-configured call. A payload mismatch is therefore a divergence between publisher configuration and the live interface, with the proof itself remaining valid.

Time variation. $C(a, t)$ is evaluated against the chain head selected by the monitoring policy, so it is time-varying. Pauses, role rotation, replay protection, and upgrades can change the answer across a window. The audit records both observed acceptance history and the endpoint snapshot.

IV. WINDOWED AUDIT CRITERION

Window notation. Let S_T^V, S_T^C, and S_T^K be the finite observation times in $[0, T]$ for proof evidence, checkpointability samples, and checkpoint-record reads. The monitoring policy fixes the window start for each audited key; in the fault campaigns of Section VI, $t = 0$ is proof completion.

If the checkpoint is still unpublished at T, S_T^C includes the endpoint sample; otherwise C sampling closes at or before the confirmed-record read. The completed-path row therefore relies on the history bit C_T^* and treats endpoint acceptance as a free field.

For any $X \in \{V, C, K\}$, define the existence markers

$$X_T^*(a) \triangleq \exists\, t \in S_T^X \ \big(X(a,t) = 1\big),$$
$$X^*(a) \triangleq \exists\, t \ \big(X(a,t) = 1\big).$$

Thus $X_T^*(a) = 1$ means X reached value 1 for a during $[0, T]$.

For unpublished traces, the endpoint value of C is

$$C_T^{\text{end}}(a) = \begin{cases} C(a,T), & \text{when a valid endpoint sample exists,} \\ \bot, & \text{otherwise.} \end{cases}$$

A $\bot$ entry leaves the corresponding field of Table II as a free entry.

Rejection evidence is captured by

$$R_T^C(a) \triangleq \exists\, t \in S_T^C \ \big(C(a,t) = 0\big),$$

defined over valid reproducible rejection samples. Thus $C_T^*(a) = 0$ marks the absence of any valid acceptance sample, while $R_T^C(a) = 1$ supplies positive rejection evidence.

A. Design Conditions

Design clauses. On the declared domain H, a checkpoint path is correct for downstream consumption when it satisfies three expectations:

$$a \in H \ \wedge\ V(a,t) = 1 \Rightarrow \exists\, t' \geq t \ \big(C(a,t') = 1\big), \quad (1)$$
$$a \in H \ \wedge\ C(a,t) = 1 \Rightarrow \exists\, t' \geq t \ \big(K(a,t') = 1\big), \quad (2)$$
$$a \in H \ \wedge\ K(a,t) = 1 \Rightarrow \exists\, t' \leq t \ \big(V(a,t') = 1\big). \quad (3)$$

The three clauses cover, in order:

- interface admissibility for eligible proved claims,
- publisher-path liveness for admissible claims,
- monitor evidence completeness.

A proof-log gap violates the third clause while the on-chain record remains valid on its own terms. Together, on H, the clauses yield

$$a \in H \quad \Longrightarrow \quad V^*(a) \Rightarrow C^*(a) \Rightarrow K^*(a),$$

and the monitor-completeness direction $K^*(a) \Rightarrow V^*(a)$.

B. Audit Regimes

Classification procedure. The monitor evaluates every checkpoint key scheduled for audit in H during $[0, T]$, plus any key appearing in the checkpoint-domain proof log or authorized checkpoint-record stream. A declared key that surfaces only as a domain entry routes to proof-production or availability triage, handled separately from the checkpointability regimes.

For each observed key, the monitor records $E(a)$, $V_T^*(a)$, $C_T^*(a)$, $R_T^C(a)$, $C_T^{\mathrm{end}}(a)$, and $K_T^*(a)$, then reads Table II top-down.

Rows that depend on C apply only after the sample-quality gate passes; if it fails, or if $C_T^* = 0$ with $R_T^C = 0$, the run lands on INSUFFICIENT-EVIDENCE, gating any acceptance-gap diagnosis until valid C samples appear. Proof-log and scope-mismatch rows remain assignable directly from E, V, and K, independent of call reconstruction.

TABLE II
AUDIT REGIMES

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

Fields shown as – remain free after the table is read top-down. The publication-stall row makes endpoint acceptance explicit. Rows with endpoint value zero require a valid endpoint rejection sample. The acceptance-gap row requires the valid-sample set to fall entirely on the rejection side, with at least one reproducible rejection. INSUFFICIENT-EVIDENCE sits alongside the table as a sampling-quality status that gates C-dependent rows whenever valid C samples are absent. Declared keys that surface only as scheduled entries route to proof-production or availability alerts. A posted checkpoint stays pending until the k_{conf}-confirmed read.

Operational meaning. The regime labels read as follows:

- *Completed checkpoint path*: the proof-to-confirmed-record path completed during the window.
- *Publication stall*: the proof completed, the live interface accepts the call at T, and the record is still absent.
- *Transient gap*: acceptance appeared during the window but is rejected at the endpoint.
- *Acceptance gap*: the proof completed, the valid-sample set fell entirely on the rejection side, and reproducible rejection evidence appears.
- *Proof-log gap*: a checkpoint record is visible while the corresponding proof evidence is still pending in the monitor state.
- *Scope mismatch*: an observed proof or record uses a key beyond H.

For INSUFFICIENT-EVIDENCE, the operator repairs the sampling path before relying on any C-dependent diagnosis.

Proposition 1 (Separation by checkpointability)**.** *Fix an eligible key $a \in H$ in a window where the proof verifies and the checkpoint record is absent. The shared projection is*

$$(E, V_T^*, K_T^*) = (1, 1, 0).$$

Any classifier that observes only $(V_T^*(a), K_T^*(a))$ assigns the same state to a publication stall and an acceptance gap. The E–V–C–K classifier separates them whenever valid C samples exist.

Proof. Consider two traces with the shared projection above. Their checkpointability evidence differs:

$$\text{publication stall:} \quad (C_T^*, R_T^C, C_T^{\mathrm{end}}) = (1, 0, 1),$$
$$\text{acceptance gap:} \quad (C_T^*, R_T^C, C_T^{\mathrm{end}}) = (0, 1, 0).$$

In the first trace, the call is accepted during the window and at the endpoint. Table II classifies it as a publication stall. In the second trace, the valid-sample set falls entirely on the rejection side and endpoint rejection is reproducible. Table II classifies it as an acceptance gap. Both traces have the same projection onto (V_T^*, K_T^*). Therefore, adding C, endpoint acceptance, and rejection evidence separates the two cases, since V and K alone share the same projection. □

C. Cost and Necessity of C

Sampling cost. V and K come from streams. C requires active simulation against the selected chain head. On EVM, each sampled key and time costs one `eth_call`. The call consumes RPC and CPU resources while staying simulated, bypassing transaction submission and on-chain gas. Sampling starts after the proof artifact verifies and follows the window policy until the record appears or the window closes.

Decision value. Adding C resolves the central ambiguity. When proof evidence exists but the record is absent, the projection $(V_T^*(a), K_T^*(a)) = (1, 0)$ fits both a publication stall and an acceptance gap; Proposition 1 makes the separation precise.

For unpublished traces, each C-derived bit resolves one ambiguity:

- $C_T^{\mathrm{end}}(a)$ separates publication stalls from endpoint rejection.
- $C_T^*(a)$ separates persistent acceptance gaps from transient gaps.
- $R_T^C(a)$ separates acceptance gaps from missing checkpointability samples.

Denser sampling captures shorter admissible intervals.

Baseline comparison. A baseline monitor that observes only (V, K) groups the bucketed runs into two visible labels: 75 completed paths (where $K^* = 1$) and 30 indistinguishable

proved-and-pending traces. Adding C partitions those 30 traces into 15 publication stalls (endpoint $C{=}1$) and 15 acceptance gaps ($C^{*}{=}0$ with reproducible rejection). The (V, K) baseline therefore achieves $0/30$ separation on the fault classes; the E–V–C–K classifier reaches $30/30$ on the same window.

Cadence sensitivity. The sampling cadence bounds the temporal resolution of the monitor: shorter admissible intervals may be missed unless the cadence is reduced or phase-randomized. The campaign uses $\delta{=}30\,\mathrm{s}$ on a $T{=}300\,\mathrm{s}$ window. Choosing δ trades RPC cost for transient-gap recall.

Operator loop. Check whether the proof exists, simulate the exact checkpoint call, then read the record after confirmation. The added step is the simulation. It turns "proved and pending" into a runtime diagnosis.

V. Live Sepolia Campaign

Deployment. We instantiate the four signals at the public checkpoint boundary, on a live Sepolia campaign fed by a private source-chain devnet. The source-side client is Reth (a Rust-based Ethereum execution client) running an in-process Execution Extension (ExEx)[2]; the proving backend is OpenVM, an open-source zkVM (zero-knowledge virtual machine) framework[3]. The source chain is a private devnet whose chain ID 1337 is an arbitrary local value; only the Sepolia settlement chain is public. Reth and OpenVM together produce a zero-knowledge proof artifact and its public inputs, and the Sepolia checkpoint path records the resulting claim.

Reporting buckets. We report results by workload bucket $b \in B = \{10, 25, 50, 75, 100\}$ source-chain blocks. The run grid uses $k_{\mathrm{conf}}{=}2$ confirmations, $n_p{=}10$ primary and $n_s{=}5$ secondary runs per bucket, a 24-hour observation window per positive run, and 300 s fault windows sampled every 30 s with three repetitions per bucket. Buckets label workloads; on-chain keys come from an injective mapping: for run i and bucket b, that mapping yields the actual key

$$a_{i,b} = m_i(b),$$

drawn from a strictly-increasing height counter and stored as `actual_height`, so repeated runs at the same bucket map to distinct keys. The declared domain H contains every $a_{i,b}$, so all audited runs are eligible. Predicates E, V, C, and K are evaluated on $a_{i,b}$ and aggregated by b. A claim is checkpointed once readable through the authorized interface on the canonical chain after $k_{\mathrm{conf}}{=}2$ confirmations.

Campaign tables. Tables III and IV summarize the settlement anchors and the positive-run measurements. Table III identifies the Sepolia deployment, the sender, the contracts, the checkpoint interface, and the publication range (*tx* denotes transaction throughout). Table IV reports proof time, the proof-to-confirmed-record delay Δ, and dry-run latency; each bucket has 10 primary and 5 secondary runs. Proof time stays roughly flat across the workload range because the zkVM's fixed overhead dominates per-block cost at this scale. In the completed-path rows, the proof verifies, the live interface accepts the call before publication, and the matching record becomes readable after confirmation.

Host topology. The campaigns run on on-demand Google Cloud (GCP) Linux hosts: a primary `n2-standard-32` (32 vCPU, 128 GB RAM) and a secondary `n2-standard-16` (16 vCPU, 64 GB RAM). The positive campaign and both fault campaigns share the primary VM, running sequentially. A separate control workstation handles orchestration, packaging, archiving, and aggregation, outside the measured path.

Positive regime. For positive runs, the locally verified proof artifact and its public inputs determine the checkpoint call under audit. C is sampled with stored Sepolia block tags captured before the matching confirmed-record read. After publication, replay protection can flip the same call from accept to reject, so the completed-path regime uses $C_T^{*} = 1$ instead of endpoint acceptance. Let t_V be the first verified-proof observation and t_K the first matching checkpoint read after $k_{\mathrm{conf}}{=}2$ confirmations. The reported delay is

$$\Delta = t_K - t_V.$$

The experiment reports proof generation time, Δ, and dry-run cost.

VI. Fault Injection Results

Experimental perimeter. The perimeter is one deployment instance with a fixed checkpoint path, confirmation threshold, and declared checkpoint domain. The live Sepolia evaluation diagnoses interface state from proof-log entries, checkpoint-path reads, and sampled block tags; production deployments can add multi-client dry-runs pinned to confirmed blocks.

Window policy. The fault campaigns target the two unpublished C-dependent regimes: publication stall and acceptance gap. Each window opens at proof completion and lasts 300 s, with one `eth_call` sample every 30 s. Each injection is repeated three times per bucket, and we record endpoint and history evidence for every mapped key. We report each fault with the signature vector

$$\sigma(a_{i,b}) \triangleq \big(V_T^{*}, C_T^{*}, R_T^{C}, C_T^{\mathrm{end}}, K_T^{*}\big).$$

Publication stall. This regime suppresses the checkpoint relay after proof completion for each bucket $b \in B$, keeping the proof path and the call aligned with the live interface. Every mapped key $a_{i,b}$ has

$$\sigma(a_{i,b}) = (1, 1, 0, 1, 0).$$

The call stayed checkpoint-admissible but publication was suppressed, matching the publication-stall row in Table II.

[2]Reth Execution Extensions (ExEx)
[3]OpenVM

TABLE III
SEPOLIA DEPLOYMENT ANCHORS

Scope	Anchor	Value
CAMPAIGN	Runs	75 positive Sepolia publications; 30 unpublished fault windows.
	Chains	source=1337; Sepolia=11155111; k_{conf}=2; RPC=`publicnode-sepolia`.
SENDER	Publisher	`0x93Aff92F79BD03807a26bE06D52A0790D68F79EE`; deployment and positive-publication sender; value 0 wei.
MANAGER	Contract	***CheckpointManager:*** `0x65A52C787B2153EA8fe980f93c5569023eE9357f`.
	Deploy tx	`0xd5dc60f2556fe44659a7b9aee79067e2c872df1b3ce0422670ebc44e1e269b7d`; block 10830148; status 1.
VERIFIER	Contract	***OpenVmHalo2Verifier:*** `0xB17fD1AE3604B296aC485bC1BcdC526671633b2a`; called by manager.
	Deploy tx	`0xa6812bdee043fd33ace5cf9d244398d8cab8d5874fbf2ea5c06f268dfe69e9e9`; block 10830147; status 1.
INTERFACE	Call/read	***publishCheckpoint***; selector `0xae8dd7f3`; confirmed reads ***checkpointExists/getCheckpoint***.
EVIDENCE	Publications	75 tx hashes; blocks 10830327–10845913; mined 2026-05-11 02:31:24–2026-05-13 15:52:36 UTC.
FOLLOW-UP	Publications	5 follow-up txs, all status 1, k_{conf}=2.
		Transient-gap recoveries: `0x384ff01ac79fdfa9c55749cf368c4e35ef16eaa387acd421c687f955fcc4c629`, `0x9df3182f5f01013d4bf64018e2ecfdd02ec480bbbb86f1462bdf8858badd5eb8`, `0x153e76b1ede6cc2049193b8dea62ddee3c28774c6b47297d2e0151ee22ed1b0e`.
		Proof-log gap at key 50000: `0xdf015dd7c9574c3e7689e35d1a88df41668ad65c2204783fb28c45be63628e5f`.
		Scope mismatch at key 60000: `0x7cea3a5573db71d9d20219e19d55a8cff16eab3fa7d7104bdfb61d45d37a33bc`.

TABLE IV
POSITIVE CAMPAIGN METRICS

b	n_p/n_s	$\tau_{\mathrm{prove},p}$ **s**	$\tau_{\mathrm{prove},s}$ **s**	Δ_p **s**	Δ_s **s**	$\tau_{\mathrm{dry},p}/\tau_{\mathrm{dry},s}$ **ms**
10	10/5	1330.5 [1302.1,1359.0]	2875.3 [2822.2,2928.4]	70.9	83.6	160.8/121.5
25	10/5	1338.0 [1322.6,1353.5]	2845.1 [2814.4,2875.7]	77.3	77.3	144.7/155.6
50	10/5	1344.9 [1324.1,1365.8]	2888.9 [2865.8,2911.9]	67.7	71.2	122.4/156.1
75	10/5	1332.9 [1320.0,1345.8]	2868.3 [2833.9,2902.8]	74.0	64.7	109.4/142.2
100	10/5	1364.3 [1342.9,1385.8]	2894.8 [2875.6,2913.9]	80.2	83.6	125.0/169.5

Notes. Proof columns give mean [95% CI]. $\Delta = t_K - t_V$.

Acceptance gap. We inject this regime via a stale publisher descriptor (payload mismatch). The proof is valid, but D^{pub} uses legacy call metadata while the live interface expects D^{decl}. The audit tests $\mathsf{Call}_{D^{\mathrm{pub}}}$ because that is the publisher-configured call, and the live manager rejects it. Every mapped key $a_{i,b}$ has

$$\sigma(a_{i,b}) = (1, 0, 1, 0, 0).$$

This is descriptor/interface divergence; the proof itself remains valid.

Transient gap. A targeted live follow-up exercises a selector-drift fault on the same Sepolia deployment. During a $\sim$300 s window the reconstructor uses a drifted selector that the manager rejects; rollback restores acceptance and the canonical call publishes. Three sequential runs at bucket b=10 yield

$$\sigma(a_{i,b}) = (1, 1, 1, 0, 0),$$

matching the transient-gap row of Table II.

Proof-log gap. A targeted live follow-up at key a=50000 publishes a valid checkpoint and re-audits with the proof-log view masked: V_T^*=0 and K_T^*=1, matching the proof-log-gap row of Table II.

Scope mismatch. A targeted live follow-up publishes a valid checkpoint at key a=60000 under a declared domain H={61000}, so $E(a)$=0 and Table II fires the scope-mismatch row.

Injected signatures. The publication-stall and acceptance-gap signatures appear in every bucket, with three runs per bucket. Transient gap, proof-log gap, and scope mismatch are each exercised in dedicated live follow-ups on the same deployment. For the controlled injections, all mapped keys are eligible and use `actual_height`, and every injection uses the same 300 s window and 30 s cadence, so the diagnosis follows the interface condition, isolating it from the proving workload.

Diagnosis. Each signature routes deterministically to one responsibility boundary: *publication stall* (endpoint C=1, K^*=0) to the publisher path; *acceptance gap* (C^*=0 with reproducible rejection) to the acceptance path; *transient gap* (C accepts then rejects within the window) to interface state change; *proof-log gap* (V_T^*=0, K_T^*=1) to monitor evidence repair; *scope mismatch* (E=0) to scope/policy review.

Sampling and diagnosis metrics. Across the bucketed runs, the monitor takes approximately T/δ valid C samples

per window on the published cadence (T=300 s, δ=30 s), with per-sample `eth_call` latency centred at the values reported in Table IV. Under this cadence, the classifier updates after the next valid C sample, so the diagnosis latency is bounded by the sampling interval plus RPC latency.

Takeaway. Batching, checkpoint-call re-encoding, authorization rotation, and pause controls can change the relation between proof availability, interface acceptance, and publication, but the same loop still applies: the monitor records the claim, reconstructs the call, samples the live interface, and reads the checkpoint record. Operationally, the diagnosis routes the incident to interface repair, publisher operation, or evidence capture.

VII. Scope and External Validity

Evaluation perimeter. The experiment focuses on the public checkpoint boundary in a proof-backed deployment. It tests whether proof evidence, interface acceptance, and confirmed records separate cleanly in a live checkpoint path. Data-availability economics, dispute games, incentives, bridge liquidity, and adversarial relay markets belong to adjacent studies.

Semantics of C. A positive C sample means interface admissibility at the sampled state; transaction inclusion is a separate matter. Gas, nonce, funding, fees, mempool policy, reorgs, timestamp guards, and later role changes belong to publication liveness. The endpoint sample reduces ambiguity inside the window; races after the window closes are tracked by publication-liveness follow-up.

Regime coverage. All six regimes of Table II are exercised end-to-end on Sepolia: completed path (75 positive publications), publication stall and acceptance gap (15 fault injections each), transient gap (3 selector-drift follow-up runs), and single-run live follow-ups for proof-log gap and scope mismatch.

Selective anchoring. Selective anchoring publishes only a sparse subset of the proof domain on chain; it is an operational concern adjacent to the classifier. A deployment keeps local proof and dry-run evidence for the broader domain, and proof-only keys become checkpoint-domain entries once explicitly promoted.

VIII. Conclusion

Result. In this paper, we introduced *checkpointability*, a runtime signal that lives between proof availability and checkpoint publication. It asks whether the live checkpoint interface will accept the call prepared for a proved claim.

The E–V–C–K decomposition turns a single ambiguous symptom (proof verified, record absent) into a four-signal audit space. Its six regimes give the operator deterministic incident routing: each incident lands at a responsibility boundary on the proof-to-record path.

In a live Sepolia campaign, adding C separated 30/30 fault windows that a (V, K)-only baseline conflates.

This signal is publicly verifiable: any party with the proof artifact can run the same dry-run, independent of the publisher. It is also forward-looking: the answer it returns is about the next call, before any record anchors on chain.

For bridge and rollup operators, the signal turns hindsight into foresight. For network observers, it surfaces the publisher's own dry-run as a public reading. For the broader ecosystem, it adds a third publicly reproducible signal on the checkpoint boundary.

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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.