

Inclusion and Exclusion Guarantees

Ritual

April 20, 2026

Abstract

Inclusion guarantees promise that a designated transaction will be present whenever a state predicate is true. The proposer cannot censor the transaction without producing an invalid block that the consensus protocol rejects. This is the strongest form of censorship resistance available: consensus-level block rejection, not economic incentive or social norm. Exclusion guarantees promise that certain transactions will not be present. Exclusion guarantees enable application-level safety controls but introduce a potential griefing vector (malicious exclusion predicates). Inclusion and exclusion predicates can conflict: an inclusion trigger may mandate inclusion of a transaction that an exclusion trigger mandates exclusion of.

1 Introduction

Symphony [15] is an execution-aware SMR framework organized around a single principle: *protocol-enforced guarantees*.

The set of blocks that a proposer can build is constrained in order to provide guarantees on *inclusion*, *exclusion*, and *sequencing* to applications and users.

Block proposers in existing protocols exercise several distinct powers that are often treated as atomic: which transactions to include, which to exclude, how to order them, and when to propose. Outside of basic validity checks, the proposer has full power to determine the contents of the block. It can censor a transaction by omitting it, and users and applications have no protocol-level way to keep a transaction out of a block: a user may want a DEX trade not to be included if the price is outside a desirable range, and a trading application may want a circuit breaker when oracle staleness exceeds a threshold. Smart contracts can implement their own rate limiting, but this consumes gas and block space even for reverted transactions.

Inclusion lists (EIP-7547 [4]) constrain the exclusion rights of proposers via forward-looking enforcement: the next proposer’s block is invalid if the current proposer’s inclusion list is not satisfied. This is validity-level enforcement, but with a one-block delay and applied to the successor rather than the originating proposer.

This paper makes the following contributions:

- Inclusion guarantees (Section 3.1) and exclusion guarantees (Section 3.2) are enforced in-protocol via the `InclusionComplete` and `ExclusionCompliant` conjuncts of the extended block validity function (Section 2.3).
- Conflict detection at registration time (Section 3.3) rejects a trigger only if it conflicts with an active trigger at the current state; conflicts that arise later are resolved at runtime by the resolution policy carried by the exclusion trigger.
- Under the default resolution policy, and assuming the mandated transactions fit within the block and satisfy the remaining conjuncts, there exists a computationally efficient procedure for producing a valid block under active inclusion/exclusion triggers (Section 4).

Section 2 establishes the model: system model, extended state, extended block validity, predictable validity, and block construction feasibility. Section 3 instantiates inclusion and exclusion guarantees. Section 4 establishes liveness. Section 5 discusses limitations. Section 6 reviews related work. Section 7 concludes.

2 Model

2.1 System Model

We consider n stake-weighted replicas (validators) running a *state machine replication (SMR)* protocol [9] under partial synchrony [3]: after an unknown Global Stabilization Time (GST), messages between correct replicas are delivered within an unknown bound Δ . At most $f < n/3$ replicas are Byzantine; replica i holds stake $w_i > 0$, the total stake is $W = \sum_{i=1}^n w_i$, Byzantine replicas control less than $W/3$, and a set of replicas holding more than $2W/3$ is a *quorum*. SMR guarantees that all correct replicas deliver the same sequence of transactions and that the resulting state is uniquely determined by applying a deterministic state transition function $\text{STF} : \mathcal{S} \times \mathcal{T} \rightarrow \mathcal{S}$ to the initial state in that order, where $\mathcal{S}$ is the state space and $\mathcal{T}$ the set of transactions.

The proposer is untrusted. Outside the checks imposed by the block validity function (Section 2.3), a proposer has full power to determine the contents of the block, and a Byzantine proposer may include, exclude, and order transactions arbitrarily. Enforcement of a guarantee must therefore be Byzantine fault-tolerant: it holds because correct validators reject blocks that violate it.

2.2 Extended State and Guarantees

Symphony extends the standard SMR model [9] with additional state components:

Definition 2.1 (Symphony State). *The state S consists of:*

- *Execution state S_{exec} : account balances, contract storage (standard).*
- *Trigger pools: a set of active inclusion triggers $\mathcal{I} = \{(P_i, T_i, \text{TTL}_i)\}$ and exclusion triggers $\mathcal{E} = \{(Q_j, \text{TTL}_j)\}$, where each Q_j is scoped to transactions targeting the registering contract and carries a resolution policy for conflicts with inclusion triggers: “inclusion overrides exclusion” (the default), “exclusion overrides inclusion”, or “reject block” (Section 3.3).*
- *Ordering policies: a map from contract addresses to ordering policies $\{(\pi_C, \{\tau_i\})\}_{C \in \mathcal{C}}$, where $\mathcal{C}$ is the set of contracts with a registered ordering policy [14].*

An inclusion or exclusion trigger is *active* from its registration until its TTL expires, and an ordering policy is active until it is deregistered; each active trigger or ordering policy is an *active guarantee*.

2.3 Extended Block Validity

Definition 2.2 (Extended Block Validity). *A block B proposed for state S is valid if and only if all of the following conjuncts hold:*

$$\text{Valid}(B, S) = \text{Structural}(B) \tag{1}$$

$$\wedge \text{Execution}(B, S) \tag{2}$$

$$\wedge \text{InclusionComplete}(B, S) \tag{3}$$

$$\wedge \text{ExclusionCompliant}(B, S) \tag{4}$$

$$\wedge \text{OrderingCompliant}(B, S) \tag{5}$$

The first two conjuncts are standard in any BFT protocol:

- **Structural(B)**: the block is well-formed, all transaction signatures are valid, the parent hash is correct, and the proposer is the designated leader for this round.
- **Execution(B, S)**: the proposed post-state S' is the correct result of applying the block's transactions to S . For replicated transactions, every correct validator re-executes and checks.

The final three conjuncts are Symphony's extensions:

- **InclusionComplete(B, S)**: for every active inclusion trigger $(P_i, T_i, \text{TTL}_i) \in \mathcal{I}$ such that $P_i(S) = 1$, the transaction T_i appears in B (subject to the resolution policy of Section 3.3). Section 3.1 defines the semantics.
- **ExclusionCompliant(B, S)**: for every active exclusion trigger $(Q_j, \text{TTL}_j) \in \mathcal{E}$, no transaction $T \in B$ satisfies $Q_j(S, T) = 1$ (subject to the resolution policy of Section 3.3). Section 3.2 defines the semantics.
- **OrderingCompliant(B, S)**: for every contract C with a registered ordering policy $(\pi_C, \{\tau_i\})$, the subsequence of transactions in B that target C appears in the order specified by π_C , a priority list of recognized call sequences over the functions of C , and the tiebreakers τ_i evaluated on S . The semantics are defined in [14].

All three guarantees follow the same structure. Each is implemented as a conjunct in Definition 2.2:

1. A beneficiary (user, application, or the protocol itself) registers a guarantee by publishing a predicate on-chain. This registers a trigger in $\mathcal{I}$, $\mathcal{E}$, or $\{(\pi_C, \{\tau_i\})\}$.
2. The predicate becomes a conjunct of the block validity function.
3. Every block proposed by a correct proposer must satisfy all active conjuncts.
4. Every correct validator independently evaluates all active conjuncts and rejects blocks that violate any. This deterministic evaluation is the default and the basis for Symphony's BFT safety argument.
5. The proposer retains only the *residual* discretion: the freedom to act that is not constrained by active guarantees.

2.4 Predictable Validity

Extended external validity requires that the block proposer and all correct validators agree on the state against which validity predicates are evaluated.

Definition 2.3 (Predictable Validity). *An SMR protocol provides predictable validity if, whenever a correct proposer constructs a block B , there is a unique state S (the state the proposer is building on), derived from the committed prefix of the ledger in the proposer's view, such that the proposer can compute S from its local information, every correct validator that receives B can verify that S is the correct pre-state and that B is valid when applied to S , and every predicate $p : \mathcal{S} \rightarrow \{0, 1\}$ yields the same value $p(S)$ for the proposer and for all correct validators.*

Proposition 2.4 (Predictable Validity Requirement [13]). *If an SMR protocol does not provide predictable validity because a correct proposer and a correct validator can derive different states from their respective local information, then there exist a state predicate p used in the block validity function and executions in which a correct proposer constructs a block that it considers valid while a correct validator considers the block invalid, or vice versa.*

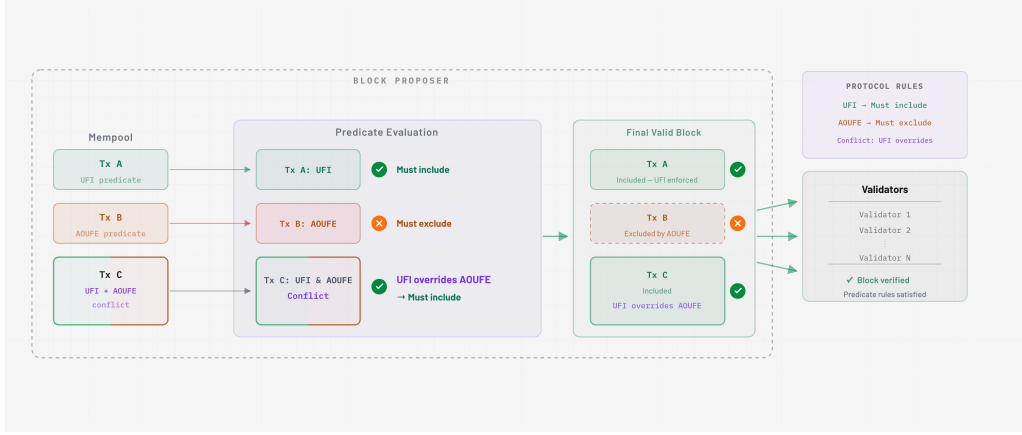


Figure 1: Inclusion and exclusion enforcement pipeline (in the figure, UFI denotes an inclusion trigger and AOUFE an exclusion trigger). The proposer evaluates mempool transactions against active inclusion triggers and exclusion triggers. By default, inclusion mandates take precedence over exclusion to prevent censorship vectors.

The proof [13] exhibits a predicate that distinguishes the states derived from the proposer’s and the validator’s local information. Single-leader protocols such as CometBFT [1] and Hot-Stuff [11] provide predictable validity.

2.5 Block Construction Feasibility

Let $g_1, \dots, g_k$ be the active guarantees (inclusion triggers, exclusion triggers, and ordering policies). Let Δ_B be the block construction time budget: the maximum time a correct proposer has to construct a valid block before the round times out and the protocol moves to the next proposer.

Each active guarantee imposes two costs on the proposer:

1. *Evaluation cost*: the proposer must evaluate the predicate $P_i(S)$ or $Q_j(S, T)$ for each relevant transaction. Let c_i be the computational cost of evaluating the i -th predicate.
2. *Constraint cost*: each satisfied predicate constrains the set of valid blocks. Let $\mathcal{B}_i(S)$ be the set of blocks valid under the conjunct contributed by g_i . The set of valid blocks under all conjuncts is $\mathcal{B}(S) = \bigcap_{i=1}^k \mathcal{B}_i(S)$.

Definition 2.5 (Feasibility). *A state S with active guarantees $g_1, \dots, g_k$ is feasible if $\mathcal{B}(S) \neq \emptyset$: there exists at least one valid block.*

Proposition 2.6 (Infeasibility from Conflicting Guarantees). *There exist sets of active guarantees that render a state infeasible. In particular, if an inclusion trigger requires inclusion of transaction T and an exclusion trigger requires exclusion of T , both predicates are satisfied, and the exclusion trigger was registered with the “reject block” resolution policy (Section 3.3), then $\mathcal{B}(S) = \emptyset$.*

Proof. The inclusion conjunct requires $T \in B$, and the exclusion conjunct requires $T \notin B$. No block can satisfy both simultaneously. $\square$

3 Inclusion and Exclusion Guarantees

This section instantiates two of the three guarantees from [13]: inclusion and exclusion. These guarantees are enforced in-protocol via the `InclusionComplete` and `ExclusionCompliant` conjuncts of the extended block validity function (Definition 2.2); Figure 1 shows the enforcement pipeline.

3.1 Inclusion Guarantees

Definition 3.1. An inclusion trigger is a triple (P, T, TTL) registered in the chain state:

- $P : \mathcal{S} \rightarrow \{0, 1\}$ is a state predicate.
- T is a transaction.
- $TTL \in \mathbb{N}$ is the time-to-live in blocks.

Definition 3.2 (Enforcement of inclusion guarantees). A block B proposed for state S satisfies the *InclusionComplete* conjunct if and only if: for every active trigger $(P_i, T_i, TTL_i) \in \mathcal{I}$ such that $P_i(S) = 1$ and the trigger has not expired, T_i appears in B , unless T_i is matched by an active exclusion trigger registered with the “exclusion overrides inclusion” policy (Section 3.3).

A block that omits a triggered transaction is invalid. Every correct validator rejects it. The proposer cannot censor the transaction without producing an invalid block that the consensus protocol rejects. This is the strongest form of censorship resistance available: consensus-level block rejection, not economic incentive or social norm. The cost of evaluating inclusion predicates in the block construction path, the existence of valid blocks under active triggers, and the efficiency of finding a valid block (even without revenue maximization) are analyzed in Section 4.

3.1.1 Comparison to Existing Mechanisms

Inclusion lists (EIP-7547) [4]. Ethereum’s inclusion lists provide validity-level enforcement with a one-block delay: the next proposer’s block is invalid if the current proposer’s inclusion list is not satisfied. Inclusion guarantees differ in two respects. First, inclusion guarantees provide same-block enforcement for every transaction after the initial registration: once a trigger is registered, the proposer’s own block is invalid if the triggered transaction is absent, with no reliance on the next proposer’s compliance. Second, inclusion guarantees are non-interactive after registration: The user registers a trigger once and does not need to be online or submit new lists each slot for the CR guarantee to hold. Inclusion lists are just-in-time (reactive, per-slot); inclusion triggers are registered in advance (proactive, persistent until TTL expiry). The initial registration transaction itself has the same CR properties as any other transaction—it depends on the underlying censorship resistance of the base protocol. The distinction is in what happens after registration: Inclusion guarantees provide persistent, non-interactive censorship-resistance for every subsequent triggering; inclusion lists require active, per-slot participation.

Chainlink Keepers. External actors are incentivized to execute maintenance transactions. There is no protocol guarantee of inclusion, merely a reliance on economic incentives and the willingness of keepers to submit transactions. Inclusion guarantees provide protocol-level guarantees independent of external actor incentives.

SQL triggers and active databases [2, 8]. The closest conceptual ancestor. Event-Condition-Action (ECA) rules in active databases fire automatically when a condition holds. Symphony’s inclusion triggers are ECA rules enshrined at the consensus level: the “event” is a state change, the “condition” is the predicate P , and the “action” is forced inclusion of T . The trust model differs fundamentally: in an active database, the DBMS is trusted; in a blockchain, the proposer is untrusted, and enforcement must be Byzantine fault-tolerant.

3.2 Exclusion Guarantees

Exclusion guarantees are the converse of inclusion guarantees.

Definition 3.3. *An exclusion trigger registered by contract C is a pair (Q, TTL) where:*

- $Q : \mathcal{S} \times \mathcal{T} \rightarrow \{0, 1\}$ *is a predicate over state and transactions.*
- $TTL \in \mathbb{N}$ *is the time-to-live in blocks. The trigger also carries the resolution policy declared for conflicts with inclusion triggers (Section 3.3).*

Scope restriction. Q may only evaluate to 1 for transactions that target the registering contract C . An exclusion trigger cannot exclude transactions that do not interact with C . Exclusion triggers registered by the protocol itself are exempt from this restriction.

Definition 3.4 (Enforcement of exclusion guarantees). *A block B proposed for state S satisfies the `ExclusionCompliant` conjunct if and only if: for every active trigger $(Q_j, TTL_j) \in \mathcal{E}$ and for every transaction $T \in B$, $Q_j(S, T) = 0$, except for a transaction T_i mandated by an active inclusion trigger with $P_i(S) = 1$ when (Q_j, TTL_j) was registered with the default resolution policy (inclusion overrides exclusion, Section 3.3).*

Equivalently: no transaction in B satisfies any active exclusion predicate, other than inclusion-mandated transactions exempted by the resolution policy. A block containing such a transaction is invalid. As with inclusion guarantees, the cost of evaluating exclusion predicates, block building complexity under active exclusion triggers, and the existence of computationally efficient strategies for finding a valid block are analyzed in Section 4.

The simplest user-defined exclusion guarantee would be attached to a single transaction (represented as Q only ever being equal to 1 when its second argument is a specific transaction). For example, a user might want to include an exclusion predicate dictating that a DEX trade won't be included if the price is below a threshold. Note that slippage-based transactions *can* be included onchain even though they revert, whereas here, the transaction is not includable and therefore does not pay any gas. Exclusion guarantees can be more general than per-transaction predicates, and can allow applications or the protocol itself to specify exclusion triggers over sets of transactions.

Comparison to inclusion predicates. Inclusion predicates have signature $P : \mathcal{S} \rightarrow \{0, 1\}$ (state-only); the transaction to include is pre-specified at registration time. Exclusion predicates have signature $Q : \mathcal{S} \times \mathcal{T} \rightarrow \{0, 1\}$ (state and transaction); they must inspect each candidate transaction because the set of transactions to exclude is not known in advance. This asymmetry is fundamental: Inclusion triggers know *what* to include and check *when*; exclusion triggers check *whether* each incoming transaction should be blocked, given the current state.

Limitations of the single-transaction signature. $Q(S, T)$ evaluates one transaction at a time. Multi-transaction relational constraints (e.g., “transaction T_1 must not appear within k positions of T_2 ”) would require extending the predicate domain to reference the full block transaction set. The current signature cannot express relative-position constraints between transactions.

3.2.1 Use Cases

Exclusion guarantees enable a class of application-level safety controls that are not available in any existing blockchain protocol:

Application circuit breakers. A DeFi contract registers an exclusion trigger: “exclude all transactions targeting me when the oracle staleness exceeds 60 blocks.” The predicate is $Q(S, T) = (S.\text{blockHeight} - S.\text{oracleLastUpdated}[C] > 60)$, where C is the registering contract (the scope restriction ensures Q is only evaluated for transactions targeting C). During oracle downtime, the contract is automatically protected from stale-data trades. Staleness is measured in blocks rather than wall-clock time, since block height is the only quantity with a strongly consistent view across all validators.

Anti-sandwich constraints. Preventing sandwich attacks (“transaction T_1 must not appear within k positions of T_2 ”) requires relative-position constraints between transactions, which exceeds the single-transaction signature $Q(S, T)$. This use case would require extending exclusion triggers to predicates over the block’s transaction set, or is better addressed by ordering constraints or the lazy evaluation pattern [14].

Quarantine during proof system degradation. When a proof system enters degraded mode [12], exclusion triggers automatically exclude transactions that consume results verified only by the degraded system. This is a protocol-level quarantine that prevents downstream contamination from potentially-unsound results.

Protocol-level invariant enforcement. The protocol itself can register exclusion triggers to enforce safety invariants: exclude transactions that would create more than N pending delegated computations (preventing resource exhaustion) or exclude transactions from sanctioned addresses (compliance).

3.2.2 Comparison to Existing Mechanisms

Transaction filtering (OFAC compliance in Ethereum). Some Ethereum builders voluntarily filter transactions from sanctioned addresses. This is not enforced at the protocol level; non-filtering builders can include any transaction. Exclusion guarantees are a protocol-enforced property, independent of builder choice.

MEV protection services (Flashbots Protect) [5]. Private transaction channels provide two properties: (a) information hiding (the user’s trade intent is not visible in the public mempool, preventing front-running based on observed intent) and (b) builder commitment to not sandwich protected transactions. Exclusion guarantees address (b) only in part: an exclusion predicate can keep a transaction out of the block when a single-transaction condition holds (e.g., a price bound), without trusting the builder, but it cannot express sandwich patterns, which require relative-position constraints between transactions (Section 3.2). Exclusion guarantees do not address (a): the user’s transaction is still visible in the public mempool. Information hiding requires encrypted mempools [10] or private relay channels, which are complementary to exclusion guarantees rather than replaced by them.

Rate limiting and access control. Smart contracts can implement their own rate limiting (e.g., “revert if called more than N times per block”), but this consumes gas and block space even for reverted transactions. Current approaches to on-chain access control on L2s use optimistic execution patterns that similarly consume resources on failure [7]. Exclusion guarantees prevent the transaction from entering the block in the first place, saving gas and block space. The exclusion check happens during block construction, before execution, so excluded transactions impose no execution cost on the network.

3.3 Interaction between inclusion and exclusion

Inclusion and exclusion predicates can conflict: an inclusion trigger may mandate inclusion of a transaction that an exclusion trigger mandates exclusion of. Proposition 2.6 (Section 2.5) shows that conflicting triggers can render block construction infeasible.

3.3.1 Conflict Detection

At registration time, the protocol checks for potential conflicts between the new trigger and all active triggers. The check is conservative: if the predicate domains of an inclusion trigger and an exclusion trigger overlap (they reference the same transaction specification or the same contract), a potential conflict is flagged.

Static analysis. For predicates with declared dependency sets, the protocol checks whether the state-slot dependencies overlap. If the inclusion predicate P and the exclusion predicate Q read disjoint state slots, they cannot conflict regardless of the state.

Registration-time evaluation. The protocol evaluates both predicates at the current state. If $P(S) = 1$ and $Q(S, T) = 1$ for the inclusion-specified transaction T , an immediate conflict exists and the registration is rejected. This is the registration rule used throughout this paper: a new trigger is rejected if and only if it conflicts with an active trigger of the opposite kind at the current state; every other registration is admitted, including pairs flagged by static analysis, and conflicts that arise later are resolved at runtime by the resolution policy of the exclusion trigger (Section 3.3).

Limitations. Static analysis is sound but not complete for predicates with declared dependency sets: a pair it does not flag (disjoint dependency sets) cannot conflict, but it may flag pairs that never conflict (false positives); a flagged pair is admitted unless it conflicts at the current state, so conflicts can still arise at runtime. Full conflict detection (whether there exists *any* future state S' such that $P(S') = 1$ and $Q(S', T) = 1$) is undecidable in general (by reduction from satisfiability of state predicates over a Turing-complete state machine). This is formally related to the safety analysis of access control policies [6], where determining whether a protection system can reach an unsafe state is undecidable in the general case.

3.3.2 Runtime Conflict Resolution

If a conflict arises at runtime (due to a state change that simultaneously triggers inclusion and exclusion predicates on the same transaction), the protocol applies a resolution policy.

1. **Default: inclusion overrides exclusion.** The transaction is included. An inclusion trigger is enshrined in the finalized chain state: the chain has already committed to including T_i when $P_i(S) = 1$. This is a consensus-level mandate, not a discretionary inclusion. Exclusion guarantees, by contrast, constrain the proposer’s discretionary inclusion power over mempool transactions. An inclusion-mandated transaction is not in the proposer’s discretionary set, so exclusion guarantees do not apply to it. The conflicting exclusion trigger remains active for transactions matching its pattern without an active inclusion trigger.
2. **Configurable.** The resolution policy can be overridden at registration time: “exclusion overrides inclusion” (prioritize exclusion in specific safety-critical contexts, e.g., quarantine during proof system degradation) or “reject block” (force the proposer to wait for a non-conflicting state). The choice is application-specific and must be declared when the conflicting exclusion trigger is registered.

Mechanism-based resolution (future work). The precedence rules above are static: they are declared at registration time and applied deterministically. The alternative is via a mechanism design framework, whether an auction or a contractual agreement between a prespecified set of actors on how to handle conflict resolution.

Why inclusion takes precedence by default. The purpose of exclusion guarantees is to prevent the proposer from including harmful transactions. But a transaction mandated by an inclusion guarantee was explicitly registered by a user or the protocol as a consensus-level obligation. Allowing an exclusion guarantee to override it would create a censorship vector: a malicious application developer that controls the exclusion constraints *on a contract called by* the victim transaction could register an exclusion predicate overlapping with a victim’s inclusion trigger to neutralize forced inclusion indefinitely. With inclusion-overrides-exclusion as the default, this attack is not possible. Applications that genuinely need exclusion to override inclusion (e.g., emergency quarantine) must opt into that policy explicitly at registration time, accepting the censorship risk.

3.4 Integration with the Verification Lattice

Inclusion and exclusion guarantees interact with the product lattice [12] in two directions:

Lattice promotions as inclusion triggers. When a delegated computation result advances to a new lattice position (e.g., SNARK verification completes, advancing its verification state, the vector recording which proof systems have verified the result [12], from $(1,0)$ to $(1,1)$), an inclusion trigger can fire: “when result R reaches lattice position $(1,1)$, include transaction T that consumes R .” This chains verification completion with downstream execution.

Proof system degradation as exclusion triggers. When a proof system enters degraded mode, exclusion triggers automatically activate: “exclude transactions that consume results verified only by the degraded system.” This is a state-dependent exclusion guarantee whose predicate references the proof system’s operational status in the chain state.

These interactions demonstrate the composability of guarantees: inclusion and exclusion can be applied concurrently and in support of other system components (the lattice, the degradation protocol) through the same predicate-based mechanism.

4 Liveness

Liveness means that every correct transaction eventually makes progress.

Forced inclusion and exclusion predicates are evaluated in the consensus critical path. Under the protocol cap $k_{\max}$ on the number of simultaneously active inclusion/exclusion triggers, the total evaluation cost is bounded: $\sum_{i=1}^k c_i \leq k_{\max} \cdot c_{\max} \leq \alpha \cdot \Delta_B$, where $c_{\max}$ is the maximum per-predicate evaluation cost (enforced by a registration deposit that is slashed if the predicate is malformed or excessively expensive to evaluate) and $\alpha < 1$ is a configurable fraction of the block construction budget Δ_B . Since predicates are independent, evaluation parallelizes across N_{par} cores [13].

Block construction under inclusion and exclusion enforcement is efficient. Given the set of active triggers and their evaluation results, the proposer constructs a valid block as follows: (a) include all transactions mandated by triggered inclusion predicates, (b) exclude all transactions matched by active exclusion triggers, (c) if a transaction is mandated by an inclusion trigger and matched by an exclusion trigger, apply the conflict resolution policy (Section 3.3)—by default, inclusion takes precedence. Each step is linear in the number of active

triggers. The conflict resolution policy is a static priority rule evaluated in $O(1)$ per conflict, so runtime conflict resolution does not add combinatorial complexity. Under the default and “exclusion overrides inclusion” policies, and assuming the mandated transactions fit within the block and satisfy the remaining conjuncts, this guarantees the existence of a computationally efficient procedure for producing a valid block under active inclusion/exclusion triggers. We make no claim about the *economic* efficiency of the resulting block—revenue maximization under guarantee constraints is a separate mechanism design problem, orthogonal to liveness, and is the subject of ongoing work.

Conflict detection and liveness. Conflict detection at registration time (Section 3.3) rejects a registration only if it conflicts with an active trigger at the current state; static analysis may flag pairs that never conflict (false positives), and flagged pairs that do not conflict at registration are admitted. Conflicts that arise later are resolved at runtime by the policy carried by the exclusion trigger: under the default and “exclusion overrides inclusion” policies the block construction procedure above succeeds, while under “reject block” a conflicting state has no valid block until the state changes. Only the “reject block” policy, which an application must opt into, can cause a liveness failure.

5 Discussion

5.1 Limitations

Inclusion/exclusion conflict resolution. The interaction between forced inclusion and forced exclusion introduces a satisfiability problem (Proposition 2.6). Full conflict detection is undecidable in the general case. The practical approach (static analysis + registration-time evaluation + runtime resolution policy) rejects only registrations that conflict at the current state; conflicts that arise later are resolved at runtime by the declared policy.

6 Related Work

Event-Condition-Action (ECA) rules [2,8] in active databases are the direct ancestor of inclusion triggers. An ECA rule fires a stored procedure when a database state change satisfies a condition. SQL triggers implement this in relational databases.

The key difference is the trust model: in a database, the DBMS is trusted to evaluate conditions and fire actions. In a blockchain, the block proposer is untrusted, and enforcement must be Byzantine fault-tolerant. Symphony’s contribution is lifting ECA semantics to the SMR level via external validity [13], where enforcement is guaranteed by block rejection rather than DBMS integrity.

7 Conclusion

Inclusion and exclusion guarantees are enforced in-protocol via the `InclusionComplete` and `ExclusionCompliant` conjuncts of the extended block validity function. A block that omits a triggered transaction is invalid, so the proposer cannot censor the transaction without producing a block that the consensus protocol rejects; after registration, inclusion guarantees provide persistent, non-interactive censorship resistance for every subsequent triggering. Exclusion guarantees enable application-level safety controls, such as circuit breakers and quarantine during proof system degradation, and the exclusion check happens during block construction, before execution, so excluded transactions impose no execution cost on the network. A registration is rejected only if it conflicts with an active trigger at the current state, and conflicts that arise

later are resolved at runtime by the policy carried by the exclusion trigger, with inclusion overriding exclusion by default. Under the default policy, and assuming the mandated transactions fit within the block and satisfy the remaining conjuncts, a valid block can be constructed in time linear in the number of active triggers.

References

- [1] E. Buchman, J. Kwon, Z. Milosevic. The latest gossip on BFT consensus. arXiv:1807.04938, 2018.
- [2] U. Dayal, D. R. McCarthy. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record* 17(1):51–70, 1988.
- [3] C. Dwork, N. Lynch, L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM* 35(2):288–323, 1988.
- [4] Ethereum Foundation. EIP-7547: Inclusion lists. 2023.
- [5] Flashbots. 2021.
- [6] M. A. Harrison, W. L. Ruzzo, J. D. Ullman. Protection in operating systems. *Commun. ACM* 19(8):461–471, 1976.
- [7] A. Ozdaglar et al. Optimistic search on blockchains. arXiv:2506.14768, 2025.
- [8] N. W. Paton, O. Diaz. Active database systems. *ACM Comp. Surveys*, 31(1), 1999.
- [9] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.* 22(4):299–319, 1990.
- [10] Shutter Network. 2022.
- [11] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, I. Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proc. 2019 ACM Symp. Principles of Distributed Computing (PODC)*, pp. 347–356, 2019.
- [12] Ritual. Multi-System Verification Model. 2026.
- [13] Ritual. Extended External Validity. 2026.
- [14] Ritual. Application-Defined Ordering Guarantees. 2026.
- [15] Ritual. Symphony: Execution-Aware Consensus and Proposer Disaggregation. 2026.