

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. The practical approach (static analysis + registration-time evaluation + runtime resolution policy) is sound but not complete: it may reject non-conflicting registrations.

1 Introduction

Symphony [12] is an execution-aware SMR framework organized around a single principle: *protocol-enforced guarantees*.

Inclusion lists (EIP-7547 [2]) 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 [10].
- Conflict detection at registration time (Section 3.3) is sound but not complete: it may reject non-conflicting trigger registrations (false positives) but will never allow a registration that creates an unsatisfiable validity predicate (no false negatives).
- This guarantees the existence of a computationally efficient procedure for producing a valid block under active inclusion/exclusion triggers (Section 4).

Section 3 instantiates inclusion and exclusion guarantees. Section 4 establishes liveness. Section 5 discusses limitations and future work. Section 7 concludes.

2 Model

Symphony extends the standard SMR model [7] with additional state components:

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

- *Execution state S_{exec} : account balances, contract storage (standard).*

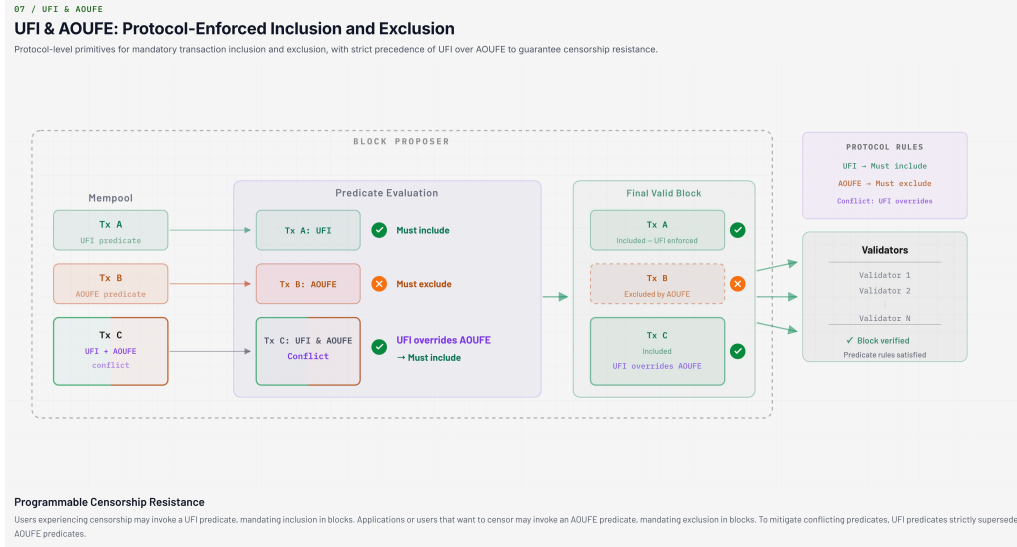


Figure 1: Inclusion and exclusion enforcement pipeline. The proposer evaluates mempool transactions against active inclusion triggers and exclusion triggers. Inclusion mandates take strict precedence over exclusion to prevent censorship vectors.

- *Trigger pools: a set of active inclusion triggers $\mathcal{I} = \{(P_i, T_i, TTL_i)\}$ and exclusion triggers $\mathcal{E} = \{(Q_j, TTL_j)\}$, where each Q_j is scoped to transactions targeting the registering contract.*

Let k be the number of 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 transfer 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 i -th conjunct. The set of valid blocks under all conjuncts is $\mathcal{B}(S) = \bigcap_{i=1}^k \mathcal{B}_i(S)$.

Definition 2.2 (Feasibility). *A state S with active transfers $\{T_1, \dots, T_k\}$ is feasible if $\mathcal{B}(S) \neq \emptyset$: there exists at least one valid block.*

Proposition 2.3 (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 , and both predicates are satisfied, 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 [10]: inclusion and exclusion. These guarantees are enforced in-protocol via the `InclusionComplete` and `ExclusionCompliant` conjuncts of the extended block validity function [10].

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 .

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 [10].

3.1.1 Comparison to Existing Mechanisms

Inclusion lists (EIP-7547) [2]. 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 [1, 6]. 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.

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 .

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 $T \in B$, $Q_j(S, T) = 0$.*

Equivalently: no transaction in B satisfies any active exclusion predicate. 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 [10].

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 [11].

Quarantine during proof system degradation. When a proof system enters degraded mode [9], 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), exclude transactions from sanctioned addresses (compliance), or exclude transactions that would exceed a contract’s per-block gas limit (DoS protection).

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) [3]. 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) at the protocol level: exclusion predicates can prevent sandwich-pattern transactions from entering the block, without trusting the builder. Exclusion guarantees do not address (a): the user’s transaction is still visible in the public mempool. Information hiding requires encrypted mempools [8] 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 [5]. 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 2.3 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.

Limitations. Static analysis is sound but not complete: it may reject non-conflicting registrations (false positives) but will not accept conflicting ones (no false negatives). 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 [4], 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 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 [9] 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 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 $k_{\max}$, 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 the deposit/slash mechanism). Since predicates are independent, evaluation parallelizes across P cores [10].

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. 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.

Soundness vs. completeness of conflict detection. Conflict detection at registration time (Section 3.3) is sound but not complete: it may reject non-conflicting trigger registrations (false positives) but will never allow a registration that creates an unsatisfiable validity predicate (no false negatives). Every set of active triggers admitted by the registration-time check has a valid block—the block construction procedure above always succeeds. The incompleteness is conservative: some safe trigger combinations are rejected, but no admitted combination causes 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 2.3). Full conflict detection is undecidable in the general case. The practical approach (static analysis + registration-time evaluation + runtime resolution policy) is sound but not complete: it may reject non-conflicting registrations.

6 Related Work

Event-Condition-Action (ECA) rules [1,6] 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 [10], where enforcement is guaranteed by block rejection rather than DBMS integrity.

7 Conclusion

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.

The exclusion check happens during block construction, before execution, so excluded transactions impose no execution cost on the network.

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.

References

- [1] U. Dayal, D. R. McCarthy. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record* 17(1):51–70, 1988.
- [2] Ethereum Foundation. EIP-7547: Inclusion lists. 2023.
- [3] Flashbots. 2021.
- [4] M. A. Harrison, W. L. Ruzzo, J. D. Ullman. Protection in operating systems. *Commun. ACM* 19(8):461–471, 1976.
- [5] A. Ozdaglar et al. Optimistic search on blockchains. arXiv:2506.14768, 2025.
- [6] N. W. Paton, O. Diaz. Active database systems. *ACM Comp. Surveys*, 31(1), 1999.
- [7] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.* 22(4):299–319, 1990.
- [8] Shutter Network. 2022.
- [9] Ritual. Multi-System Verification Model. 2026.

- [10] Ritual. Extended External Validity. 2026.
- [11] Ritual. Application-Defined Ordering Guarantees. 2026.
- [12] Ritual. Symphony: Execution-Aware Consensus and Proposer Disaggregation. 2026.