A Specification for Application-Preferred Execution Sequencing (APES) on Monad

Andrea Canidio, Andrei Constantinescu, Lioba Heimbach, and Haythem Sellami

Category Labs

Version 1

November 16, 2026

1 Motivation

Applications may require that transactions execute in a specific order: a PropAMM requires its oracle update to execute before any swap; an order-book exchange requires cancellations of limit orders before market orders; and an on-chain auction may require bids ordered from highest to lowest. However, the ordering of transactions is constrained by the underlying blockchain, so an application with ordering requirements must either find a chain whose default rule happens to match its needs or create its own app-chain (or rollup, or off-chain sequencer) and trade off composability.

To address this trade-off, we propose an upgrade to the Monad blockchain called Application-Preferred Execution Sequencing (APES). APES is based on an algorithm called *unanimity override* proposed and studied in [2]: whenever all applications that score two transactions agree on their relative order, the protocol respects that agreement unless the resulting unanimity relation has cycles; if cycles are present, a fallback rule breaks them through a process called *demotion*, and the default order (e.g., the priority-fee order) completes the resulting partial order into a total execution order. This creates an attack vector: by inserting transactions that create cycles, an attacker can obtain an execution order that favors it.

This document discusses the implementation details of APES, describes how it can be attacked, and derives the properties that survive those attacks. Throughout this document, we assume the Multiple Concurrent Proposers (MCP) architecture of Cadence, a recently proposed consensus protocol for Monad [1]. Under MCP, no single party can censor a transaction, which APES assumes. Furthermore, APES computes the execution order during translation, a new step that takes as input a consensus block (i.e., the output of the consensus protocol) and outputs a block that can be executed. MCP itself and other upcoming features, such as encrypted transactions, also rely on translation.

At a high level, the APES architecture has four parts:

1. Transactions that want to access one or more applications with declared ordering preferences must carry a static declaration, similar to Solana's per-transaction account lists. The declaration lists all applications with declared ordering preferences that the transaction may access and, for each, the methods and parameters these applications need in order to compute their preferences over the transaction (Section 2.1).

2. Applications express their preferences by registering them in a precompile. Preferences are represented by a function that produces a score for each transaction that has statically declared that application, where higher scores correspond to earlier execution. We say that an application with registered preferences is "opinionated" (Section 2.2).

3. The execution order is computed at translation from the scores assigned by each application with registered preferences to each transaction that declares it (Section 2.3).

4. The initial declaration of a transaction is checked against its execution: a call that violates the declaration reverts (Section 2.4).

We study the guarantees APES provides against an attacker who observes all transactions in a block, deploys applications, and injects transactions, but cannot censor.[1] We show that transactions with no ordering constraints of their own (e.g., nonces) are always ordered according to the applications' unanimous preferences in three cases. The first covers *single-opinion* transactions, that is, those that interact with a single application that expresses preferences, plus any number of applications that do not. The protocol guarantees that when an application ranks a single-opinion transaction above another transaction (which may or may not be single-opinion), this preference is always respected in the final execution order. This covers oracle updates in a PropAMM, as long as they are embedded in single-opinion transactions. This guarantee also implies that all single-opinion transactions interacting with the same application execute in the order specified by that application. This covers on-chain auctions, for example liquidation auctions in lending protocols, as long as bidders send their bids as single-opinion transactions. It also implies that an application can obtain "rollup-like" ordering guarantees without needing to bridge assets to the app/rollup, since transactions interacting with the application can also interact with the asset contracts deployed on the base layer.

The second guarantee covers transactions that are multi-opinion (and hence not covered by the first guarantee) but *top cycle-safe*: those that an attacker cannot place in a cycle. This category includes *top-gated* transactions, which no attacker's transaction can outrank, either directly or indirectly. It also includes transactions that an attacker can outrank, but only with a transaction that cannot be used to form a cycle—for example, an attacker's transaction that interacts with only one application. These transactions always execute as the applications unanimously prefer. This covers cancellations spanning several markets and multiple oracle updates in the same transaction, which execute before market orders and swaps, respectively. The third guarantee is the mirror image of the second: multi-opinion transactions that are *bottom cycle-safe* keep every unanimous comparison placing other transactions above them. This covers closing transactions such as batch settlements and end-of-block checkpoints.

The remaining transactions are multi-opinion transactions that are neither top nor bottom cycle-safe. For those transactions, an attacker can insert transactions that create cycles or break existing unanimity links and, by doing so, overturn the applications' preferences. Senders of such transactions therefore face a trade-off: keep a transaction multi-opinion, preserving atomicity but risking an attack, or split it into several single-opinion transactions, gaining safety but losing atomicity.

We also consider attacks against the default order (and not the application preferences). When the applications are indifferent between two transactions, APES orders them by their priority fees. Because unanimity comparisons take precedence over priority fees, an attacker can front-run a transaction by maliciously creating unanimity links. This type of attack emerges when an application is indifferent where it should not be: the relative order of two transactions affects their outcome but the application's registered preferences treat them as equal. The guarantees provided against these types of attacks mirror the ones above. Transactions that the attacker cannot outrank in the unanimity relation, either directly or through the transactions ranked above them, are safe from front-running. So are transactions that interact with a single application whose execution order matters, provided that this application either expresses no preferences or ranks the relevant calls by a value, which can be the priority fee (Section 3).

Structure of this document. The remainder of this document is organized as follows. The next section discusses each component of APES in detail. Section 3 studies how a malicious actor could try to attack the system, derives the guarantees that the system provides, and discusses recommended best practices for application designers and users.

In Section 4, we illustrate how to apply those best practices to several examples of applications with ordering preferences: (i) a PropAMM, whose oracle updates execute before swaps; (ii) a CLOB in which cancellations of limit orders execute before market orders; (iii) a lending protocol's liquidation auction, in which the oracle update executes first, followed by the liquidations ordered by their bids, from high to low;

[1] Note that, under MCP, each proposer sees only the transactions in its own sub-block. Furthermore, each user chooses which proposers to submit to, based on their past behavior and reputation. Finally, a parallel workstream aims to introduce encrypted transactions on Monad. Our threat model is therefore particularly conservative.

(iv) a variant of the CLOB in which, in addition, the market deployer holds a special action (e.g., "halt trading") that executes even before cancellations, and partners can send "special swaps" that execute before regular market orders; and (v) a cross-market batch exchange whose matching engine settles all orders at the bottom of the block.

Section 5 discusses open questions on which we invite the reader to comment. Section 6 concludes: APES does not make the trade-off between composability and ordering preferences disappear, but pushes it to individual applications and users, who can best resolve it for their use case; in particular, "simple" transactions, which interact with a single application where ordering matters plus any number of applications where ordering does not matter (e.g., ERC20 contracts), are safe under very mild assumptions.

Appendix A collects the proofs of Section 3, Appendix B gives the Solidity interface of the preference registry, and Appendix C lists some implementation details that remain to be specified.

2 APES in four parts

Preliminaries. We use the following notation: t and u denote transactions, A an application, m a method named by its four-byte selector, and v a declared parameter. Throughout, an "application" is a smart contract. Section 5 discusses how APES could be extended to consider domains within a contract, such as the individual markets of a contract that holds several markets, as separate applications.

2.1 The Declaration: `interaction_list`

A transaction t declares which applications with ordering preferences it intends to access via its `interaction_list`: a field of a new EIP-2718 transaction type, covered by the signature. An `interaction_list` holds zero or more entries. Each entry specifies:

- the address of an opinionated application, that is, one with preferences in force (Section 2.2), distinct across the entries of t;
- the methods field: all methods that t may call on that application, among those for which the application registered its preferences. Each method is named by its four-byte selector and appears at most once per entry/application.[2]
- a parameter field per named method: an unsigned integer of at most `max_parameter`, or empty. The purpose of `max_parameter` is to allow the preference registry to verify, once and at registration time, that every possible score fits fixed-width arithmetic (see Section 2.2).

For example, consider a transaction that cancels a limit order on market A, while also placing a market order on the same market, declaring a value of 12 for that method. Such a transaction carries the entry (A, [`cancel`: –, `order`: 12]).

The semantics of a parameter are application- and method-specific. For example, for an auction that ranks bids from highest to lowest, the parameter declared for the `bid` method is the amount of the bid. A parameter can also stand for a quantity derived from several arguments of the call: an AMM may rank swaps by total fee paid, in which case transactions must declare as parameter "swap fee × amount", where the swap fee and the amount are two arguments of its `swap` call. It can also stand for a quantity unrelated to the arguments of the call, for example the transaction's priority fee or the balance of a given token held by the sender's address.

Not every application expresses preferences, and an opinionated application may register preferences for some methods but not others (for example, an AMM may leave deposits or withdrawals unranked). A transaction that calls only methods for which no preference is registered can carry an empty interaction list. Conversely, a transaction must declare the application and the method whenever it may access a method for which that application registered preferences (an empty parameter field is treated as zero).

[2] Among the open implementation details (Appendix C), we discuss whether transactions could also declare the pseudo-method `receive`, via a specific encoding. The `fallback()` pseudo-method, instead, cannot be declared: as Section 2.4 makes clear, a transaction should declare whatever method `fallback()` forwards to.

Transaction classes. An exception to the above rule is the *wildcard* transaction, whose list holds the wildcard entry: a reserved entry declaring that the transaction may call anything. Any other list, the empty list included, makes the transaction a *declaring* transaction.

As we will discuss later, wildcard transactions can access every method on any application, but are ordered at the end of the block (Section 2.3). They can be used in situations in which it would be impractical to specify the entire interaction list, for example, a swap transaction going through a router. The alternative, for a transaction whose execution path is hard to predict, is to over-declare: to list every opinionated application and method it may call. An over-declaring transaction is ordered like any other declaring transaction, but it is multi-opinion whenever it lists several opinionated applications, which exposes it to the attacks of Section 3.

A *legacy* transaction (one of the existing transaction types, which carry no such field) is treated as having the wildcard entry as its `interaction_list`, and is therefore a wildcard transaction: existing transaction types keep their access to every application.

Validity. A declaration is valid when it holds no entry, when it holds the wildcard entry and nothing else, or when all of the following hold:

- The list decodes to entries, each an application with its methods. For methods carrying a non-empty parameter field, the parameter must decode to a number no greater than `max_parameter`.
- The list holds at most `max_declared_applications` entries, and each entry at least one and at most `max_methods_per_application` methods.
- No application appears in two entries, and no method appears twice within an entry (this rule only forbids duplicates, so that each named method carries exactly one parameter field).
- Every entry declares an application with preferences in force, and only methods held by a clause of those preferences (see Section 2.2).

Intrinsic gas. The declaration adds to the intrinsic gas of t: `gas_per_declared_application` per entry and `gas_per_declared_method` per method. The wildcard entry carries no charge, so that a legacy transaction and a transaction declaring the wildcard entry pay the same intrinsic gas.

2.2 The Preference Registry

Applications register their preferences as a list of *clauses* (see below), in a precompile at the reserved address `PREFERENCE_REGISTRY`:[3]

- `register(Clause[] clauses) returns (uint256 activationBlock)` sets the preferences of the caller to `clauses` (at least one clause), provided the caller has opted in (see "Explicit opt-in" below), and returns the block from which they are in force, which is always later than the block in which the call runs (see "Activation delay" below);
- `revoke() returns (uint256 activationBlock)` sets the preferences of the caller to none, and returns the block from which the revocation is in force;
- `getPreferencesOf(address app) returns (Preference[])` returns the preferences of `app` in force, if any, followed by the pending ones, ordered by activation block; a `Preference` holds a list of clauses, empty after a revocation, and the block from which it is in force.

The precompile can be invoked only through a plain `CALL`, plus `STATICCALL` for `getPreferencesOf`; in particular, it rejects `DELEGATECALL`s. An application registers only for itself; an application with preferences in force is *opinionated*. An application can register its preferences at deployment, from its constructor, and it can call `register` again whenever it wants to change its preferences, or `revoke` to remove them.[4]

[3] Appendix B gives its full Solidity interface.

[4] Note that an already-deployed upgradeable proxy can adopt preferences without being redeployed: since the implementation's code runs through a `delegatecall` from the proxy, the precompile sees the proxy as the caller, and stores the record for the proxy. An already-deployed non-upgradeable contract, instead, cannot register.

Explicit opt-in. To protect applications deployed before APES, an application can register only after explicitly opting in by implementing a method

```
bytes4 constant MAGIC_VALUE = bytes4(keccak256("isApesEnabled()"));  // 0x45719097
function isApesEnabled() external view returns (bytes4) { return MAGIC_VALUE; }
```

The registry calls `isApesEnabled` on the caller of `register` and `revoke`, and reverts unless the call returns `MAGIC_VALUE`. Without this check, a deployed contract that lets users make arbitrary calls, such as a router, could be abused: an attacker could make it call the registry and register preferences on its behalf, making it opinionated against its developers' intent (see the following attack on Avalanche [3]). As a consequence, a deployed contract can become opinionated only if it can be upgraded to implement `isApesEnabled`.

The preference clauses. An application's preferences are represented by its `clauses`, where each clause is a tuple $(m,\ S,\ \{a_1, a_2, a_3, a_4\}, \{b_1, b_2, b_3, b_4\})$ with:

- m is a method selector,
- S is a set of addresses (the same set can appear in multiple clauses),
- $a_1, b_1, a_2, b_2, a_3, b_3, a_4, b_4$ are signed integers.

A method appears in at most one clause. One clause may name no method, in which case it *holds* every method the other clauses do not. At the ABI level, a clause carries a `MethodMatch` flag (`Exact`, or `AllOtherSelectors` for the catch-all clause), the selector m, the sender set S, and four `LinearScore` pairs (a_i, b_i) (see Appendix B).

Constraints. `clauses` holds at least one and at most `max_clauses` clauses, at most one clause names no method, and each S holds at most `max_senders_per_clause` addresses. `register` reverts unless, for every $i \in \{1, 2, 3, 4\}$, both b_i and $a_i \cdot$ `max_parameter` $+ b_i$ fit an `int128`.

Gas. Like any precompile call, `register`, `revoke`, and `getPreferencesOf` charge gas. The cost of `register` should grow with the number of clauses and of addresses in the registration, since translation has to load and evaluate all of them.

Scoring. A transaction t declaring a method m is assigned a score for that method. The score depends on the clause for that method, on whether the sender's address is in S, and on whether the transaction is single-opinion (it declares a single application in its list) or multi-opinion (it declares several).[5]

Recall that v denotes the parameter declared for m in t's list (an empty parameter field counts as $v = 0$). The score of m in t is:

- $a_1 \cdot v + b_1$ if t is single-opinion and its sender is in S;
- $a_2 \cdot v + b_2$ if t is single-opinion and its sender is not in S;
- $a_3 \cdot v + b_3$ if t is multi-opinion and its sender is in S;
- $a_4 \cdot v + b_4$ if t is multi-opinion and its sender is not in S.

Finally, for a transaction t with an entry for A, the registry computes the overall score assigned by application A to transaction t as

$$U_A(t) = \min\{\text{ the score of } m \text{ in } t \ :\ m \text{ a named method } t \text{ declares for } A \text{ that a clause holds }\}.$$

Preference. A prefers t before u if and only if $U_A(t) > U_A(u)$, and is indifferent if and only if $U_A(t) = U_A(u)$: a higher score means earlier desired execution. A transaction may also produce no score: when no named method it declares for A is held by a clause, $U_A(t)$ is undefined and, for ordering purposes, the transaction is treated as if it did not interact with that application.

[5] Note that, in a relayed transaction, for example under ERC-4337, the sender is the relayer, not the account on whose behalf the relayer acts.

Activation delay. An update to the registry takes effect with a k-block *activation delay* after a call to `register` or `revoke`: a call in block b sets the activation block of the new preferences to $b + k$ and returns it. Until then, the new preferences are pending. If an application calls `register` or `revoke` several times in the same block, only the last call that does not revert counts; calls in different blocks coexist as pending changes. The delay gives users and front ends time to see new preferences before they take effect.

The delay is also what makes APES compatible with Monad's asynchronous execution. To order block r, translation needs the preferences in force at r, all of which were set by calls in block $r - k$ or earlier. Since execution lags behind consensus and translation, the registry's state after block $r - k$ is available to the translation of block r only if k is at least Monad's asynchronous execution delay (three blocks today).

2.3 The Execution Order

The execution order of a block is computed in *translation*, a new step that takes as input a consensus block (i.e., the output of the consensus protocol) and outputs a block that can be executed. As already mentioned, MCP itself and other upcoming features, such as encrypted transactions, also rely on translation. The execution order is computed from the transactions' declarations, from the preferences in force and the default order.

Default order. The *default order* is built one transaction at a time. A transaction can be placed only after its nonce predecessor (see "Nonce edges" below). Among the transactions that can be placed, the next one is chosen as follows: (i) a non-wildcard transaction if there is one, and a wildcard transaction otherwise; (ii) among these, the one with the highest effective priority fee per gas,

$$\min(\text{priority_fee_per_gas}_t,\ \text{max_fee_per_gas}_t - \text{base_fee});$$

(iii) among these, the one with the lowest transaction hash. Absent wildcard transactions, this is the execution order the block would have under standard priority-fee ordering. A non-wildcard transaction whose nonce predecessor is a wildcard transaction can be placed only once that predecessor is placed, so it is ordered among the wildcard transactions.

Preference matrix. The scores of Section 2.2 are collected in a matrix M with one row per declaring transaction and one column per opinionated application: $M[t, A] = U_A(t)$ when t declares A and the score is defined, and $M[t, A] = \perp$ (no entry) otherwise.

Unanimity relation. The unanimity relation can be stated as a dominance relationship between rows of the preference matrix. For a pair of declaring transactions t and u, take the applications that score both: the columns A with $M[t, A] \neq \perp$ and $M[u, A] \neq \perp$. The relation places t before u when $M[t, A] \geq M[u, A]$ on every such column, with at least one strict inequality. A pair with no common column is left unranked, and so is a pair on which two columns disagree (one strictly greater, another strictly smaller). See Figure 1.

Nonce edges. Every transaction has at most one nonce predecessor: the transaction consuming the previous nonce of the same sender, or the carrier of the authorization that supplies it under EIP-7702. The *nonce edges*, written N, are these links, and they form a forest. We write N^+ for the transitive closure of N: $(t, u) \in N^+$ when t is a transitive nonce predecessor of u.

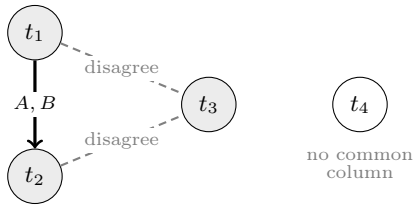
Pruning the unanimity relation. The *pruned unanimity relation* is obtained by removing two kinds of unanimity links:

- links that would place a transaction before one of its transitive nonce predecessors, which would trivially violate the nonce constraints. For example, with nonce edges $a \to b \to c \to d$, a unanimity link placing d before a would be deleted.
- all unanimity links of the transitive nonce successors of wildcard transactions.[6] Hence, non-wildcard nonce successors of a wildcard transaction are ordered like wildcard transactions: the applications' preferences no longer apply to them, and they execute among the wildcard transactions.

[6] Wildcard transactions cannot have unanimity links by construction.

	A	B	C
t_1	2	1	$\perp$
t_2	1	1	$\perp$
t_3	3	0	$\perp$
t_4	$\perp$	$\perp$	5

(a) preference matrix



(b) unanimity relation

Figure 1: The unanimity relation. (a) The preference matrix of four transactions and three applications; $\perp$ marks an application the transaction does not declare. (b) A and B both weakly prefer t_1 to t_2, and A strictly, so the relation places t_1 before t_2. The transaction t_3 disagrees with both: A ranks it above them, B below. The transaction t_4 shares no column with any other transaction and is unranked. Shaded transactions are multi-opinion.

Cycle detection and repair. The graph built from the pruned unanimity relation and the nonce edges can have cycles. These cycles need to be detected and repaired.

Write P for the pruned unanimity comparisons, F for the comparisons imposed by repair (initially empty), N for the nonce edges, and $G = P \cup F \cup N$ for the working graph, with a pair (u, t) in any of these relations placing u before t. A *cyclic component* is a strongly connected component of G with at least two transactions; define $\mathrm{Cyc}(G)$ as the set of transactions lying in a cyclic component of G, $\mathrm{SCC}_G(t)$ as the cyclic component holding t, and $\mathrm{Desc}_N(t)$ as the transitive nonce successors of t. Call a transaction *single-opinion* when exactly one column of its row of M holds a score and *multi-opinion* when two or more do.

The first step of the algorithm detects cycles. Two commonly used algorithms are Kosaraju's algorithm and Tarjan's strongly-connected-components algorithm, which return the cyclic components directly. Both run in linear time in the number of transactions plus comparisons.

If a cycle is detected, the next step breaks it via a process called *demotion*. Demoting a transaction d from its cyclic component $Y = \mathrm{SCC}_G(d)$ means:

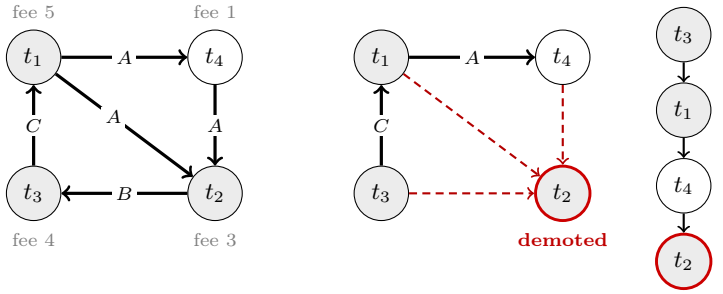
- Delete unanimity links: Remove the unanimity links between d and its nonce successors in Y, on one side, and all other transactions of Y, on the other. That is, remove all unanimity links between the set $D = (\{d\} \cup \mathrm{Desc}_N(d)) \cap Y$ and the set Y, in both directions, together with any comparison imposed by an earlier demotion between the two sets.
- Impose additional comparisons: Place all transactions in $Y \setminus D$ above those in D. That is, add to F a comparison placing every transaction of $Y \setminus D$ before every transaction of D.

Each round of repair demotes a single transaction, chosen as follows. If some cyclic component contains multi-opinion transactions, demote the one that is lowest in the default order. If no cyclic component contains a multi-opinion transaction, demote one of the transactions that hold a nonce predecessor in their own cyclic component; again, if there are several candidates, choose the one that is lowest in the default order.

After each demotion, cycle detection is run again on the previously detected cyclic component. If another cycle is detected, another transaction is demoted. Every demotion removes at least one unanimity comparison, there are finitely many, and no cycle is possible once all unanimity comparisons are gone (the nonce edges alone form a forest and the imposed comparisons close no cycle), so the process terminates. Figure 2 illustrates the procedure.

Completion. The cycle repair step produces an acyclic graph of surviving unanimity comparisons, imposed repair comparisons, and nonce edges: a directed acyclic graph representing a partial order. This partial order is then completed into an *execution order* E using the default order via Kahn's topological sorting algorithm.

Kahn's topological sorting algorithm works in the following way. E starts empty and grows by one transaction at a time; a transaction is *available* when every transaction preceding it in G is already in E. At each step of the algorithm, E grows by the available transaction with the highest default order. Note that, by construction, wildcard transactions and their nonce successors are sequenced after all non-wildcard



(a) cyclic component (b) after demoting t_2 (c) execution order

Figure 2: Cycle repair. (a) Four transactions form a cyclic component; each edge is labeled by the application ranking the pair, and the fees give the default order. The transactions t_1, t_2, and t_3 are multi-opinion (shaded), while t_4 is single-opinion. (b) Repair demotes t_2, the multi-opinion transaction lowest in the default order (t_4 pays less, but is not multi-opinion): it removes the unanimity links between t_2 and the rest of the component, and imposes comparisons (dashed) placing t_1, t_3, and t_4 above it. (c) Kahn's algorithm completes the result: t_3 has no predecessor and comes first, followed by t_1, t_4, and t_2.

transactions, and, among wildcard transactions, the only links Kahn's algorithm needs to respect are nonce links. The resulting execution order E is unique: every choice Kahn's algorithm leaves is resolved by the default order, with ties broken by transaction hash. Algorithm 1 gives the full pseudocode.

2.4 Execution

The transactions in the block execute in the order produced by Algorithm 1. Ordering is computed before execution, from the declarations alone, but must be consistent with the transactions' behavior at execution.

Interactions. We say that t *interacts* with an application A when a frame of its execution, at the top level or at any depth, can modify A's storage, that is, has A as its executing account (the address returned by `ADDRESS`): a `call` to A, including the transaction's top-level call, or a `delegatecall` or `callcode` executed by A. Where the code run by the frame lives does not matter: the application is the one whose storage the frame acts on. The method of the interaction is the first four bytes of the frame's calldata. This matches the ABI convention that Solidity uses to dispatch calls. A frame with fewer than four bytes of calldata carries no method: the protocol check below skips it, even when A's preferences hold a catch-all clause. A `staticcall` opens no interaction, since it cannot write. Note that the protocol does not see which function of A handles a frame—a named function, or `fallback()`—only its calldata. For example, consider a proxy whose `fallback()` forwards every call to its implementation through a `delegatecall`. A call to the proxy carries the selector of the implementation's method, and so does the `delegatecall` the proxy makes. Both frames act on the proxy's storage, and the protocol sees the forwarded method in both.

Protocol check: methods. Let t be a declaring transaction and let A be opinionated. If t interacts with A via a method that A ranks (i.e., one held by a clause of its preferences in force) without having declared it for A, the frame reverts without running any code.

`DECLAREDPARAM`. A new opcode `DECLAREDPARAM` reports, for the frame it runs in, one word holding a tag byte and a payload: either `0x00` and the parameter that transaction t declares for the method of the frame (zero when the parameter field was left empty), or `0x01` and zero bytes where t declared no such method or the frame carries no method, or `0x02` and zero bytes where t declared the wildcard entry.

Application check: parameters. The application enforces that the parameters specified in the transaction declaration conform with its execution: it reads the declared parameter of the call through `DECLAREDPARAM` and decides how to treat a transaction whose execution does not honor it.

Algorithm 1: Repairing and ordering a block

$P \leftarrow$ the unanimity relation over the declaring transactions
$N \leftarrow$ the nonce edges
$W \leftarrow$ the wildcard transactions and their transitive nonce successors
$F \leftarrow \emptyset$ ▷ imposed repair comparisons
delete from P every comparison involving a transaction of W ▷ Step 0: wildcard pruning
delete from P every (u, t) such that $(t, u) \in N^+$ ▷ Step 0: nonce pruning
$G \leftarrow P \cup F \cup N$
while $\mathrm{Cyc}(G) \neq \emptyset$ **do**
    $C \leftarrow \{t \in \mathrm{Cyc}(G) : t \text{ is multi-opinion}\}$ ▷ Step 1
    if $C = \emptyset$ **then**
        $C \leftarrow \{t \in \mathrm{Cyc}(G) : (p, t) \in N \text{ for some } p \in \mathrm{SCC}_G(t)\}$ ▷ Step 2
    end if
    $d \leftarrow$ the transaction of C with the lowest default order
    $Y \leftarrow \mathrm{SCC}_G(d)$
    $D \leftarrow (\{d\} \cup \mathrm{Desc}_N(d)) \cap Y$ ▷ d and its nonce successors in Y are demoted together
    $P \leftarrow P \setminus ((D \times Y) \cup (Y \times D))$
    $F \leftarrow F \setminus ((D \times Y) \cup (Y \times D))$
    $F \leftarrow F \cup ((Y \setminus D) \times D)$
    $G \leftarrow P \cup F \cup N$
end while
$E \leftarrow \langle\rangle$
$R \leftarrow$ all transactions of the block
while $R \neq \emptyset$ **do**
    $A \leftarrow \{t \in R : \nexists u \in R \text{ with } (u, t) \in G\}$ ▷ available
    $B \leftarrow A \setminus W$
    $t \leftarrow$ the transaction with the highest default order in B if $B \neq \emptyset$, and in A otherwise
    append t to E
    $R \leftarrow R \setminus \{t\}$
end while
return E

Phantom declarations. A transaction could declare a method without ever calling it at execution. In this case, the protocol checks never fire, but the (incorrect) declaration is still used to order transactions. The possibility of inserting transactions with phantom declarations to manipulate the execution order is discussed in Section 3.

2.5 Parameters

- `PREFERENCE_REGISTRY`: reserved address of the preference registry precompile.
- `max_clauses`: maximum number of clauses in a registration.
- `max_senders_per_clause`: maximum number of addresses in the S of a clause.
- `max_declared_applications`: maximum number of entries in a declaration.
- `max_methods_per_application`: maximum number of methods per entry of a declaration.
- `max_parameter`: the largest parameter a transaction may declare for a method, provisionally $2^{96} - 1$.
- `gas_per_declared_application`: intrinsic gas per entry of a declaration.
- `gas_per_declared_method`: intrinsic gas per declared method.
- `k`: the delay after which a registration, replacement, or revocation takes effect.

3 Attack vectors and guarantees

This section studies how an attacker could try to subvert the execution order produced by APES, and which guarantees survive. Note that, today, Monad's block proposers choose the order of transactions freely, so every manipulation described below is already possible for a proposer, at no cost. APES removes most of this freedom, since the protocol enforces the order. The attacks below show what room for manipulation remains.

We assume that the attacker observes the block, injects transactions carrying any valid declaration from any address outside the sets S of the clauses, deploys applications (with preferences), and influences the default order, for example by outbidding on priority fees. The attacker cannot censor transactions, thanks to the multiple concurrent proposers of the MCP architecture we assume (Section 1). We only consider attacks against non-wildcard transactions with no nonce edges.

We first define the classes of transactions covered by the guarantees (Section 3.1), then present four attacks (Section 3.2), and finally state the guarantees (Section 3.3).

3.1 Definitions

Throughout, X denotes the transactions of the block and R the transactions that the attacker could inject.

Definition 3.1 (Applications and transactions)**.** An application is *order-sensitive* if the outcome of a call to it can depend on which other transactions execute before it on the same application; AMMs, order books, auctions, and liquidations in lending protocols are order-sensitive, while a token contract such as an ERC20 is not. An application is *opinionated* if it has registered preferences and those preferences are in force (Section 2.2). We assume that every opinionated application is order-sensitive.

A transaction is *single-app* if it interacts with a single order-sensitive application (plus any number of non-order-sensitive applications), and *single-opinion* if it interacts with a single opinionated application (plus any number of non-opinionated applications).

For two declaring transactions u and t in $X \cup R$, we write $u \succ t$ if u precedes t in the graph composed of the pruned unanimity relation and the nonce edges (see Section 2.3). If $u \succ t$, we say that u *ranks above* t, and t *ranks below* u. The *ancestors* of t are the transactions of X linked to t by a chain of transactions of X, each ranking above the next; its *descendants* are defined symmetrically. The ancestors and descendants of t are therefore all the transactions of X linked to t, directly or indirectly.

Definition 3.2 (Cycle-safe transactions)**.** Let X^* be the largest subset of $X \cup R$ such that (i) no cycle of $\succ$ lies within X^*, and (ii) every transaction that ranks above a transaction in X^* also belongs to X^*. We say that a transaction that belongs to X^* is *top cycle-safe*. Symmetrically, let X_* be the largest subset of $X \cup R$ that satisfies (i) and contains every transaction ranking below one of its transactions; a transaction that belongs to X_* is *bottom cycle-safe*.

The largest such sets exist, because the union of two sets satisfying (i) and (ii) satisfies them as well. The above definition implies that a transaction that is top or bottom cycle-safe cannot lie on a cycle, no matter what the attacker injects.

Definition 3.3 (Gated transactions)**.** A transaction t is *top-gated* if it is top cycle-safe, and no transaction of R ranks above t or any of its ancestors. Symmetrically, t is *bottom-gated* if it is bottom cycle-safe, and no transaction of R ranks below t or any of its descendants.

Note that cycle-safety is about an attacker not being able to close a cycle through a transaction, whereas gating is about an attacker not being able to craft a transaction that directly or indirectly ranks above (top-gated) or below (bottom-gated) an honest transaction. Hence, gated transactions are also cycle-safe, but the converse is not true.

3.2 Attacks

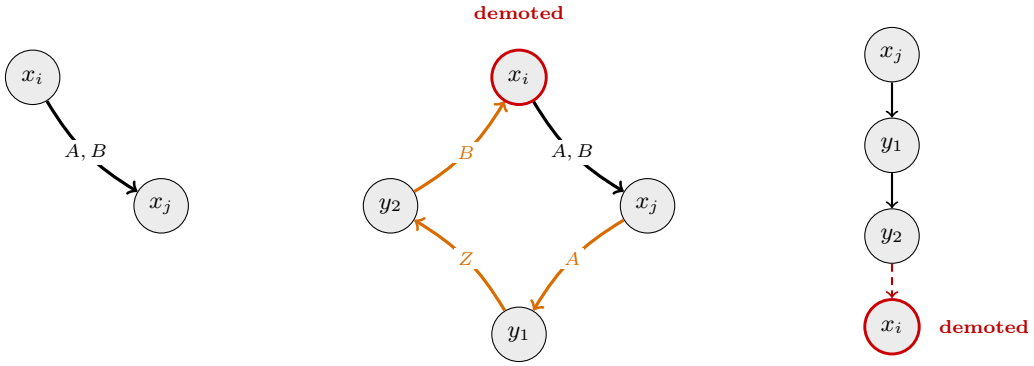
The first two attacks target the applications' preferences: (i) an attacker can force the system to ignore a unanimity ranking by creating a cycle and triggering demotion, and (ii) an attacker can break a unanimity

ranking by creating a disagreement between two applications. The last two attacks instead target the execution order of transactions among which the applications are indifferent: priority fees should determine this order, but an attacker can overturn it.

Attack 1 (Cycle creation and demotion)**.** Suppose the applications unanimously place x_i before x_j, so that $x_i \succ x_j$. An attack succeeds if it places x_i and x_j in the same cyclic component and causes the fallback to demote x_i.
As an illustrative example, consider two honest applications A and B that both rank x_i above x_j, so $x_i \succ x_j$. The attacker leaves the target transactions unchanged, deploys one application Z, and injects two transactions: y_1, declaring A and Z, whose declaration at A satisfies $U_A(x_j) > U_A(y_1)$; and y_2, declaring B and Z, whose declaration at B satisfies $U_B(y_2) > U_B(x_i)$. Since Z is the only application ranking the pair (y_1, y_2), the attacker registers clauses for Z that give $y_1 \succ y_2$. The resulting cycle is

$$x_i \succ x_j \succ y_1 \succ y_2 \succ x_i.$$

All four transactions are multi-opinion, so Step 1 of the repair demotes the one lowest in the default order. The attacker therefore sets the priority fees of its transactions so as to trigger the demotion of x_i. Figure 3 shows the cycle and the demotion of x_i. Note that neither y_1 nor y_2 needs to call anything: phantom declarations suffice, so the attack costs two transactions' intrinsic gas plus one deployment.



(a) baseline: $x_i \succ x_j$ (b) the attacker's cycle (c) execution order

Figure 3: The cycle-and-demotion attack. The attacker preserves the target transactions, creates a cycle around the baseline edge, and makes the fallback demote x_i. Kahn's algorithm then extends a repaired relation that places x_j above x_i.

Two conditions are necessary for the above attack to succeed: the target must be multi-opinion, and therefore eligible for demotion, and it must not be cycle-safe, so that a cycle through it can be closed—here, by y_2 above x_i and y_1 below x_j.

Attack 2 (Erasing comparability)**.** A unanimity comparison can also be prevented from arising. Consider two honest opinionated applications, A and B, and an honest transaction x interacting with both. The attacker holds a transaction y that declares A, with

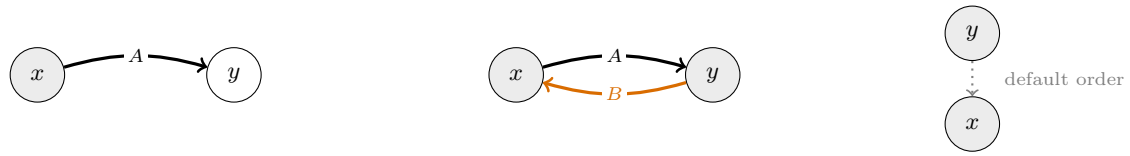
$$U_A(x) > U_A(y),$$

so that $x \succ y$ and y cannot execute before x. However, the attacker can include in y's list a phantom declaration that y will also access application B, choosing the method and parameter such that

$$U_B(y) > U_B(x).$$

The two applications now disagree over (x, y), so no unanimity edge exists in either direction, and the

relative order of the pair falls to the default order, which the attacker influences: y can now execute before x. Figure 4 illustrates the attack.



(a) without the phantom entry: $x \succ y$

(b) with the phantom entry at B: disagreement, no unanimity edge

(c) execution order: y before x

Figure 4: The comparability-erasing attack. At A, the attacker's transaction y ranks below x. A phantom declaration at B, with a score above x's, makes the two applications disagree: the pair holds no unanimity edge, and its relative order falls to the default order, which the attacker wins by outbidding.

Here too, the attack can succeed only if the target transaction is multi-opinion and the attacker can craft a multi-opinion transaction that scores strictly below the target at one application and strictly above it at another (here, a transaction that need not call anything and carries a phantom declaration for application B). When these conditions fail, this attack is infeasible; in particular, it is infeasible against transactions that are either top or bottom cycle-safe.

The previous attacks target the applications' preferences. Respecting those preferences is also in the users' interest, since users submit transactions to applications knowing their preferences. The converse, however, need not hold: applications may be indifferent where users are not. In particular, users may not want to be front-run. A natural requirement is therefore that front-running a transaction be at least as expensive under APES as under standard priority-fee ordering. The next attacks show that this is not necessarily the case.

Attack 3 (Cheap front-running)**.** Suppose a transaction x interacts with two applications, A and B, and does not hold the top score at B. For example, x may place market orders on two markets that rank cancellations above market orders and are indifferent among market orders. The attacker submits a transaction y declaring the same method as x at A (and hence obtaining the same score at A), while also declaring, *but not calling*, a method at B with a score higher than x's. To continue with our example, y may declare (and execute) a market order at A and declare (but not call) a cancellation at B. Because y scores higher than x at B and the same as x at A, y is unanimously ranked above x and, absent cycles, executes before x, even if it pays less than x in priority fee. That is, the attacker front-runs x while paying only the minimal inclusion fee. Figure 5 illustrates the attack.

Variant: single-opinion victims. It is easy to check that the attack also works when application A expresses no preferences (such as a traditional AMM), so that both the victim transaction x and the attacker transaction y are single-opinion.

In both forms, the attacker succeeds if (i) the victim is not single-app: besides an opinionated application, it interacts with another order-sensitive application, opinionated or not; and (ii) the attacker can rank above the victim, so that the victim is not top-gated. Again, the application preferences are respected, but the attacker bypasses the rule that, when applications are indifferent, the priority fee determines the relative order of transactions.

A second version of the attack targets single-app transactions.

Attack 4 (Cheap front-running through a cycle)**.** Let x be a transaction that only places a market order on market A, which has the same preferences as in Attack 3. The attacker deploys two applications, Z_1 and Z_2, and sends three transactions:

- f, which places a market order on A and also declares Z_1;

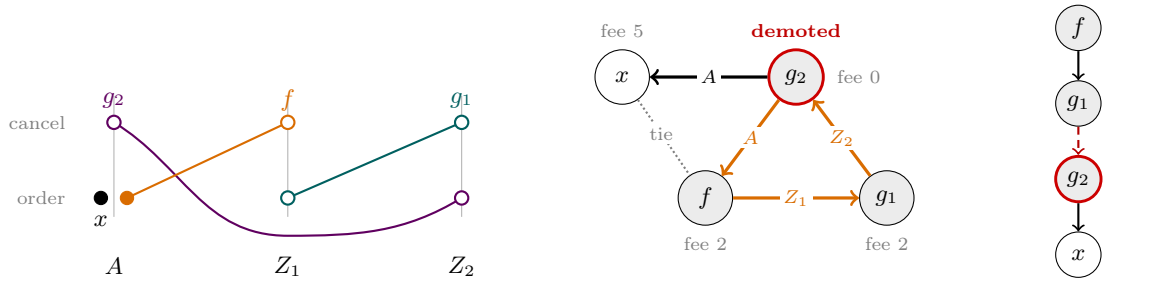


(a) without the phantom entry (b) with a phantom cancellation at B

Figure 5: The cheap front-running attack, drawn by application. Each column is a market, with cancellations ranked above market orders, and each transaction is a line through the methods it declares (filled dots: methods it calls; hollow dots: phantom declarations). (a) Without the phantom entry, y only places a market order at A, tying x: the pair falls to the default order, and x, which pays more, executes first. (b) A phantom cancellation at B lifts y's line above x's at B, while the two still tie at A, so $y \succ x$: y executes first, at the minimal fee.

- g_1, which declares Z_1 and Z_2;
- g_2, which declares Z_2 and phantom-declares a cancellation at A.

The attacker registers clauses for Z_1 and Z_2 that place f above g_1 and g_1 above g_2, while A places g_2's cancellation above the market orders of both f and x. Hence f, g_1, and g_2 form a cycle, while f and x, which both place a market order at A, tie. All three attacker transactions are multi-opinion, so if g_2 pays the lowest fee of the three, repair demotes it and places f and g_1 above it, but leaves the comparison placing g_2 above x untouched, since x lies outside the cyclic component. Therefore f executes before g_2, and g_2 before x: f's market order executes on A before x's, at a fee barely above the minimum. Figure 6 illustrates the attack.



(a) the transactions, by application (b) the unanimity relation (c) execution order

Figure 6: Cheap front-running through a cycle. (a) The transactions, drawn as in Figure 5. The victim's transaction is x (fee 5). The attacker deploys applications Z_1 and Z_2 and injects transactions f (fee 2), g_1 (fee 2), and g_2 (fee 0). (b) In the resulting unanimity relation, the attacker's three transactions form a cycle (orange), and repair demotes g_2, which pays the lowest fee among them. (c) The execution order still places g_2 above x, since that comparison lies outside the cycle and survives. Thus, f executes before x.

Attack 4 relies on f tying x at A, which is possible because A ranks all market orders equally. Suppose instead that A ranks market orders by a value, for example a bid charged by the application or the transaction's priority fee.[7] If f bids less than x, then A, the only application scoring both, ranks x above f. Theorem 3.1 (see below) guarantees that x executes before f: x now ranks above f and joins the cyclic component, and repair still demotes g_2, which loses all its comparisons within the component, including the one placing it above x. To front-run x, the attacker must therefore bid at least as much as x, which is as

[7]The value can also be the priority fee itself: the application requires each market order to declare its priority fee per gas as the parameter and, at execution, checks the declared parameter, read through `DECLAREDPARAM`, against the transaction's effective priority fee per gas.

costly as outbidding x under priority-fee ordering.

Hence, single-app transactions are protected from cheap front-running if their application also ranks by value—which in a sense means that "no cheap front-running" is part of the application's preferences. Note, however, that this comes at a cost. Under the preferences of Attack 3, a transaction placing market orders on several markets is bottom cycle-safe: nothing can score below it, so it lies on no cycle, and no attacker transaction can score below it at one market and above it at another. It is therefore protected from the first two attacks (though exposed to cheap front-running). If the markets also rank market orders by priority fee, such a transaction is no longer bottom cycle-safe—an attacker can phantom-declare a market order paying no fee, which ranks below every other—and could, in principle, fall victim to Attack 1 or Attack 2. Protecting single-app transactions from cheap front-running therefore opens another attack vector against multi-opinion transactions touching the same application.

3.3 Guarantees

Against these attacks, the protocol provides the following guarantees. Before listing them, we note that the guarantees stem partly from the protocol, partly from the application design, and partly from users' behavior. Again, we only consider non-wildcard transactions with no nonce edges.

The first guarantee is that no attack can overturn an application's preference for a single-opinion transaction over a transaction it ranks lower. This guarantee is stated formally in the following theorem; the proof is in [2].

Theorem 3.1 (Single-opinion invariance)**.** *Let t be a single-opinion transaction, let A be the application scoring it, and let u be any other transaction that A scores. If $U_A(t) > U_A(u)$, then t executes before u, whatever the attacker does.*

For intuition, note that if t declares a single opinionated application A, then A is the only application that scores both t and u, so the unanimity relation over the pair coincides with the preference of A. Furthermore, t is never demoted: it is not multi-opinion, and it holds no nonce predecessor. Finally, if u is demoted from a cyclic component holding both, the comparisons imposed by the repair still place t above u. Applying the theorem to transactions that are single-opinion at the same application implies that those transactions always execute in the order the application specifies.

The second guarantee covers cycle-safe transactions, even when multi-opinion.

Proposition 3.2 (Cycle-safe transactions)**.** *Let t be a top cycle-safe transaction. If t ranks above another transaction u, then t executes before u, whatever the attacker does.*

The proof is in Appendix A. A symmetric statement holds for bottom cycle-safe transactions: they lie on no cycle, and every transaction that ranks above them executes before them.[8]

The next guarantees concern protection against front-running attacks. The first concerns top-gated transactions, and shows that an attacker's transaction executes before a top-gated transaction x only if it outspends either x or one of its ancestors.

Proposition 3.3 (Top-gated transactions)**.** *Let x be a top-gated transaction. Then Proposition 3.2 applies to x, and every transaction that executes before x precedes, in the default order, either x or one of its ancestors, all of which are "honest" (i.e., not injectable by an attacker). In particular, if x has no ancestors, then front-running x is at least as expensive as under priority-fee ordering.*

The proof is in Appendix A. Note that a transaction that interacts with no opinionated application is trivially gated, with no ancestors: by Kahn's algorithm, every transaction executing before it precedes it in the default order.

Finally, single-opinion transactions are protected from cheap front-running when the application also ranks by value.

[8] This guarantee is weaker than its top counterpart, because wildcard transactions are still sequenced after bottom cycle-safe transactions. The application can, however, tell whether a transaction executes after such a transaction, and can condition its behavior on this information (the batch exchange of Section 4.5 illustrates this point).

Corollary 3.1 (Value ranking)**.** *Let x be a single-opinion transaction that declares a single method m at its application A, and suppose that the clause of m has $a_1 = a_2 = a_3 = a_4 = a > 0$ and $b_1 = b_2 = b_3 = b_4 = b$, so that A ranks calls to m by their declared value. Then every transaction that calls m on A and executes before x declares a value at least as large as that of x.*

The proof is in Appendix A. When A enforces the declared value at execution, front-running x on m therefore requires outbidding it on that value; when the value is the priority fee, this is exactly as costly as under priority-fee ordering.

Note that the above guarantee concerns front-running on the opinionated application. As Attack 3 shows, a transaction that is single-opinion but multi-app may be front-run on the application with no ordering preferences. However, Proposition 3.3 and Corollary 3.1 together protect single-app transactions under mild conditions: if the order-sensitive application they interact with expresses no preferences, they interact with no opinionated application and are covered by Proposition 3.3; if it is opinionated and ranks by value, they are covered by Corollary 3.1. We summarize these observations in the following corollary.

Corollary 3.2 (Single-app guarantee)**.** *Let x be a single-app transaction, and let A be the order-sensitive application it interacts with. Suppose that either A expresses no preferences, or x satisfies the conditions of Corollary 3.1 at A and A enforces the declared value at execution. Then x cannot be cheaply front-run.*

3.4 Best practices

Which transactions are cycle-safe or gated depends on how clauses are specified, and is therefore part of the application's design space. For example, if the top method's score is accessible to any sender and any transaction (its clause is unrestricted), an attacker can always send a multi-opinion transaction that copies the declaration of a top-ranked transaction and obtains the same scores. The copy does not rank above the original: equal scores create no unanimity edge, so the original remains gated. But the copy ranks above every transaction with a strictly lower score. As a consequence, only transactions holding the top score are cycle-safe. On the other hand, if an application's top score is reserved to the senders in S while senders outside S receive an arbitrarily low score, both its highest- and its *second*-highest-ranked transactions are gated and therefore cannot be attacked (see Section 4.4).

Similarly, a clause can reserve a score to single-opinion transactions, so that multi-opinion transactions receive an arbitrarily small score for the same method. Suppose that, at some application, every transaction that can obtain one of the k highest score levels is cycle-safe, and that the $(k+1)$-th level is reserved to single-opinion transactions. The unanimity ranking between a single-opinion transaction and any other transaction depends on the preferences of a single application, so a single-opinion transaction at the $(k+1)$-th level is cycle-safe too, even when sent by the attacker. The transactions at the $(k+2)$-th level then benefit as well: they are cycle-safe themselves, provided the same holds at every other application scoring them.

Finally, as already discussed, whether cheap front-running on an opinionated application is possible depends on whether that application ranks by value.

These considerations can be summarized in three best practices for application designers.

Use the reserved senders S in the preference clauses. If the highest-ranked methods are reserved to some senders, then transactions calling the method ranked immediately below are protected. Transactions calling that highest non-reserved method (including across different instances of the same application) are top-gated, and hence immune to all the attacks discussed so far.

Assign higher scores to single-opinion transactions calling a given method. When the highest-ranked methods are reserved to single-opinion transactions, transactions calling the method ranked immediately below are top cycle-safe, but not top-gated. That is, an attacker can outrank those transactions by declaring the reserved method, but because the reserved method is only for single-opinion transactions, the attacker cannot use such a declaration to place the victim's transaction on a cycle or to break a unanimity link. Attacks 1 and 2 are therefore prevented.

Rank by priority fee (or some other value). If, once the application's preferences are applied, accessing a given method before other transactions is still valuable, then the application developer should reflect this in the preference clauses, by ranking that method by priority fee (or some other value). This protects single-opinion transactions calling that method (Corollary 3.1).

The above guarantees also depend on users' behavior. In this respect, note that, as long as the application where ordering matters follows the last best practice (or expresses no preferences), users can guarantee the safety of their transactions by interacting with that single application, plus any number of applications, such as ERC20 contracts, where ordering does not matter. In all other contexts, the safety of a transaction depends on the details of the preferences expressed by the applications it interacts with and on the specific threat model.[9] The recommended best practice for users is therefore the following:

Keep it simple. As far as possible, each transaction should interact with a single application where ordering matters.

4 Examples

We now show how the best practices of Section 3.4 can be applied by discussing five illustrative examples of applications with ordering preferences, in increasing order of complexity. The first two rank mainly by method: a PropAMM that wants its oracle update before swaps, and a CLOB whose markets want cancellations before market orders. The main difference between the two is that oracle updates come from a known set of keepers, while cancellations come from any trader, and a single transaction may cancel limit orders across multiple markets.

The third is a liquidation auction where an oracle update executes first and all bids are then ordered from high to low. Here the point is to show how an application's preferences can depend on the value a transaction declares. The fourth is a CLOB with a special operation (for example, "halt trading") that executes before cancellations, and where swaps from partners and integrators execute before regular ones.[10] The main point there is that the application's ordering guarantees depend on appropriately specifying the sender sets S and on distinguishing single- from multi-opinion transactions.

The last example is a CLOB with batch trading, where a matching engine triggers trades by executing a special transaction after all market orders. In this example, market orders can also bid for their placement, with the highest-paying orders executing last. This auction captures the option value of the "last look", that is, the value of knowing at what price the trade will execute. This last application combines all the elements of the previous examples: there are preferences based on methods, but also on values (here for being ranked later), and the ranking is correct only if the `settle` score is reserved to the matching engine's addresses in S and only single-opinion orders can bid for their position.

Again, these are illustrative examples, and are not meant to be full specifications of how to design, for example, a multi-market batch trading system based on APES. When writing clauses, if the four coefficients $a_1, \dots, a_4$ of a clause are equal we write their common value only once, and similarly for $b_1, \dots, b_4$; for example, (`order`, $\emptyset$, $\{1\}$, $\{-1\}$) assigns score $v - 1$ to all transactions calling `order`, whatever their senders and whether they are single- or multi-opinion.

4.1 A PropAMM: oracle updates before swaps

This example covers applications that want a specific method, called by a known set of addresses, to execute before every other method.

The PropAMM's oracle updates are sent by a set S of keepers, and the PropAMM registers two clauses:

$$(\texttt{update}, S, \{0\}, \{1, 0, 1, 0\}), \qquad (\texttt{swap}, \emptyset, \{0\}, \{0\}).$$

[9] It is worth reiterating that "safety" here is relative to an attacker who can observe the content of the block. If instead transactions are private (for example, only proposers observe transactions, and each proposer only observes a subset of transactions) or even encrypted, then the set of transactions that are safe is larger.

[10] On Hyperliquid, the deployer of a market holds a `haltTrading` action, which cancels all resting orders and settles positions at the mark price (HIP-3).

The oracle update declares (`AMM`, `update`) and, sent by a keeper, scores 1; declared by any other sender, `update` scores 0, like a swap. Each swap declares (`AMM`, `swap`), with an empty parameter field, and scores 0. The AMM therefore prefers the keepers' updates before every swap and is indifferent among swaps. Note also that a transaction declaring (`AMM`, `update`) *and* (`AMM`, `swap`) is assigned score 0, like swaps.

No transaction can score above a keeper's update, so updates are gated, and always execute before all swaps, whether single- or multi-opinion. A swap that declares no other opinionated application can be outranked only by keepers' updates, and no transaction the attacker can send scores above it, so it is gated as well. Such a swap is also safe from cheap front-running, even though the AMM is indifferent among swaps: Attacks 3 and 4 need a transaction ranked above the victim by an application scoring it, and here no such transaction is accessible to the attacker. Multi-opinion swaps are covered by the same guarantees as long as they cannot be outranked by an attacker's transaction, for example, because they swap on several PropAMMs with the same design. The restriction to keepers must be in the clause, even though the PropAMM's code already reverts updates from other senders. Ordering is computed from declarations alone: with an unrestricted clause, any transaction could phantom-declare `update`, outrank every swap, and carry out Attack 3, and the contract's check would never fire, since a phantom declaration never calls `update`.

Under this record, only `update` and `swap` are ranked. A liquidity deposit or withdrawal is unconstrained: it needs no entry for the AMM (a transaction that only deposits can carry an empty interaction list), and it can execute anywhere in the block, even before the oracle update. A PropAMM that instead wants every other method below update and swap can register

$$(\texttt{update},\, S,\, \{0\},\, \{1,0,1,0\}), \qquad (\texttt{swap},\, \emptyset,\, \{0\},\, \{0\}), \qquad (\emptyset,\, \emptyset,\, \{0\},\, \{-1\}),$$

where the third clause holds every method the others do not. Now every method of the AMM is ranked, so every transaction touching it must declare the method it calls: a deposit declares (`AMM`, `deposit`) with an empty parameter field and scores -1, below every swap, while the keepers' updates remain above swaps, deposits, and withdrawals alike.

Finally, this example assumes that each PropAMM is a separate contract or, if several markets live in the same contract, that the same keepers update the oracle for all of them: under the current semantics, it is not possible to specify different keepers for different markets that share the same contract and the same methods. It is also worth noting that most PropAMMs are implemented as a proxy contract plus an implementation contract. Since it is the proxy's state that is modified, the preferences are registered by the proxy. As already mentioned, this can be done via a `delegatecall` to a gated `setPreference()` function on the implementation contract. Specifying preferences therefore requires neither updating nor redeploying the proxy contract.

4.2 A CLOB: cross-market cancellations before market orders

Treat each market M_k as a separate application with preferences

$$(\texttt{cancel},\, \emptyset,\, \{0\},\, \{B\}), \qquad (\texttt{order},\, \emptyset,\, \{1\},\, \{-1\}),$$

with $B =$ `max_parameter`. A market order declares $(M_k, \texttt{order}, v)$, with v its priority fee per gas, and scores $v - 1 < B$; the market checks the declared parameter against the transaction's effective priority fee at execution. A cancellation spanning several markets declares $(M_k, \texttt{cancel})$ for every market it touches, and is therefore multi-opinion. Every market scoring it assigns B, above every market order, so the unanimity relation places it before all market orders of all its markets. Although multi-opinion transactions are in principle eligible for demotion, this one is gated: no transaction can score strictly higher at any of its markets, so Proposition 3.3 applies. Ranking market orders by priority fee, in turn, protects single-opinion market orders from cheap front-running through a cycle (Attack 4). As discussed in Section 3, however, fee ranking exposes multi-opinion market orders to Attack 1.

Again, the example assumes that each market is a separate contract: a CLOB holding all its markets in a single contract cannot express different preferences for each market under the current semantics. A single contract does, however, shrink the attack surface. A transaction that places market orders on several markets of the same contract declares a single application, and is therefore single-opinion (as long as it touches no other opinionated application). Attacks 1 and 2 require a multi-opinion target, so they no longer

apply, and Corollary 3.1 protects such a transaction from cheap front-running on the CLOB. The cost is coarser preferences: a cancellation on one market executes before the market orders of every market, and a transaction that cancels on one market and places a market order on another scores as a market order, since an application's score is the minimum over the declared methods.

4.3 An auction for liquidations

A lending protocol wants fresh oracle prices before any liquidation, and the liquidations ranked by the amount each bids to the protocol for priority. With S the addresses of its oracle and $B =$ `max_parameter`, it registers

$$(\texttt{update},\ S,\ \{0\},\ \{B,-1,B,-1\}), \qquad (\texttt{liquidate},\ \emptyset,\ \{1\},\ \{-1\}).$$

A liquidation declares $(\texttt{Lending}, \texttt{liquidate}, v)$ with v the amount it bids. An oracle update sent from S scores B, while a liquidation scores $v - 1 < B$; from any other sender, declaring `update` scores -1, tying the liquidations that bid nothing. Bids are enforced by the application check: the protocol reads `DECLAREDPARAM` and reverts a transaction if its bid differs from its declaration.

No transaction can score above an update from S, so updates are gated and always execute before all liquidations, whether single- or multi-opinion (Proposition 3.3). If liquidations are single-opinion, they execute from the highest to the lowest declared bid, regardless of priority fees and of anything the attacker injects. If liquidations are instead embedded in multi-opinion transactions, they could be attacked and demoted. Hence a liquidator has the incentive to send the liquidation as a stand-alone, single-opinion transaction.

The liquidation priority race thus moves from the default order, where liquidators compete in priority fees, into the application, which captures the payments itself (and can, for example, pass them on to the position holders or its liquidity providers).

4.4 A CLOB with an operator's "halt trading" and special swaps: sender-restricted and single-opinion clauses

Extend the CLOB example of Section 4.2 to include an operator-only operation that must execute before cancellations. Per market:

$$(\texttt{halt},\ S,\ \{0\},\ \{B+1,-1,B+1,-1\}), \qquad (\texttt{cancel},\ \emptyset,\ \{0\},\ \{B\}), \qquad (\texttt{order},\ \emptyset,\ \{1\},\ \{-1\}),$$

with S the one or two operator addresses. From a sender in S, declaring `halt` scores $B + 1$, above every cancellation; from any other sender, the same declaration scores -1, tying the market orders that pay no priority fee.

Contrast this with the unrestricted clause $(\texttt{halt},\ \emptyset,\ \{0\},\ \{B+1\})$: there, any transaction could phantom-declare `halt` and outrank every cancellation, so cross-market cancellations would stop being cycle-safe, could be welded into cycles, and could be demoted below market orders—introducing the special operation would destroy the protection of Section 4.2. The two cases are illustrated in Figure 7. If instead the `halt` score is reserved to the senders in S, the attacker cannot inject a transaction whose score exceeds a cancellation's at any market, so cancellations, including cross-market ones, remain gated.

Special swaps. Suppose the market also lets its partners and integrators send *special swaps*: market orders that execute before regular ones, but after cancellations. The set of partners may be too large to list in a clause, so the market restricts special swaps at the contract level: they go through a dedicated method `sswap`, which reverts unless the caller is a partner. Note that this does not stop the attacker from *declaring* `sswap`.

From the application designer's viewpoint, the problem is to keep special swaps safe while preserving the protections of Section 4.2 for regular market orders. This can be achieved by penalizing multi-opinion cancellations: only single-opinion cancellations obtain priority, while multi-opinion cancellations are ranked at the bottom of the block, below every market order. Special swaps carry no such restriction, and market orders are ranked by priority fee, as in Section 4.2. Per market, with $B =$ `max_parameter`, v the declared priority fee per gas, and S_o the operator addresses, the clauses assign the following scores (a dash marks a case that cannot arise, since the clause's S is empty):



(a) unrestricted `halt`: cancellations can be outranked

(b) `halt` reserved to S: cancellations remain gated

Figure 7: Sender restriction and the attacker's reach. Left: with an unrestricted `halt` clause, every score up to the halt level is reachable, so cross-market cancellations can be outranked, welded into cycles, and demoted. Right: with the halt score reserved to the senders in S, nothing the attacker can send scores above the cancellation level: cancellations can at most be tied, and remain gated.

		single-opinion		multi-opinion	
method	S	in S	not in S	in S	not in S
`halt`	S_o	$B+2$	-2	$B+2$	-2
`cancel`	$\emptyset$	–	$B+1$	–	-2
`sswap`	$\emptyset$	–	B	–	B
`order`	$\emptyset$	–	$v-1$	–	$v-1$

That is, the clauses are

$$(\texttt{halt},\, S_o,\, \{0\},\, \{B+2, -2, B+2, -2\}), \qquad (\texttt{cancel},\, \emptyset,\, \{0\},\, \{B+1, B+1, -2, -2\}),$$
$$(\texttt{sswap},\, \emptyset,\, \{0\},\, \{B\}), \qquad (\texttt{order},\, \emptyset,\, \{1\},\, \{-1\}).$$

A special swap can be outranked only by halts and single-opinion cancellations, both of which are cycle-safe, so it is cycle-safe as well, whether single- or multi-opinion, as long as it is a special swap at every market it declares.[11] By Proposition 3.2, neither Attack 1 nor Attack 2 can touch special swaps.

Special swaps are cycle-safe but *not* gated—the attacker can outrank them with a single-opinion phantom cancellation—so Proposition 3.3 does not protect them from cheap front-running, especially special swaps spanning multiple markets with the same design. Nonetheless, a call to `sswap` reverts unless the caller is a partner, and cheap front-running from an address registered as a partner can be detected.

Regular market orders, instead, are neither gated nor cycle-safe, because an attacker can insert a multi-opinion transaction that phantom-declares a special swap while scoring low at another market. They are protected as in Section 4.2: single-opinion market orders are covered by Theorem 3.1, and ranking them by priority fee protects them from cheap front-running on the market (Corollary 3.1); multi-opinion market orders remain exposed to Attack 1.

The price is paid by cross-market cancellations, which execute after every market order of their markets: a maker whose cancellations must take effect before market orders has to send them market by market.

4.5 A cross-market batch exchange: settlement after all orders

Consider a CoW-style batch exchange: traders submit and cancel orders throughout the block, and a matching engine, called from a dedicated address, settles the batch at a uniform clearing price. The exchange is the CLOB of Section 4.2 with settlement in place of continuous matching: cancellations first, then orders, then the settlement (a single one, across markets) after every order of the block. Single-opinion orders can bid for their position. Per market:

$$(\texttt{cancel},\, \emptyset,\, \{0\},\, \{2\}), \qquad (\texttt{order},\, \emptyset,\, \{-1, -1, 0, 0\},\, \{1\}), \qquad (\texttt{settle},\, S,\, \{0\},\, \{-B, 1, -B, 1\}),$$

[11] A transaction that combines a special swap on one market with a regular market order on another is not cycle-safe: on the second market, an attacker can insert a multi-opinion transaction that phantom-declares a special swap while scoring low at a third market, placing the victim's transaction on a cycle.

with S the address of the matching engine and B a constant at least as large as `max_parameter`, e.g., $B =$ `max_parameter`. An order declares $(M_k, \texttt{order}, v)$ with v its bid. A single-opinion order scores $1 - v$: the higher the bid, the later the order executes, and an order with an empty parameter field scores 1 and executes in the earliest class. A multi-opinion order scores 1, whatever its bid: only single-opinion orders can bid for their position, for reasons we explain below. Sent from the engine, the settlement scores $-B$, below every order and cancellation ($1 -$ `max_parameter` $> -B$); from any other sender, declaring `settle` scores 1, tying the orders with the lowest possible bid. As in the liquidation auction (Section 4.3), the enforcement of the bids is the application's responsibility: the exchange reads `DECLAREDPARAM` and reverts an order whose bid at execution differs from its declared value.

The settlement, although it spans several markets and is therefore multi-opinion, always executes after all cancellations and orders, whatever the attacker does: no reachable score lies below $-B$, so the settlement has no outgoing unanimity edge, never lies on a cycle, and is never demoted. Because cancellations are gated, they always execute before all orders and the settlement. Wildcard transactions are ordered at the end of the block, after the settlement, and the application may reject them or roll them into the next batch.

Uniform clearing implies that all trades in a market receive the same price. Nonetheless, the last trade has an option value (a "last look"): it is the only trade that can deterministically compute the clearing price from the previously submitted trades and its own. Hence, in this design, there is value to trading last, and the bid captures this value for the application: the highest-bidding single-opinion order executes last, immediately before the settlement.

Again, note that the design requires the `settle` score to be reserved to S. With an unrestricted clause (`settle`, $\emptyset$, $\{0\}$, $\{-B\}$), a transaction declaring `order` and `settle` at every market scores $\min(1, -B) = -B$; by outbidding the matching engine in the priority fee, it executes after every order and immediately before the settlement, and it can call `order` there—a free last look. With the score reserved to S, the same declaration scores $\min(1, 1) = 1$ at every market, since a multi-opinion order cannot bid, and the transaction executes among the earliest orders.

Bids are reserved to single-opinion orders for a related reason. In this exchange, demotion is a reward rather than a penalty: repair places the demoted transaction after the rest of its cycle, and executing late is what orders bid for. If multi-opinion orders could bid, an attacker could place its own order, with no bid, on a cycle with a victim's single-opinion order, using two applications it deploys and a phantom order with a very high bid, which is never charged since it never executes. Repair would then demote the attacker's order, which pays the lowest fee, and place it after the victim's, giving it the last look for free. With bids reserved to single-opinion orders, no multi-opinion transaction the attacker sends can rank below a single-opinion order, so a single-opinion order lies on no cycle, and another order executes after it only by bidding at least as much. The price is that an order spanning several markets cannot bid for its position.

5 Open questions

We have tried to cover all the ways an attacker could exploit APES, and to derive guarantees against them. The system could, however, be exploited in ways we have not anticipated, and we invite the reader to consider whether attacks not discussed here are possible.

With respect to the design of APES, the main open question is the semantics of the preferences. In principle, an application could express its preferences through any function that takes as inputs variables observable by the protocol and returns a score for each transaction that interacts with the application. In practice, it would not be safe for the protocol to allow *any* function: the preferences that can be specified need to be computationally tractable. The specification of Section 2.2 aims to strike a balance between expressiveness and computational simplicity, and it supports all the use cases discussed in Section 4. Nonetheless, there may be use cases we did not consider that would require different preference semantics. Conversely, some of the examples we considered may turn out to be unrealistic, in which case the semantics could be simplified further.

A related question is whether preferences should be allowed to vary within the same contract. As already mentioned, in the implementation discussed above, a contract holding many pools or markets cannot give each of them its own keepers (Sections 4.1 and 4.2). However, several existing protocols, such as Uniswap v4 and Morpho, hold many markets created by different parties in a single contract. To accommodate these

protocols, APES could be extended with *ordering domains*: an identifier d chosen by the application, such as a pool identifier, that transactions declare together with each method. An entry of the declaration then names (A, d, m, v), and each pair (A, d) is a separate column of the preference matrix.

In principle, each domain within the same contract could specify completely different preferences for the same method. However, for the applications discussed earlier, a simpler implementation would keep the clauses defined at the level of the application and the method, and let only the sender set S depend on the domain. That is, each pool, market, or vault can have its own keepers, while the ranking rules follow the design of the application. Also, declared domains could be treated like parameters: the protocol uses them for ordering, and the application checks them, for example through a version of `DECLAREDPARAM` that takes the domain as input.

Finally, the main downside of this extension is that a transaction touching several domains of the same contract becomes multi-opinion, and loses the protection that a single contract with a single set of preferences provides (Section 4.2). To keep the current specification simple, we have not included this extension, and we welcome feedback on whether applications need it.

6 Conclusion

Applications with ordering preferences are currently limited to a few chains (sometimes, their own chain) or must rely on off-chain sequencing mechanisms, which usually means giving up composability.

APES allows applications with ordering preferences to exist natively on Monad. As this document makes clear, APES does not make the trade-off between composability and ordering preferences disappear. It does, however, push it to individual applications and users, who can best resolve it depending on the specific use case.

In particular, from a user's perspective, "simple stuff"—interacting with a single application where ordering matters, plus any number of applications where it does not, such as ERC20 contracts—is safe under very mild assumptions on the application's preferences (Section 3.4), whereas the safety of "complex stuff" depends on the exact preference specifications and on the threat model.

References

[1] Kushal Babel, Fatima Elsheimy, Lioba Heimbach, Mohammad Mussadiq Jalalzai, Tobias Klenze, Jovan Komatovic, Jason Milionis, Mike Setrin, and Victor Shoup. Cadence: Extreme pipelining with multiple concurrent proposers. arXiv preprint arXiv:2607.02275, 2026.

[2] Andrea Canidio. Ordering by unanimity: Giving applications sequencing rights without breaking composability, 2026.

[3] Statemind. Avalanche vulnerability report. `https://statemind.io/blog/avalanche-vulnerability-report-tech`, 2025.

A Proofs

Proof of Proposition 3.2. We structure the proof around the possibility of Attack 1 and Attack 2.

(a) By the definition of cycle-safety, t lies on no cycle formed by the pruned unanimity relation and the nonce edges, whatever the attacker injects, and hence lies in no cyclic component. Repair removes comparisons and imposes new ones only within cyclic components, and creates no cycle through transactions outside them, so it never demotes t and never deletes its comparisons, and the completion respects them.

(b) Let u be a transaction of R that some application A ranks below t. By contradiction, suppose that another application B ranks u above t, so there is no unanimity ranking between u and t. Then u is multi-opinion, and the attacker could also send a transaction u' that declares at B exactly what u declares there, while also declaring some other application that the attacker deploys. Note, however, that u' ranks above t, since the only application scoring both is B. By part (ii) of Definition 3.2, u' belongs to X^*. At the same

time, the attacker can then insert additional transactions on the application it controls so as to place u' on a cycle, which contradicts part (i) of the same definition.

Hence u scores at most as much as t at every application scoring both, and strictly less at A, so t ranks above u. □

Proof of Proposition 3.3. Since x is top cycle-safe, it lies on no cycle and is never demoted, by Proposition 3.2. The ancestors of x rank above it, directly or indirectly, so by (ii) of Definition 3.2 they are top cycle-safe as well; like x, they lie on no cycle, so repair imposes no comparison on them, and their predecessors in the repaired graph are the transactions that rank above them. No transaction of R ranks above x or above one of its ancestors, so the predecessors of x in the repaired graph, direct or indirect, are exactly its ancestors. None of them is a wildcard transaction, since the pruned unanimity relation links no wildcard transaction. At each step, Kahn's algorithm appends the available non-wildcard transaction that is highest in the default order. Let y be appended before x. If x was available at that step, y precedes x in the default order. Otherwise, x had a predecessor p_1 not appended yet; if p_1 was not available either, it had a predecessor p_2 not appended yet, and so on. Since the ancestors of x lie on no cycle, the walk visits distinct transactions and stops at an ancestor that was available, which y therefore precedes in the default order. □

Proof of Corollary 3.1. Let y call m on A and execute before x, and let v_y and v_x be the values the two transactions declare for m. Since y calls m, it must declare it, so $U_A(y) \le a \cdot v_y + b$, while $U_A(x) = a \cdot v_x + b$. If $v_y < v_x$, then $U_A(x) > U_A(y)$, and Theorem 3.1 places x before y: a contradiction. □

B The preference registry interface

This appendix reproduces the Solidity interface of the preference registry precompile of Section 2.2, together with the opt-in interface that an application must implement before it can register. The `Clause` layout represents the tuple $(m, S, \{a_1, \dots, a_4\}, \{b_1, \dots, b_4\})$ of Section 2.2: `matchKind` and `selector` encode m (with `AllOtherSelectors` marking the catch-all clause), `senders` is S, and the four `LinearScore` fields hold the pairs (a_i, b_i), one for each case of the scoring rule.

```
pragma solidity ^0.8.20;

/// @title APES application opt-in
/// @notice Implemented by every application that registers preferences.
interface IAPESApplication {
    /// @notice Signals consent to APES registration.
    /// @dev Must return its own selector, IAPESApplication.isApesEnabled.selector
    ///      (0x45719097); any other result, or a revert, means no consent.
    ///      The registry calls it on the caller of register and revoke.
    function isApesEnabled() external view returns (bytes4);
}

/// @title APES Preference Registry
/// @dev Precompile at PREFERENCE_REGISTRY. Accepts CALL, and STATICCALL for
///      view functions.
interface IAPESRegistry {
    enum MethodMatch {
        /// @dev Matches only the clause's selector.
        Exact,
        /// @dev Matches four-byte selectors not named by an Exact clause.
        AllOtherSelectors
    }

    /// @dev score(v) = slope * v + intercept.
    ///      v is the transaction's declared parameter for this method;
    ///      an empty parameter field is treated as zero.
    struct LinearScore {
        int128 slope;
```

```
    int128 intercept;
}

/// @dev "single" and "multi" refer to the number of declared
///      opinionated applications, not the number of methods or calls.
///      Membership in senders uses the transaction sender, not
///      necessarily the caller of an inner EVM call.
struct Clause {
    MethodMatch matchKind;

    /// @dev Ignored when matchKind is AllOtherSelectors.
    bytes4 selector;

    /// @dev The transaction-sender set S, shared by all four cases.
    address[] senders;

    LinearScore singleInSet;
    LinearScore singleOutOfSet;
    LinearScore multiInSet;
    LinearScore multiOutOfSet;
}

struct Preference {
    /// @dev Complete preferences; an empty list represents a revocation.
    Clause[] clauses;

    /// @dev Block from which this preference change is in force.
    uint256 activationBlock;
}

/// @notice Schedules the caller's complete replacement preferences.
/// @dev Invalid preferences revert. Registration requires:
///      - The caller to opt in (IAPESApplication.isApesEnabled).
///      - Between one and max_clauses clauses.
///      - Unique selectors among Exact clauses.
///      - At most one AllOtherSelectors clause.
///      - At most max_senders_per_clause senders per clause.
///      - Every scoring function to fit int128 at both v = 0 and
///        v = max_parameter.
///
///      A call in block b schedules activation at b + k, where the
///      protocol delay k is at least the execution delay. Until then,
///      the change is pending. Only the last surviving register/revoke
///      update for this application in the same block counts.
///      Updates in different blocks may coexist as pending changes.
/// @param clauses Complete replacement preferences; must be nonempty.
/// @return activationBlock The future block from which they are in force.
function register(Clause[] calldata clauses)
    external
    returns (uint256 activationBlock);

/// @notice Schedules removal of the caller's preferences.
/// @dev Requires the caller to opt in (IAPESApplication.isApesEnabled).
///      A call in block b schedules revocation at b + k.
///      The change is represented by a Preference with empty clauses.
///      The same activation delay and same-block last-update rule
///      apply as for register.
/// @return activationBlock The future block from which revocation is in force.
function revoke()
    external
```

```
        returns (uint256 activationBlock);

    /// @notice Returns the application's preferences in force, if any,
    ///         followed by its pending preference changes.
    /// @dev Entries are ordered by activationBlock. This is not a history
    ///      of superseded preferences. The first entry may be pending
    ///      when the application has no preferences currently in force.
    ///      An empty clauses array denotes a revocation.
    /// @param app The application address.
    /// @return preferences The preferences in force, if any, and pending changes.
    function getPreferencesOf(address app)
        external
        view
        returns (Preference[] memory preferences);
}
```

C Other open implementation details

This appendix lists the implementation details that remain to be specified or may need further discussion, grouped by the part of APES they concern.

C.1 The declaration

Encoding The type number, the field encoding, the signing hash, and the maximum encoded size of the list are, for the moment, unspecified.

Declaration-generating RPC An RPC call that simulates t and returns the declaration it needs, in the manner of `eth_createAccessList`. Front ends can generate the list themselves, but bots do not use front ends, so the call would likely be helpful. Also, users who do not want their transactions treated as wildcard transactions need wallets that support the new transaction type, even when the list is empty.

Gas values The values of `gas_per_declared_application` and `gas_per_declared_method`.

The pseudo-method `receive` We could allow transactions to declare (and applications to rank) the pseudo-method `receive`, via a special encoding that lies outside the range of four-byte selectors, for example using an extra bit. The open question is how to check such a declaration at execution: a frame with empty calldata may be handled by the application's `receive()` function, by its `fallback()` function, or by neither, and the protocol sees only the frame's calldata, not which function handles it. Relatedly, under the current specification a frame with fewer than four bytes of calldata carries no method, so it is neither ranked nor checked.

Account abstraction A transaction relayed under ERC-4337 bundles the operations of several accounts, which cannot be ordered separately. A bundle can declare every opinionated application and method that its operations may call, but this is impractical, so bundles will likely use the wildcard.

C.2 The preference registry

The value of `max_clauses` The maximum number of clauses a registration may hold.

Registry gas The gas the precompile charges for `register`, for `revoke`, and for `getPreferencesOf`.

The value of `max_senders_per_clause` The maximum number of addresses in the set S of a clause.

C.3 The execution order

Batch demotion and worst-case translation cost Demoting one transaction at a time may degenerate toward quadratic repair cost on adversarial blocks. An alternative is to demote in batches: cap the number of demotion rounds (say, at ten), so that a cyclic component of 100 transactions is repaired by demoting 10 transactions at a time. At the extreme, cycle repair can occur in two steps (first demote all multi-opinion transactions and, if a cycle remains, all transactions with a nonce predecessor on it) or even in a single step, demoting both groups at once.

Batching changes the output relative to one-at-a-time demotion, because fewer unanimity links are preserved: it is a mechanism variant, not an optimization. The granularity of demotion must therefore be specified ex ante—validators cannot switch to batches under load. Whatever the choice, the specification needs a hard bound on translation time for a full block of maximally declaring transactions, with `gas_per_declared_application` and `gas_per_declared_method` set to price that worst case.

Optimistic approach Instead of checking for the existence of cycles and then running Kahn's algorithm, run Kahn's algorithm first, optimistically: it consumes the whole block if and only if G is acyclic, so it is also a cycle detector. On a stall, run a cycle-detection algorithm (e.g., Tarjan's strongly-connected-components algorithm) on the unconsumed remainder only, which must contain every cyclic component; note that a stalled remainder may hold several independent cyclic components. After each demotion, re-run Tarjan's algorithm on the affected component only. After all cycles are broken, restart Kahn's algorithm.

An important implementation detail: Kahn's algorithm needs to restart from the top after the cycle-repair step, and cannot just resume from where it stalled. Take three transactions: a isolated with fee 1, and $b \rightleftarrows c$ with fee 10 each. The only available transaction is a, so the optimistic run emits a and stalls; repair demotes c, freeing the survivor b; a fresh run has both a and b available at position one, and b pays more, so the specified order is b, a, c. Resuming would give a, b, c, which extends the repaired relation but breaks the fee property: b is preceded by a cheaper transaction that no preference placed before it. Telling the safe case from the unsafe one requires a check of its own, so the simple rule is to restart. Only the first optimistic run is ever discarded: all cycles are broken before Kahn's algorithm runs again.

This optimistic approach outputs the same order as Algorithm 1, which runs repair before completion, since the final run operates on the fully repaired graph. If no cycle is present, the procedure requires a single run of Kahn's algorithm; if cycles are present, the initial run is wasted. The alternative, breaking all cycles before running Kahn's algorithm, costs an extra run of cycle detection when there are no cycles and is efficient otherwise. The optimistic approach is preferable if cycles are expected to be infrequent.

Computational cost of the ordering algorithm The computational complexity, running time, and memory footprint of the different versions of the ordering algorithm—one-at-a-time versus batch demotion, and repair-then-complete versus the optimistic approach—are still to be quantified, both asymptotically and on realistic and adversarial blocks.

C.4 Execution

`DECLAREDPARAM` cost The gas `DECLAREDPARAM` charges, and its opcode byte.