Execution Transparency: Low-Trust Networking through Secure, Public Logs

Paul Frazee — Jan 18, 2022

1 Introduction

Decentralized networks require open participation. To counteract the effects of hostile actors, the network must be designed defensively. A common defensive technique now used by decentralized networks is "trustlessness". A trustless network constrains the activity of each participant such that good actors are able to transact while bad actors are either quickly detected or are prevented entirely from executing an attack.

Blockchains rely on decentralized Byzantine-Fault-Tolerant consensus, cryptographic structures, and economic incentives to create trustlessness. Their notable downsides include reliance on large networks to provide majority honesty, per-transaction fees, global consensus on all transactions, and wasteful hash mining in PoW schemes. While blockchains have many useful properties, the cost and low throughput indicate that they're not suitable for all use-cases.

Blockchains use trust minimization to block the actions of malicious actors. By choosing instead to merely expose malicious actors, we can make a simpler, more efficient system. In this paper, we present an alternative model for trust minimization called "Execution Transparency" (or ET) which provides low cost, high throughput transactions while maintaining useful security guarantees. We highlight the Hypercore Protocol[1] as a primitive for implementing Execution Transparency and reference the Vitra[2] implementation for specific suggestions on implementing ET.

Execution Transparency has several use-cases. As with Certificate Transparency, it can be used to provide auditable registries of credentials or software packages. As with Smart Contracts, it can be used to enforce multi-party agreements such as voting rules on changes to social software (user governance). Execution Transparency may also provide an effective framework for Oracles which are reputational by nature.

2 Execution Transparency

Execution Transparency is a generalization of Certificate Transparency[3] (or CT) a widely-used protocol for TLS certificate issuance. Like CT, Execution Transparency uses publicly auditable, append-only, untrusted logs of all transactions. ET does not stop all invalid transactions, but it makes them easily detectable so that third parties may respond.

Execution Transparency generalizes Certificate Transparency through programs called "contracts" which are similar to smart contracts in blockchains. Database hosts (known as "executors") operate the contracts by producing a

single output log (the "index") which represents the current state and all execution history. Additional writers (known as "participants") each produce an input log (or "oplog") to the contract. Auditors (known as "monitors") validate execution by replaying the oplog(s) against the contract and comparing against the published index log.

As operations are written to the participant oplogs, the executor reads the operations and applies them according to the contract. Only the executor can publish new state to the contract's index log, so the participants are unable to violate the contract. To ensure correct application by the executor, a third party may replay the oplogs and index log against the contract in order to verify their validity. If an output generated in the verification process does not match the current state, a "breach notice" can be published indicating that the executor has violated the contract and should not be trusted.

2.1 Executors

ET databases are operated by users known as "executors." Executors may be one or more device so long as they produce a linearized ordering of transactions. Malicious behavior (contract breaches) by executors cannot be prevented, but can be detected afterwards by third-party auditing.

2.2 Participants

Users which transact with an ET database are referred to as "participants." Participants produce logs which are declared as inputs to the database. Unlike executors, malicious behavior by participants is prevented. Any attempted transaction by a participant which violates the contract will be rejected by the executor.

2.3 Monitors

Any third party which validates an ET contract's logs is known as a "monitor." Monitors do not need a pre-defined relationship with the executor or the contract; any device may sync the logs of an ET contract and verify their correctness. The processes which monitors follow mimic the executor processes; the key difference is that monitors have no authority to publish updates to the output log. Instead, their job is to watch for breaches by the executor and warn participants if a breach is found.

2.4 Public Auditable Logs

Execution Transparency relies on logs with the following properties:

- **Append-only**. Published messages must only be added; they may not be mutated or deleted.
- **Signed**. All messages must be signed by a keypair which represents the executor or participants (if participants produce their own oplogs).

The suggested cryptographic construction of ET logs is similar to that of Certificate Transparency logs. A Merkle Tree is used to efficiently identify the state of a log at a given point in its history. Signed root hashes may then be compared over a gossip network to ensure equivalent histories; if divergent histories have been published, the signed roots can be used as proofs that the append-only constraint has been violated.

From RFC 6962, "Certificate Transparency"[3]:

> The append-only property of each log is technically achieved using Merkle Trees, which can be used to show that any particular version of the log is a superset of any particular previous version. Likewise, Merkle Trees avoid the need to blindly trust logs: if a log attempts to show different things to different people, this can be efficiently detected by comparing tree roots and consistency proofs.

The Hypercore Protocol[1] provides a usable construction of signed, append-only logs using these techniques. These logs, known as "Hypercores," are identified using the public key of the keypair which signs all logs. A log-gossiping network is provided in the Hypercore Protocol which includes distributed peer-discovery of Hypercores.

While the oplogs should only act as a sequence of operations, the index log may include embedded indexes which enable efficient reads of the contract state. These indexes have been explored heavily in the Hypercore Protocol, including embedded B-Trees (Hyperbee[4]) and rolling hash-array mapped tries (Hypertrie[5]). Using these indexes, it is possible for the contract output methods to behave as key-value stores or filesystems with efficient sparse synchronization of the index log to achieve reads of the current state.

2.5 Contract Programs

Each ET database declares a "contract program" in its index log. The contract is responsible for defining an API to the database. Contract methods are used both to read the current state and to produce operations for processing.

In addition to the API, each contract exports a pure function known as the "apply" function for deterministically producing the index from the oplog(s). Deterministic execution is a key requirement of the apply function; repeated executions of the oplogs at any state must result in an equivalent output log.

To achieve a deterministic mapping, the following properties must apply:

- **Purity**. Side-effects such as randomness, timestamps, or external data must not be introduced in the apply function.
- **Reproducable Ordering**. Operations must complete in the same order for each execution. If asynchronous operations are permitted, they must resolve in a consistent order.

Some of these properties may be enforced by the contract runtime (e.g. by providing no `random()` in the ABI).

2.6 Contract Audits

To audit a contract, a monitor must synchronize the full known state of a contract's oplogs and index log. Monitors should persistently sync the logs to ensure their availability.

Auditing does not rely on public operation. Contracts may be executed privately between interested parties so long as each participant is capable of monitoring the logs. It's recommended to use a persistent gossip protocol to sync the logs to multiple third-party monitors as transactions occur, as this will reduce the potential for log rewriting.

Once the oplogs and index log are synchronized, a monitor validates the contract's execution by replaying the oplog(s) against the contract's apply function. The monitor then compares its generated index log against the executor's published index log. If the generated message content is found to differ from the equivalent published message in the index log, the monitor should regard the executor as "in breach." As a contract-breach is reproduceable, the monitor can give notice of the breach by sharing the sequence number of the index log in which the breach occurred along with the inclusion proofs (root hash and signature) of the relevant messages.

In addition to contract breaches, monitors must watch for violations of the append-only constraint as described in the "2.4 Public Auditable Logs." In the event of a contract breach or append-only violation, monitors should contact other participants and take action to resolve the issue. The nature of the breach should be evaluated to determine if the cause was technical (e.g. a bug in the contract) or malicious. The contract or the executor should be replaced accordingly.

2.7 Inclusion Proofs

Every message produced on the append-only log can produce an "inclusion proof" which is comprised of the log's public key, the message's sequence number, a hash of the log's state at the length of the message, and a signature on the hash. Using this information it's possible to demonstrate to third parties that the message was published by the log, as the signature proves authorship and the hash confirms the log state. If any log should break the append-only invariant, such a proof could be used to demonstrate the violation. By recording inclusion proofs for all messages involved in a transaction, clients of an ET database can demonstrate the existence of a transaction to third parties.

It may be possible to expand the inclusion proofs to include higher-level semantics in the authenticated data. For example, a proof could include an assertion such as "the output log now includes entry K with value V." These

proofs could be shared outside of the contract and quickly verified by third parties without consulting the current contract state. Note however that "freshness" is not indicated by inclusion proofs and so the current state of a database may need to be consulted in some cases.

3 Processes

The implementation details of Execution Transparency may differ. In this section, we will outline the processes used by the Vitra reference implementation[2].

3.1 Index layout

Vitra's index log has a set of fixed entries:

Key	Usage
`.sys/contract/source`	The source code of the contract
`.sys/inputs/{pubkey-hex}`	Declarations of oplogs
`.sys/acks/{pubkey-hex}/{seq}`	Acknowledgements of processed ops

Entries under `.sys/acks/` can not be modified by the contract. Acks are stored in the output logs to ensure atomicity of transaction-handling.

3.2 Initialization flow (executor)

The executor host initializes a contract with the following steps:

- The index is created.
- The Hyperbee header is written to the index log. (block 0)
- The `.sys/contract/source` entry is written to the index log with the source code of the contract. (block 1)
- Any number of `.sys/inputs/{key}` entries may be written.
- An empty entry is written at `.sys/acks/genesis` indicating that initialization is complete and that all further entries will be dictated by the contract.

3.3 Operation processing flow (executor)

The executor host watches all active oplogs for new entries and enters the following flow as each entry is detected:

- If the contract exports a `process()` function
  - Call `process(op)` and retain the returned metadata.
- Create a new `ack` object which includes:
  - The oplog pubkey.
  - The op sequence number.
  - The root hash of the oplog.

    - A local timestamp.
    - Metadata returned by `process()`.
- Call `apply()` with the following arguments:
    - `tx` An object with `put(key, value)` and `del(key)` operations for queueing updates to the index.
    - `op` The operation.
    - `ack` The generated ack.
- If the `apply()` call:
    - Returns a resolved promise
        * Set `ack.success` to true
    - Returns a rejected promise
        * Set `ack.success` to false
        * Set `ack.error` to a string (the message of the error)
        * Empty the `tx` queue of actions
- Prepend the `ack` entry to the `tx` with a path of `.sys/acks/{oplog-pubkey-hex}/{seq}`.
- Atomically apply the queued actions in `tx` to the index.
- Iterate the actions in `tx` using offset `i`:
    - If the `tx[i]` key is `.sys/contract/source`:
        * Replace the active VM with the value of `tx[i]`.
    - If the `tx[i]` key is prefixed by `.sys/input/`:
        * If the `tx[i]` action is `put`:
            · Add the encoded oplog to the active oplogs.
        * If the `tx[i]` action is `delete`:
            · Remove the encoded oplog from the active oplogs.

3.4 Transaction flow (participant)

The transaction flow is divided into "creation" and "receiving" as time may pass between the initial op-generating call and result processing by the executor.

Creating the transaction:

- Sync the index head from the network.
- Initialize the contract VM with the current contract source.
- Capture the index root proof as `indexProof`.
- Call the specified contract method.
- Respond with the following information:
    - Was the call successful?
    - The response or error returned by the call.
    - `indexProof`
    - An array of all operations generated, including:
        * The oplog root proof.
        * The operation value.

Receiving transaction result:

- Iterate each operation generated by the transaction as `op`:

  - Await the matching ack in the index.
  - Fetch the matching ack as `ack`.
  - Fetch all mutations to the index matching `ack` as `mutations`.
  - Fetch the index root proof at the seq of the `ack` as `indexProof`.
  - Respond with the following information:
    * `op`
    * `ack`
    * `mutations`
    * `indexProof`

3.5 Full verification flow (monitor)

Verification occurs by iterating the index's log and comparing published tx-results against generated tx-results. All transactions are preceded by an ack entry, so the majority of the flow is looking for acks at expected places, comparing all updates that result from the message indicated by the ack, and then skipping forward.

Verify initialization entries:

- Verify that `idxLog[0]` is a valid Hyperbee header.
- Read contract source from `idxLog[1]` and instantiate the VM with it.
- Create a map `processedSeqs` to for each active oplog with each entry initialized at `-1`.
- Set `idxSeq` to `2`
- While `idxSeq < idxLog.length`:
  - If `idxLog[idxSeq]` key is `.sys/ack/genesis`, exit while loop.
  - If `idxLog[idxSeq]` key is not `.sys/inputs/{key}`, fail verification.
  - Increment `idxSeq`
- Increment `idxSeq`

Verify operations:

- While `idxSeq < idxLog.length`:
  - Set `ack` to `idxLog[idxSeq]`
  - If `ack` key does not match `.sys/ack/{pubkey}/{seq}`, fail verification.
  - If `ack` write-type is not `put`, fail verification.
  - If the `{seq}` segment of the key does not equal `processedSeqs[pubkey] + 1`, fail verification.
  - Fetch the `op` from the oplog specified by the `ack` value.
  - Rewind VM `index` state to `idxSeq`.
  - Call `apply()` with the following arguments:
    * `tx` An object with `put(key, value)` and `del(key)` operations for queueing updates to the index.
    * `op` The operation.
    * `ack` The `ack` value.

- Set `newContractSource` to `null`.
- Set `oplogChanges` to an empty array.
- Iterate the actions in `tx` using offset `i`:
  - If the `tx[i]` type does not equal the `idxLog[idxSeq + i]` type, fail verification.
  - If the `tx[i]` key does not equal the `idxLog[idxSeq + i]` key, fail verification.
  - If the `tx[i]` value does not equal the `idxLog[idxSeq + i]` value, fail verification.
  - If the `tx[i]` key is `.sys/contract/source`, set `newContractSource` to the `tx[i]` value.
  - If the `tx[i]` key matches `.sys/inputs/{pubkey}`, add the value to `oplogChanges`.
- Set `processedSeqs[pubkey]` to the `{seq}` segment of the `ack` key.
- Increment `idxLogSeq` by `tx.length + 1`.
- If `newContractSource` is not `null`:
  - Replace the active VM with `newContractSource`
- Iterate each entry in `oplogChanges`:
  - Add or remove oplogs according to the encoded change.

4 Technical and Security Considerations

4.1 Log Availability

Monitors require access to the full histories of a contract's oplogs and index log to perform validation. If some part of history is not made available, then validation can not be accomplished. If an executor can not or will not produce the full histories of logs, it may be necessary to regard the executor as untrustworthy.

4.2 Transaction Censorship and Ordering

As ET contracts are operated by a sole-executor, it is possible for a participant's messages to be rejected or ignored without recourse (censorship). In cases where timeliness is a factor, the ability to censor messages may prove harmful. This concern can be mitigated by introducing a Paxos-like consensus protocol for multiple executors. Likewise, the executor has sole discretion over the order by which messages are processed. As the execution order can have significant effects on the result of a transaction, it is important that participants consider the effects of message ordering on their use-case.

4.3 Contract Sandboxing

A key requirement for the contract runtime is a secure sandbox for executing the untrusted contracts. Interpretted VMs such as Javascript or WASM are recommended, as are OS restrictions on the execution process (seccomp, seatbelt, etc).

4.4 Incentive and Reputation Models

No incentive model for contract execution or monitoring is defined in this paper. Incentives are important in practice but their design is orthogonal to ET's core mechanics and can take many forms, including social altruism within semi-trusted groups or per-transaction payment through cryptocurrencies. Therefore we chose to leave incentive models unspecified in this paper.

No formal reputation model is specified in this paper for similar reasons. Implementators may create automated reputation networks based on honest and dishonest execution, but such a network is not strictly necessary for Execution Transparency as reputation can be managed manually by end-users.

5 Use cases

5.1 Registries

Execution Transparency can improve the security of sensitive registries such as user credentials or software packages. Third party monitors can watch updates to the registry to ensure that no unauthorized changes occur, and data distributed by an ET's logs provide a strong guarantee that tampering has not occurred (i.e. by including a modified binary to some requests). These benefits are similar to what Certificate Transparency provides to TLS certificate issuance.

Example: Simple public-key registry. The registry service maintains a mapping of usernames to public keys. A governing contract is specified with the following methods: `get(username)`, `put(username, publicKey)`. Calls to `put()` occur through RPC and authentication is handled off-contract by the service. Participants monitor the output log to ensure no unauthorized key-changes occur.

Example: Multisig public-key registry. Expands on the simple public-key registry by requiring verifications of updates by participants. A governing contract is specified with the following methods:

- `get(username)`
- `put(username, publicKey)`
- `verify(username, publicKey)`

Participants transact using oplogs while end-users transact using RPC to the participants. Logic within the contract `apply()` dictates that each `put()` must be followed by one `verify()` from a separate participant before the updating the username mapping. Participants then monitor the output log to ensure no unauthorized key-changes occur. (The processes for changing the participants have been omitted for brevity.)

5.2 User Governance

ET contracts can be used to stipulate multi-party governance over shared resources. For example, an ET contract could include rules for updating the contract itself; these rules would require a majority vote by participants in the contract before the code could be updated. Similar schemes could be used to govern changes to software which is attached to the contract (such as the UI of an application) giving users a say in the development of shared social applications.

Example: Governed fileset. The fileset service maintains a collection of files which requires a vote on all changes ("commits"). The contract is created with the following methods:

- `listFiles(path)`
- `getFile(path)`
- `listProposals()`
- `proposeCommit(proposalId, files)`
- `acceptCommit(proposalId)`
- `rejectCommit(proposalId)`

New "commit proposals" may be created by any participant with `proposeCommit()`, placing it in the "proposed" state. If a majority of participants publish an `acceptCommit()` message the proposal enters the "accepted" state and the file mapping is updated in the apply function. If a majority publish a `rejectCommit()` message the proposal enters the "rejected" state. Participants can change their vote only if the proposal is in the "proposed" state.

5.3 Oracles

"Oracles" are actors who provide information to networks which come from a third party. They might be used to capture pricing information around commodities or stocks, sensor data from IoT devices, web page responses, and more. Critically, Oracles rely on trust in the actor to provide actionable information.

ET contracts could be used to publish Oracle data auditably. Oracle output logs would be referenced by other ET contracts, ideally including the signed root hash of the referenced log-message to ensure authenticity. (The details of using ET oracles feeds in blockchain applications requires future exploration.)

Some ET contract schemes could combine data captures from oracles run by multiple different parties, adding an extra layer of trustworthiness. The contract could decide how to handle disagreement between the oracles according to the application, potentially rejecting disagreements or blending them if appropriate.

Example: Website capture. The website capture service would provide a snapshot of the response data from a URL at a regular period. A contract would provide the following functions:

- `config(interval, agreementWindow)`
- `get(url, timestamp)`
- `put(url, timestamp, responseData)`

Each put call must possess a timestamp which is greater than the previous put's timstamp by the `interval` amount. The executor includes its local timestamp when processing the `put()` and the proposed timestamp must be within `agreementWindow` milliseconds of the executor's local timestamp. Readers can fetch the snapshot of the contract's target URL at any time by calling the `get()` function, which will round the given timestamp down to the closest available capture.

6 Future work

6.1 Inter-contract Linking

ET contracts as described in this paper are self-contained; they provide an index log based strictly on the messages of the oplogs. It should however be possible for ET contracts to reference data from other contracts using a URL scheme. Depending on the guarantees of the URL scheme, it might be sufficient to encode only the URL so long as there's a strong guarantee the referenced data is accessible and deterministically resolved. Alternatively the contract may choose to embed the referenced data as a captured side-effect.

6.2 Cross-contract RPC

As each contract specifics an API, it is possible that RPC methods – that is, methods which can be accessed by third parties rather than owners of the oplogs – could be encoded in ET contracts. This can be used to enable contracts to send arbitrary API calls to each other.

Any such scheme is contingent on authentication and spam prevention as a naive approach would inevitably lead to DoS attacks. A future area to explore is the use of reputation networks and payment systems to effectively meter the resource usages of RPC.

6.3 Multi-authority Execution

Execution Transparency relies on a deterministic, linear execution of the contract's transactions. In this paper, we suggest appointing a sole authority (the executor) to ensure these properties. Future work may explore how multiple executors can provide these guarantees. A working multi-executor scheme could be used to mitigate censorship risks and could provide a pathway to breach-resolution without having to replace the contract.

It may be possible to relax the linearized consistency constraint of contract inputs to a less strict consistency model such as causal ordering or other forms of eventual consistency. Any relaxation of the consistency model needs

to consider how the contract program's semantics will be affected. For instance, causal ordering cannot enforce some invariants in a program's business logic (e.g. "foreign key" constraints).

6.4 Bounded State Growth

As input and output logs are append-only, there is no mechanism for removing historic state. While embedded indexes can counteract the performance costs of state reads on large output logs, executors and monitors must maintain all historic state to correctly execute and audit ET contracts. Future work should explore mechanisms for truncating history without sacrificing the security properties of contracts.

7 Conclusion

Reducing the trust required for participation is key to ensuring open, decentralized networks. In this paper, we've given a high-level overview of a solution for externally-auditable services whose operations are constrained by software contracts. This transparency ensures that bad actors are quickly caught and stopped from harming the overall network.

While Execution Transparency provides weaker trust-minimization than blockchains, it provides significant performance improvements by relaxing some constraints. As with blockchains, ET may not be the ideal solution for all use-cases, but can be a much better fit for applications in which global consensus is not required.

References

1. *Hypercore Protocol*. (n.d.). https://hypercore-protocol.org/
2. *Vitra*. (n.d.). https://github.com/pfrazee/vitra
3. *RFC 6962, "Certificate Transparency"*. (2013). https://datatracker.ietf.org/doc/html/rfc6962
4. *Hyperbee*. (n.d.). https://github.com/hypercore-protocol/hyperbee
5. *Hypertrie*. (n.d.). https://github.com/hypercore-protocol/hypertrie