iBatch: Saving Ethereum Fees via Secure and Cost-Effective Batching of Smart-Contract Invocations

Ethereum charges a high unit cost for data movement (via transactions) and for data processing (via smart-contract execution). To optimize the transaction fee, a natural idea is to batch multiple smart-contract invocations in a single transaction so that the fee can be amortized. However, the third-party server that assumes the job of batching can mount attacks to forge, replay, modify and even omit the invocations. A naive design to defend against the invocation manipulations by validating invocations on blockchains incur high cost that offsets the saving intended by the batching.

We present iBatch, a middleware system retrofittable on an unmodified Ethereum network to batch smart-contract invocations securely against an off-chain relay server. iBatch achieves low-cost invocation validation by offloading the majority of validation work to the caller EOAs (assumed to be available here) and leaving on-chain validation stateless.

We conduct extensive cost evaluations, which shows iBatch saves 14.6% ~ 59.1% Gas cost per invocation with a moderate 2 minute delay and 19.06% ~ 31.52% Ether cost per invocation with a delay of 0.26 ~ 1.66 blocks

Summary

- We design a security batching framework. it prevents the Batcher from forging or replaying a caller's invocation in a batch transaction. It also ensures the Batcher's attempt to omit a caller's invocations can be detected by the victim caller. In addition, iBatch can be extended to prevent a denial-of-service caller from delaying a batch.
- We propose mechanisms and policies for the Batcher to properly batch invocations for design goals in cost and delay.

Method:

- The iBatch protocol works in the following four steps:
- In the batch time window, a caller submits the i-th invocation request, denoted by call_i, to the Batcher service.
- By the end of the batch time window, the Batcher prepares a batch message bmsg and sends it to the callers for validation and signing. Message bmsg is a concatenation of the N requests, call_i's, their caller nonces nonce_i 's, and Batcher account's nonce, nonce_B. Each of the callers checks if there is one and only one copy of its invocation in the batch message. After a successful check of equality, the caller signs the message bmsg_sign, that is, bmsg without callers' nonces. She then sends her signature to the Batcher.
- Batcher includes the signed batch message in a transaction's data field and sends the transaction, called batch transaction, to be received by the Dispatcher smart contract.
- In function dispatch_func, smart contract Dispatcher parses the transaction and extracts the original invocations call_i before forwarding them to callees. smart-contract Dispatcher internally verifies the signature of each extracted invocation against its caller's public key; this can be done by using Solidity function ecrecover(call$i$,sig$i$,account$i$).

[

Screen Shot 2021-06-27 at 1.11.25 PM

1718×846 55.5 KB

](https://ethresear.ch/uploads/default/original/2X/2/29e89d5bbd202f1550bb31b24e7319f9e0257481.png)

- In the batch time window, a caller submits the i-th invocation request, denoted by call_i, to the Batcher service.
- By the end of the batch time window, the Batcher prepares a batch message bmsg and sends it to the callers for validation and signing. Message bmsg is a concatenation of the N requests, call_i's, their caller nonces nonce_i 's, and Batcher account's nonce, nonce_B. Each of the callers checks if there is one and only one copy of its invocation in the batch message. After a successful check of equality, the caller signs the message bmsg_sign, that is, bmsg without callers' nonces. She then sends her signature to the Batcher.
- Batcher includes the signed batch message in a transaction's data field and sends the transaction, called batch transaction, to be received by the Dispatcher smart contract.
- In function dispatch_func, smart contract Dispatcher parses the transaction and extracts the original invocations call_i before forwarding them to callees. smart-contract Dispatcher internally verifies the signature of each extracted invocation against its caller's public key; this can be done by using Solidity function ecrecover(call$i$,sig$i$,account$i$).

[

Screen Shot 2021-06-27 at 1.11.25 PM

1718×846 55.5 KB

](https://ethresear.ch/uploads/default/original/2X/2/29e89d5bbd202f1550bb31b24e7319f9e0257481.png)

- When running legacy smart contracts on iBatch, the smart contracts need to be rewritten to authenticate the internal calls from Dispatcher smart contract. We propose source code and bytecode rewriting techniques.

Results

- We conduct extensive cost evaluations, which shows iBatch saves 14.6% ~ 59.1% Gas cost per invocation with a moderate 2-minute delay and 19.06% ~ 31.52% Ether cost per invocation with a delay of 0.26 ~ 1.66 blocks.

Discussion & Key Takeaways

- We design a lightweight security protocol for batching of smart contract invocations in Ethereum without trusting third-party servers
- We design a middleware system implementing the above protocol and propose batching policies from conservative to aggressive batching.
- We built an evaluation platform for fast and cost-accurate transaction replaying and constructed transaction benchmarks on popular Ethereum applications.

Implications and Follow-ups

- We note that a recent Ethereum Improvement Proposal (i.e., EIP- 3074) may facilitate the integration of iBatch with legacy smart contracts. In the future, if EIP-3074 is adopted by the Ethereum protocol, it would allow integrating iBatch with legacy smart contracts without rewriting.

Applicability

- Our iBatch is a general protocol for all kinds of Dapp transactions with a tradeoff between cost-saving and delay. It is a very useful middleware for Ethereum users. For the Dapp team or institutional caller, which may send a lot of transactions during a short time, iBatch can help them save cost.

The technical details are in our paper:https://tristartom.github.io/docs/fse21-ibatch.pdf