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

Yibo Wang
ywang349@syr.edu
Syracuse University
Syracuse, NY, USA

Qi Zhang
qzhang71@syr.edu
Syracuse University
Syracuse, NY, USA

Kai Li
kli111@syr.edu
Syracuse University
Syracuse, NY, USA

Yuzhe Tang*
ytang100@syr.edu
Syracuse University
Syracuse, NY, USA

Jiaqi Chen
jchen217@syr.edu
Syracuse University
Syracuse, NY, USA

Xiapu Luo
csxluo@comp.polyu.edu.hk
HKPU
Hong Kong, China

Ting Chen
brokendragon@uestc.edu.cn
UESTC
Chengdu, Sichuan, China

ABSTRACT

This paper presents iBatch, a middleware system running on top of an operational Ethereum network to enable secure batching of smart-contract invocations against an untrusted relay server off-chain. iBatch does so at a low overhead by validating the server's batched invocations in smart contracts without additional states. The iBatch mechanism supports a variety of policies, ranging from conservative to aggressive batching, and can be configured adaptively to the current workloads. iBatch automatically rewrites smart contracts to integrate with legacy applications and support large-scale deployment.

For cost evaluation, we develop a platform with fast and cost-accurate transaction replaying, build real transaction benchmarks on popular Ethereum applications, and build a functional prototype of iBatch on Ethereum. The evaluation results show that 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.

CCS CONCEPTS

• **Security and privacy** → **Security protocols**; • **Software and its engineering** → *Ultra-large-scale systems.*

KEYWORDS

Blockchains, smart contracts, DeFi, cost effectiveness, replay attacks

ACM Reference Format:
Yibo Wang, Qi Zhang, Kai Li, Yuzhe Tang, Jiaqi Chen, Xiapu Luo, and Ting Chen. 2021. iBatch: Saving Ethereum Fees via Secure and Cost-Effective Batching of Smart-Contract Invocations. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468568

*Yuzhe Tang is the corresponding author.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468568

1 INTRODUCTION

The recent paradigm shift to building decentralized applications (DApps) on blockchains has nurtured a number of fast-growing domains, such as decentralized finance (DeFi), decentralized online gaming, et al. that have the potential of disrupting conventional business in finance, gaming, et al. The core value brought by DApps is their decentralized system architecture that is amendable to tackle the mistrust crisis in many security-oriented businesses (e.g., "trusted" authorities are constantly caught misbehaving). However, despite the attractive trustless architecture and moderate popularity in practice, an impediment to DApps' broader adoption is their intensive use of underlying blockchain and the associated high costs. Ethereum [13], the second largest blockchain after Bitcoin and the most popular DApp platform, charges a high unit cost for data movement (via transactions) and for data processing (via smart-contract execution). For instance, sending one-megabyte application data to Ethereum costs 17.5 Ether or more than 25,000 USD (at the exchange rate as of Jan. 2021), which is much more expensive than alternative centralized solutions (e.g., cloud services) and has scared away customers (e.g., Binance [21]).

Towards cost-effective use of blockchains, existing research mainly tackles the problem from the angle of designing new protocols at blockchain layer one (i.e., redesigning the consensus protocol and building a new blockchain system [29, 32]) and at layer two (i.e., by offloading the workload from the blockchain to off-chain clients, such as in payment channel networks [19, 26, 28, 38]). However, these new protocols are designed without the legacy platform of an operational blockchain and deployed DApps in mind and result in unsatisfactory deployability: For instance, existing protocols either require bootstrapping a brand new blockchain network (as in the layer-one approach) or develop from scratch the on-chain and off-chain components of a DApp (i.e., to support payment networks). As a result, there is a lack of adoption of these protocols among legacy DApps at scale.

This work aims at optimizing legacy DApps' use of expensive Ethereum blockchain and achieving cost effectiveness. Towards the goal, we focus on designing a middleware system on top of an unmodified Ethereum network and DApp clients. We also develop

software tools to facilitate integrating the middleware with legacy DApps' smart contracts.

To motivate our approach, consider a typical DApp architecture where a DApp client holding an Ethereum account sends a transaction on the Ethereum blockchain to invoke a smart-contract function there. A typical DApp's smart contract runs event-driven logic, and a popular DApp would receive a "large" number of "small" invocations: 1) An individual invocation is often with a small amount of data and triggers few lines of smart-contract code; think as an example the `transfer()` function in an ERC20 token smart contract. 2) A popular DApp features an intensive stream of invocations that arrive at a high rate. This workload characteristic holds over time, as we verified on various Ethereum traces (see the IDEX trace in § 2.1 and Chainlink/Binance (BNB)/Tether traces in § 6), and it is also corroborated by external Ethereum exploration services [12, 15]. The workload with a high rate of small invocations renders the transaction fee a significant cost component that alone is worth optimization. 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 [1, 30]. For instance, under Ethereum's current block limit, one can theoretically batch up to 20 normal invocations in a transaction, leading to a potential fee reduction by $\frac{1}{20}\times$. By this promise, invocation batching has long been craved for among Ethereum developers, evidenced by a number of Ethereum Improvement Proposals (EIPs) [6, 7, 20]. Despite the strong interest, it still lacks real-world support of invocation batching in Ethereum, as these EIPs are not made into production after years of discussion. We believe this unsatisfactory status is due to the design challenges raised by the tradeoff among batching's security, cost-effectiveness and timeliness (short delay), as presented next.

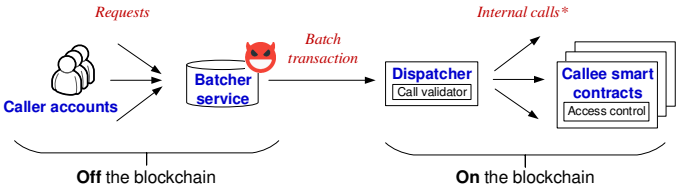


Figure 1: Batching smart-contract invocations in Ethereum: Note that the `Dispatcher` can be a standalone smart contract or be a function inlining the callee function; in the latter case, the "internal call*" is straight-line code execution.

Challenges: First, Ethereum does not have native support of batching in the sense that an Ethereum transaction transfers Ether from one account to another account. This is different from Bitcoin and other blockchains whose transaction can encode multiple coin transfers. This difference renders the existing architectures to batch transfers in non-Ethereum blockchains [1, 24, 31, 37, 39] inapplicable to batch invocations on Ethereum. To address the challenge, we introduce two intermediaries between a caller account and a callee smart contract. As depicted in Figure 1, they are a relay service off-chain, called `Batcher`, and an on-chain component, called `Dispatcher`. The `Batcher`'s job is to batch multiple invocation requests sent from the caller accounts and send them in a single transaction to the `Dispatcher`, which further unmarshalls the original invocations and relays them individual to the intended callee smart contracts.

Second, the off-chain `Batcher` service need not be trusted by the callers (who, in a decentralized world, are reluctant to trust any third-parties beyond the blockchain). Defending against the untrusted `Batcher` incurs overhead that may offset the cost saving from batching and instead result in net cost increases. Specifically, in our threat model, the adversarial `Batcher` is financially incentivized to mount attacks and to modify, forge, replay or omit the invocation requests in the batch transaction; for instance, replaying a `transfer()` of an ERC20 token can benefit the receiver of the transfer. To defend against the threat, a baseline design is to run the entire transaction validation logic in the trusted `Dispatcher` smart contract on-chain, which bloats the contract program and incurs overhead (e.g., to maintain additional program states). Our evaluation study in Technical Report [40] shows this baseline denoted by B2 increases the net cost per invocation rather than decreasing it. For secure and cost-effective batching, we propose a security protocol that allows off-chain DApp callers in the same batch to jointly sign the batch transaction so that the additional program states (e.g., the per-account nonces as a defense to replaying attacks) can be offloaded offline and the `Dispatcher` smart contract can be *stateless*, rendering low overhead and positive net cost savings.

Third, batching requires waiting for enough invocations and can introduce delay to when the batched invocations are included in the blockchain. For the many DApps sensitive to invocation timing (e.g., real-time trading, auctions and other DeFi applications), such delay is undesirable. To attain delay-free batching, we propose to use the transaction price to the rescue. Briefly, Ethereum blockchain admits a limited number of transaction per block and prioritizes the processing of incoming transactions with a higher "price" (i.e., the so-call Gas price which is the amount of Ether per computation unit paid to miners). Thus, our idea is to generate a batch transaction with a higher price so that it can be included in the blockchain more quickly, and this saved time can offset the waiting time caused by batching, resulting in an overall zero delay in blockchain inclusion. We propose an online mechanism to conservatively batch invocations originally in one block and carefully set Gas price of batch transactions with several heuristics to counter the limited knowledge in online batching.

Systems solutions: Overall, this work systematically addresses the challenges above and presents a comprehensive framework, named iBatch, that incorporates the proposed techniques under one roof. iBatch includes the middleware system of `Dispatcher` and `Batcher` and a series of policies that configure the system to adapt the batching to specific DApps' workloads. Concretely, the middleware system exposes knobs to tune the batching in timing (how long to wait for invocations to be batched), target invocations (what invocations to batch) and other conditions. Through this, policies that range from conservative to aggressive batching are proposed, so that the system can be tailored to the different needs of DApps. For instance, the DApps sensitive to invocation timing can be best supported by the conservative batching policy with minimal delay. Other DApps more tolerable with delays can be supported by more aggressive batching policies, which result in a higher degree of cost saving. We demonstrate the feasibility of iBatch's middleware design by building a functional prototype with Ethereum's Geth

client [16]. Particularly, we statically instrument Geth to hook the `Batcher`'s code.

We further address the integration of iBatch with legacy DApps and the operational Ethereum network by automatically rewriting their smart contracts. Briefly, with batching, the internal calls are sent from `Dispatcher` (instead of the original caller account), which makes them unauthorized access to the original callee, leading to failed invocations. In iBatch, we propose techniques to rewrite callee smart contracts, particularly their access-control structure to white-list `Dispatcher`. We acknowledge the recent Ethereum development EIP-3074 [8], which, if made into an operational Ethereum network, will facilitate iBatch's integration without rewriting smart contracts (discussed in § 5 and full details in Technical Report [40]).

Systematic evaluation: We systematically evaluate the invocation cost and delay in iBatch, under both real and synthetic workloads. First, we build a fast transaction-replay engine that executes transactions at a much higher speed than the transactions are originally included in the blockchain. This allows us to conduct large-scale measurements, say replaying a trace of transactions that last for months in real life within hours in the experiments. Second, we collect the trace of transactions/calls under four representative DApps, that is, IDEX [33] representing decentralized exchanges (DEX), BNB [4] and Tether [22] for tokens, and Chainlink [3] for data feeding. From there, we build a benchmark of traces that can be replayed in our platform. Third, we conduct extensive evaluations based on the developed platform (i.e., replay engine, benchmarks, and prototype we built). The target performance metrics are the system's costs (in terms of Ether and Gas) and delays between invocation submission time and block inclusion time.

The result under the BNB-token/IDEX/Chainlink trace shows that iBatch configured with a time window of 120 seconds to batch all invocations can save around 50%/24%/17.6% of the Gas per invocation of the unbatched baseline. For delay-sensitive DApps, as evaluated under the workloads of Tether tokens, iBatch saves 19.06% (31.52%) cost at the expense of a delay of 0.26 (1.66) blocks.

Contributions: This work makes the following contributions:

• *Security protocol*: We design a lightweight security protocol for batching of smart contract invocations in Ethereum without trusting third-party servers (i.e., the `Batcher`). The security protocol defends against a variety of invocation manipulations. New techniques are proposed to jointly sign invocations off-chain and validate invocations on-chain without states against replay attacks.

• *Cost-effective systems*: We design a middleware system implementing the above protocol and propose batching policies from conservative to aggressive batching. Particularly, we propose an online mechanism to optimize the cost without delaying invocation execution. We further address the integration with the current Ethereum client by automatically rewrite smart contracts.

• *Systematic evaluation*: We built an evaluation platform for fast and cost-accurate transaction replaying and constructed transaction benchmarks on popular Ethereum applications. With a functional prototype of iBatch, 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.

Overall, this work tackles the design tradeoff among security, cost-effectiveness and delays in batching invocations. While the implementation and evaluation in this work are on Ethereum, we believe the design tradeoffs and principles are directly applicable to Ethereum forks (e.g., Binance smart chain [2]) and generalizable to other smart-contract platforms [10].

Roadmap: Section § 2 formulates the research. § 3 presents the iBatch's security protocol. iBatch's batching policies are described in § 4. § 5 presents the smart-contract rewriters to facilitate iBatch's integration with legacy smart contracts. § 6 shows the evaluation results in cost and invocation delay. Related works are described in § 7 and conclusion in § 8.

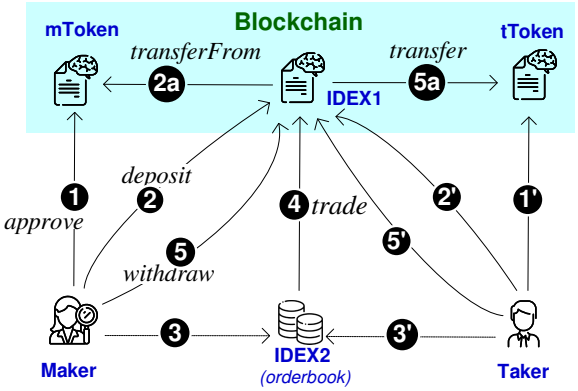
2 RESEARCH FORMULATION

2.1 Motivating Example

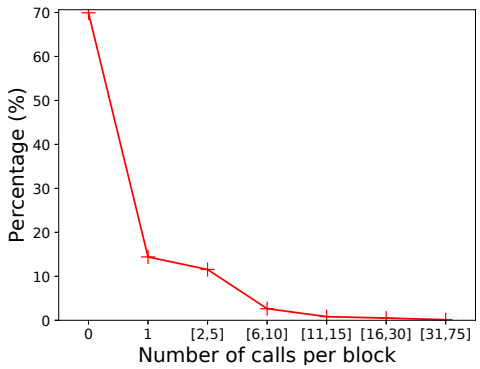
We use a real-world scenario, namely IDEX [5, 33], to motivate our work. IDEX is a decentralized exchange protocol that allows owners of different ERC20 tokens to exchange their tokens at the preferred price/volume. Consider that account Alice sells her tokens `mToken` to another account Bob in return of his tokens `tToken`. To do so, Alice makes an order to be taken by Bob, and Alice (Bob) is called a `maker` (a `taker`). The IDEX protocol is executed among six Ethereum accounts include a `maker` account, a `taker` account, maker's token contract `mToken`, taker's token contract `tToken`, the core IDEX smart contract `IDEX1` [18], and `IDEX1`'s off-chain owner `IDEX2` [9]. The protocol execution is depicted and described in Figure 2a.

In particular, there are four types of transactions in IDEX that invoke smart contracts, that is, maker's (taker's) call to `approve` her (his) token contract (i.e., ❶ and ❶'), maker's (taker's) call to `deposit` to `IDEX1` (i.e., ❷ and ❷'), IDEX2's call to `trade` on `IDEX1` (i.e., ❹), and maker's (taker's) call to `withdraw` (i.e., ❺). Among the transaction-triggered external calls, `trade` is most intensively invoked. As we examine the Ethereum history via Ethereum-ETL service on Google BigQuery [11], 61.59% of the invocations received by the `IDEX1` contract from its launch on Sep. 27, 2017 to Feb. 23, 2019 are on `trade()`.[1] More importantly, *the* `trade` *invocations are so intensively issued that many of them wind up in the same Ethereum block.* We measured the number of `trade` calls in the same Ethereum block, on the call trace above. Figure 2b plots the cumulative distribution of Ethereum blocks by the per-block call number. For instance, about 30% of Ethereum blocks have more than one `trade` calls in them, 5% of blocks have more than four `trade` calls, and 0.36% of blocks have 20 `trade` calls. If one batches the 20 `trade` invocations of these Ethereum blocks into a single transaction, the transaction fee can be reduced to $\frac{1}{20}$, although it may incur additional costs for smart-contract execution. In general,

[1] We did not take the `IDEX` transactions after Feb. 2019 when IDEX's traffic started to decline and was then shadowed by other more popular DEX, such as Uniswap [23]. Here, we stress that although our IDEX's trace ends in Feb. 2019 (as of this writing in May 2021), Ethereum's transaction rate steadily increases over time. Particularly, recent years see drastic rate growth as Ether price soars since early 2020. This is verified by the more recent traces we collected in 2020, such as Chainlink and Tether tokens, as in the cost evaluation in § 6, and also corroborates external Ethereum exploration websites [12, 15].



(a) The IDEX system model and protocol: The example scenario shows running IDEX among six Ethereum accounts: Three user accounts (`maker`, `taker` and `IDEX2`) and three contracts (`mToken`, `tToken` and `IDEX1`).



(b) Distribution of `trade` calls over Ethereum blocks; in this figure, $X = 5$, $Y = 11.4$ means in 11.4% of Ethereum blocks, the number of `trade` calls per block is between 2 and 5.

Figure 2: IDEX protocol and call distribution over blocks. The protocol execution in Figure 2a involves five steps: 1) The `maker` deposits her tokens to `IDEX1`, which invokes three functions: ❶ `maker.approve()`, ❷ `maker.deposit()` and ❷ₐ `IDEX1.transferFrom(maker,DEX1)`. 2) The taker similarly deposits his tokens by issuing ❶' `taker.approve()`, ❷' `taker.deposit()` and `IDEX1.transferFrom(taker,DEX1)` (not shown in the figure). 3) The maker and taker sends their respective selling and buying orders (❸ and ❸') to the off-chain `IDEX2`, who match-make orders in an order-book. 4) The owner `IDEX2` calls contract `IDEX1`'s function `trade(taker,maker)` (❹) to execute the trade on-chain. 5) The maker issues `withdraw` (❺) which further sends `transfer()` calls (❺ₐ) to `tToken` contract. Similarly, the taker can submit calls to withdraw her tokens (❺'). The icons used in the figure are by www.freepik.com.

for blocks with X `trade` calls, one can batch the calls into one transaction, leading to an X-fold fee reduction. By plugging into X the measurement results in Figure 2b, we can expect the overall fee-saving in the case IDEX to be 10.7%. This is the saving from `trade` calls only. Note that because the original `trade` calls are in the same block, batching them in a single transaction does not introduce additional delay/inconsistency.

Generally, there are four types of batching strategies: **Type S1)** Batch invocations of the same caller and same callee, such as all `trade` calls from the same caller (`IDEX2`) and sent to the same callee smart contract (`IDEX1`), **S2)** batch invocations of different callers and the same callee, such as all the `deposit` calls, **S3)** batch invocations of the same caller and different callees, and **S4)** batch invocations of different callers and different callees, such as the `approve` calls in IDEX. We mainly consider the general case of S4 in the paper and tailor the system to other invocation types in § 4.

2.2 Threat Model

Recall the system model in Figure 1 that introduces the `Batcher` and `Dispatcher`, as two intermediaries between caller accounts and callee smart contracts. For generalizability, our threat model considers an untrusted third-party `Batcher`. For instance, in the case of IDEX, the `Batcher` can batch `approve`, `deposit` and `trade`, and does not require the trust from their callers. The third-party `Batcher` can mount attacks to forge, replay, modify and even omit the invocations from the callers. Our model assumes unmodified the trust relationship among callers; for instance, if there is a counterparty risk between a maker account and a taker account in the vanilla IDEX, the same trust remains in iBatch.

The smart contracts, including both `Dispatcher` and application contracts, are trusted in terms of program security (no exploitable security bugs), execution unstoppability, etc. We also make a standard assumption on blockchain security that the blockchain is immutable, fork-consistent, and Sybil-secure. The underlying security assumption is that a deployed blockchain system runs among a large number of peers with an honest majority, and compromising the majority of peers is hard. This work is built on Ethereum's smart contracts, cost model, and transaction model. It treats Ethereum's consensus and underlying P2P networks as a blackbox.

2.3 Design Goals & Baselines

The design goal of iBatch is this: Through batching invocations, there should be a significant portion of the transaction cost saved (1. cost saving) for calling generic smart contracts (2. generalizability), while staying secure against the newly introduced adversary of off-chain `Batcher` (3. security). Specifically, the cost-saving goal is to reduce a significant portion of the Gas cost per invocation, via batching calls under the constraint of maximal transaction size. The generalizability goal is that the system should work with the general case of Batch Type S4. The security goal is to detect and prevent attacks mounted by the untrusted `Batcher` and protect the integrity of invocation information.

There is limited research on batching smart-contract invocations. In Table 1, we compare iBatch's research goal with the two baseline designs (covering the existing research, e.g., Airdrop batching [30]), which we will describe next.

Table 1: iBatch's design choices and related works

	Generalizable	Cost Saving
Baseline B1	✗	✗
Baseline B2	✓	✗
iBatch	✓	✓

Baseline B1: This baseline design of batching considers a special case. Suppose account A is about to transfer tokens to N other accounts $B_1, B_2, \ldots, B_N$. Instead of sending N transactions, account A can set up a smart contract C and send one transaction to C that sends the N transfers (e.g., by calling solidity's `transfer()` function N times) in one shot. This is essentially the batching scheme used in existing works [30] for airdropping tokens (a common practice to give away free tokens [17]). While this scheme handles the case of a single sender A, it can be naturally extended to support multiple senders $A_1, A2 \ldots$. In this case, multiple senders calls `approve` to delegate their account balance to a smart contract C before C can batch-transfer tokens to multiple receivers.

Overall, this batching scheme is limited as it depends on ERC20 functions (`approve`/`transfer`). Also, it does not necessarily lead to cost saving, as each transfer still incurs at least one transaction (i.e., `approve`).

3 THE IBATCH SECURITY PROTOCOL

This section presents the design rationale, protocol description, its security analysis (sketch), and the resultant system design. We defer the full protocol analysis to § A.

3.1 Design Space: Security-Cost Tradeoff

Batching framework: We start by describing the design framework to support batching of invocations to generic smart contracts. In this framework, the `Batcher` batches a number of invocation requests and sends them in a batch translation to the `Dispatcher` smart contract. The `Dispatcher` extracts the invocations and relays them to the callee smart contracts.

In our threat model, the `Batcher` mount invocation-manipulation attacks. To prevent a forged invocation, the `Dispatcher` verifies the signatures of the original callers.

Baseline B2: To prevent replaying an invocation, a baseline design (B2) is to elevate a blockchain's native replay protection into the smart-contract level. Specifically, existing blockchain systems defend against transaction replaying attacks by maintaining certain states on blockchain and check any incoming transaction against such states to detect replay. For instance, Ethereum maintains a monotonic counter per account, called nonce, and checks if the nonce in any incoming transaction increments the nonce state on-chain; a false condition implies replayed transaction. Bitcoin maintains the states of UTXO to detect replayed transactions.

In B2, we implement per-account nonces in the `Dispatcher` smart contract and use them to check against incoming invocations, in order to detect replayed invocations.

A cost observation: In our preliminary cost evaluation on Ethereum, we found *a sweet spot that the batching framework without replay protection can lead to positive cost saving, while adding the baseline design (B2) of replay protection end up with a negative cost saving*. That is, the overhead of maintaining nonces in smart contracts in B2 offsets the cost saving by batching invocations.

Thus, in iBatch, we avoid placing nonces in `Dispatcher` and focus on an off-chain defense against invocation replaying. With an untrusted `Batcher`, we assume every caller is online for an extended period that covers the batch time window its invocation is submitted. We propose an off-chain protocol in which callers interactively sign a batch transaction. Note that there is an alternative design that callers audit batch transactions after they are acknowledged from the blockchain; however, the audit scheme does not prevent (only detects) a replayed invocation.

3.2 Protocol Description

The protocol supports the general-case batching, that is, batching Type S4 invocations. Suppose in a batch time window, there are N invocations submitted from different callers. The iBatch protocol follows the batching framework described above and it works in the following four steps:

1) In the batch time window, a caller submits the i-th invocation request, denoted by $\texttt{call}_i$, to the `Batcher` service. As in Equation 1, the request $\texttt{call}_i$ contains the caller's address/public key $\texttt{account}_i$, callee smart contract address $\texttt{cntr}_i$, function name $\texttt{func}_i$, and argument list $\texttt{args}_i$. With $i \in [1, N]$, there are N such invocations in the time window.

2) 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. As shown in Equation 2, message `bmsg` is a concatenation of the N requests, $\texttt{call}_i$'s, their caller nonces $\texttt{nonce}_i$'s, and `Batcher` *account's nonce,* $\texttt{nonce}_\texttt{B}$. Then, the `Batcher` broadcasts the batch message `bmsg` in parallel to all N callers of this batch. Each of the callers checks if there is one and only one copy of its invocation $\texttt{call}_1$ in the batch message; specifically, this is done by checking equality between $\texttt{nonce}_1$ in the batch message and the nonce maintained locally by the caller. After a successful check of equality, the caller signs the message `bmsg_sign`, that is, `bmsg` without callers' nonces as shown in Equation 3. The caller signs `bmsg_sign` using the private key in her Ethereum account. She then sends her signature to the `Batcher`. This step finishes until all N callers have signed the message and return their signatures to the `Batcher`.

3) `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.This is presented in Equation 4 where `CA` is the address of smart contract `Dispatcher`.

$$\forall i, \texttt{call}_i = \langle \texttt{account}_i, \texttt{cntr}_i, \texttt{func}_i, \texttt{args}_i \rangle \quad (1)$$

$$\texttt{bmsg} = \texttt{call}_1 \| \texttt{nonce}_1 \| \texttt{call}_2 \| \texttt{nonce}_2 \| \ldots \| \texttt{call}_N \| \texttt{nonce}_N \| \texttt{nonce}_\texttt{B} \quad (2)$$

$$\texttt{bmsg\_sign} = \texttt{call}_1 \| \texttt{call}_2 \| \ldots \| \texttt{call}_N \| \texttt{nonce}_\texttt{B} \quad (3)$$

$$\forall i, \texttt{sig}_i = sign_{\texttt{account}_i}(\texttt{bmsg\_sign})$$

$$\texttt{bsig} = \texttt{sig}_1 \| \texttt{sig}_2 \| \ldots \| \texttt{sig}_N$$

$$\texttt{data} = \langle \texttt{dispatch\_func}, \texttt{bmsg\_sign} \| \texttt{bsig} \rangle$$

$$\texttt{tx} = \langle \texttt{account}_\texttt{B}, \texttt{nonce}_\texttt{B}, \texttt{CA}_\texttt{D}, \texttt{sig}_\texttt{B}, \texttt{value}, \texttt{data} \rangle \quad (4)$$

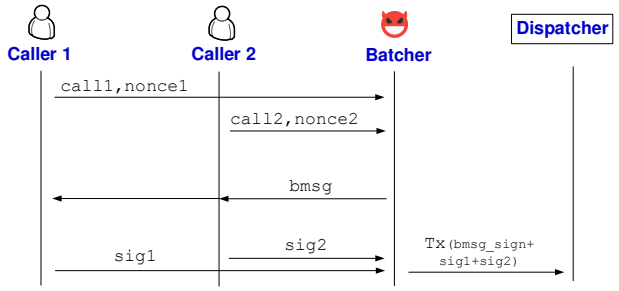


Figure 3: Generation of batch transaction off-chain among `Batcher` and two caller accounts

On the blockchain, 4) in function `dispatch_func`, smart contract `Dispatcher` parses the transaction and extract the original invocations $\texttt{call}_i$ before forwarding them to callees, namely $\texttt{cntr}_i$ and $\texttt{func}_i$. Particularly, 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 $\texttt{ecrecover}(\texttt{call}_i, \texttt{sig}_i, \texttt{account}_i)$. If successful, the `Dispatcher` then internal-calls the callee smart contract. At last, the callee function executes the body of the function under the given arguments $\texttt{args}_i$.

An example: Figure 3 shows an example of the interactive signing process, which involves $N = 2$ callers respectively sending two invocations. The process is executed in five messages among the two callers and the `Batcher` off-chain.

3.3 Security Analysis (Sketch)

The iBatch protocol achieves invocation integrity against malicious `Batcher`. Specifically, 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. iBatch assumes the availability of `Batcher` which is reasonable on today's highly available platforms (e.g., clouds). Briefly, the security proof is due to the following intuition: The hardness of forging/replaying invocations and the hardness of omitting invocations without detection in iBatch can be reduced to the unforgeability of digital signatures and the hardness of double-spending attacks in the underlying blockchain. The full protocol security analysis is in Appendix A.

3.4 System Overview

To materialize the protocol, we design a middleware system atop the underlying Ethereum-DApp ecosystem. Specifically, the system runs the `Batcher` middleware on an Ethereum node (e.g., a Geth client) that is synchronized with an Ethereum network. The `Dispatcher` smart contract runs on the Ethereum network and forwards invocations to the callee smart contracts.

The off-chain `Batcher` is a middleware running on an untrusted third-party host. In general, the `Batcher` buffers incoming invocations submitted by callers and under certain conditions (as described below) triggers the batching of invocations. Once a batch of invocations is determined, the `Batcher` jointly works with original callers to generate the batch transaction (as described by the joint-signing process in § 3.2).

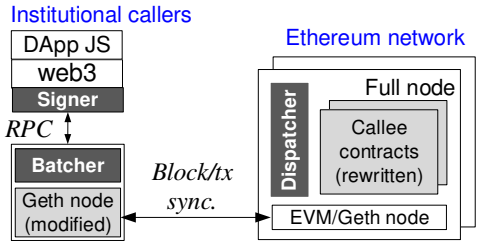


Figure 4: Retrofitting iBatch to Ethereum-based DApps: The right-hand side of this figure illustrates the general mechanism where the two dark shades are the core system components of iBatch, and the light shade is a statically instrumented Ethereum full node (running `Geth`).

Implementation: To transparently support unmodified DApp clients, we statically instrument Geth's handling of raw transactions and expose hooks to call back the `Batcher`'s code that makes decisions on batching, as will be described next. Specifically, the instrumented Geth node unmarshalls a raw transaction received, extracts its arguments, places it in `Batcher`' internal buffer (e.g., `bpool` as will be described) and makes essential decisions regarding which invocations to be included in the next batch transaction before actually generating and sending it (as described above). The statically instrumented Geth node retains the same `sendRawTransaction()`/`sendTransaction()` API and thus supports unmodified DApp clients.

Next in § 4, we propose policies for `Batcher`'s decision-making that strikes a balance between costs and delay. To integrate iBatch with legacy smart contracts, we propose schemes to automatically rewrite smart contracts at scale, which is described in § 5.

4 BATCHER'S POLICIES

In this section, we propose mechanisms and policies for the `Batcher` to properly batch invocations for design goals in cost and delay. We first formulate the design goal of optimizing Gas cost per invocation in the presence of the workload. We then formulate the design goal of reducing Ether cost per invocation without causing delay to when the invocation is executed on Ethereum.

4.1 Optimizing Gas Cost

The degree of amortizing the cost by iBatch is dependent on the number and type of invocations put in a batch. In this subsection, we propose a series of policies that the `Batcher` can use in practice. The motivating observation is that there is no single policy that fits all (workloads), and under different workloads, the most cost-effective policy may differ.

Note that the cost unit we consider here is Gas per invocation (which measures the amount of computational load an Ethereum node needs to carry out to serve an invocation). The proposed policies may cause invocation delay, and the policies are suitable for DApps that are insensitive to such delay.

- **Policy Wsec**: *Batching all invocations that arrive in a time window, say W seconds.* In practice, the larger W it is, the more invocations will end up in a batch and hence the lower Gas each invocation is amortized. However, a larger W value means the `Batcher` needs to wait longer, potentially causing

inconsistency and delay of invocation execution. We will systematically study the cost-delay tradeoff when taking into account the factor of Gas price in § 4.2.

- **Policy Top1**: *Batching only the invocations that are sent from one account, such as the most intensive sender.* The motivation for doing this is that if all invocations needed batching are from one sender account, the batch transaction (of multiple invocations) only needs to be verified for once, thus eliminating the needs of verifying signatures in smart contracts and lowering the overhead.
  In practice, Top1 can be toggled on top of a Wsec policy. For instance, Xsecond-Top1 means batching only the invocations that arrive in a W-second window and are from the most intensive sender in that window.
  Whether the presence of Top1 batching policy can actually lead to positive Gas saving is dependent on workloads. If there is an institutional account sending invocations much more intensive than others, applying Top1 can lead to sufficient invocations in a batch and positive Gas saving. Otherwise, if the workload does not contain enough such invocations, the batch may be smaller than the one without Top1, which limits the degree of cost amortization.
- **Policy Min**X: *Only batch when there are more than X candidate invocations in a batch time window.* The intuition here is that if there are too few invocations, the degree of cost amortization may be too low and can be offset by the batching overhead to result in negative cost saving. In the Technical Report [40], we conduct cost analysis based on Ethereum's Gas cost profile on different transaction operations and derive the minimal value of X should be 5. That is, a batch is only beneficial when it has at least 5 invocations.

4.2 Optimizing Ether Cost with Minimal Delay

In this subsection, we consider a class of DApps, notably DeFi applications, that are sensitive to invocation timing. In these DApps, manipulating invocation timing or introducing invocation delay may cause consequences ranging from DApp service unresponsiveness to security damage (e.g., under the front-running attacks). Thus, we formulate the design goal to be optimizing Ether cost per invocation without introducing any invocation delay. We call the no-delay policy described in this subsection by 1block. Note that in Ethereum, the Ether cost of a transaction is the product of the transaction's Gas and its Gas price.

Assume an oracle who can predict what invocations are included in a block (without batching) at the time when the invocations are submitted. An ideal, optimal offline algorithm is to batch the invocations in a future block and generate a batch transaction. If the Gas price of the batch transaction is set to be higher than at least one transaction in that future block, it is bound the batch transaction can be included in the same block with the unbatched case. In other words, no block delay is introduced. We call this approach by offline optimal batching as an ideal scheme.

In practice, the `Batcher` at the invocation submission time may not accurately predict when a block will be found and which block will include the invocation. We propose a realistic, online batching mechanism to reduce or eliminate the block delay.

Online batching w. minimal delay (1block): We propose a system design of `Batcher` atop an Ethereum client extending its memory pool (or `txpool`) functionality. We call this design by 1block. We first describe the proposed system design and then decision-making heuristics. In a vanilla Ethereum client, a transaction is first buffered in memory (in a data structure called `txpool`), is then selected (by comparing its Gas price against other transactions in the `txpool`) by miners, and is included in the next block.

In iBatch, the Ethereum client running on `Batcher` is extended with an additional memory buffer that we call `bpool` and that stores submitted invocations prior to the batch transaction.

The `Batcher` service continuously receives the submitted invocations of registered DApps and buffers them into the `bpool`. To manage and evict invocations, the service periodically runs the following process: Every time it receives a block, the service waits for d seconds and then executes Procedure `bpoolEvict` which produces a batch transaction to send to the Ethereum network. More specifically, the `bpoolEvict` procedure reads as input the transactions residing in the `txpool` and the invocations residing in the `bpool`. The procedure produces a batch transaction encoding selected invocations to be sent to the Ethereum network. There are two essential decisions to make by Procedure `bpoolEvict`: C1) What invocations to be evicted from `bpool` and to be put in the batch transaction. It also needs to decide C2) What Gas-price value should be set on the batch transaction.

In addition to C1 and C2, the batching mechanism can be configured by d, that is, how long it waits after a received block to run Procedure `bpoolEvict`. In the following, we describe a series of policies to configure C1), C2) and d of the `Batcher`.

Example: We show an example process illustrated in Figure 5: It shows the timeline in which `bpool` on the `Batcher` operates and interacts with the remote Ethereum network. At the beginning (0-th second), the `Batcher` receives a block B_0 of 2 transactions, which evicts the 2 transactions from `txpool` and leaves it of 10 transactions. Also assume there are 10 invocations in the `bpool` in the beginning. On the d = 10-th seconds, the service runs `bpoolEvict` which results in a batch transaction of 3 invocations. It sends the batch transaction to the Ethereum network. As the Gas price of the batch transaction is high, it will be selected by the miners in the remote Ethereum network upon the next block B_1 being propagated, say on the 13-th second. If the next-next block B_2 is found on the 20-th second, the batch transaction will be included in B_2.

Heuristics: For C1), we propose to select the invocations in the `bpool` that have higher Gas price than h such that the total Gas of transactions (and invocations) whose prices are higher than h is under the block limit. Moreover, the total Gas of transactions whose prices are higher than $h-1$ is above the block limit.

For C2), a baseline is to set a fixed Gas price for every batch transaction, which does not reflect the price distribution in the current batch/block and can lead to excessive cost. We propose "dynamic" Gas pricing policies where the price of a batch transaction is dynamically set to ensure low Ether cost yet without delaying the block it will be included. We propose two policies:

- **Policy Batch-**$X\%$: The Gas price of a batch transaction is set to be above $X\%$ of the invocations in the batch.

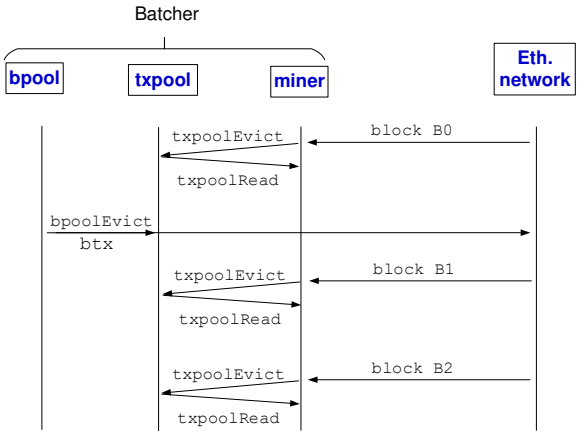


Figure 5: An example process of running `bpool` and its eviction on `Batcher`.

- **Policy Block-X%**: The Gas price of a batch transaction is set to be above X% of the transactions in the block (also including the invocations in the batch).

For instance, suppose there are 7 regular transactions included in a block and a batch transaction which consists of 3 invocations. The three invocations are associated with prices 8, 9 and 10, and the 7 regular transactions' Gas prices are $1, 2 \ldots 7$. With policy batch-50%, the batch transaction's price is 9. With policy block-10%, the batch transaction's price is 1.

5 INTEGRATING IBATCH WITH LEGACY SMART CONTRACTS

```
1  //original functions
2  contract Token {
3  ...
4  modifier noBlacklisted {
5    assert(!isBlackListed[msg.sender]);_;}
6  function transfer(address to, unit256 value) noBlacklisted {
7    super.transfer(to,value);
8    balances[msg.sender] = SafeMath.safeSub(balances[msg.sender],
         value);
9    balances[to] = SafeMath.safeAdd(balances[_to], value);}
10 //rewritten functions by iBatch
11 modifier noBlacklistedByD(address from) {
12   assert(!isBlackListed[from]);}
13 function transferByD(address from,address to,unit256 value)
14     noBlacklistedByD(from){
15   assert(msg.sender!=dispatcher);
16   super.transferByD(from,to,value);
17   balances[from]=SafeMath.safeSub(balances[from],value);
18   balances[to]=SafeMath.safeAdd(balances[to],value);
19 }}
```

Listing 1: Rewriting application smart contract: The example of rewriting an ERC20 token.

Running iBatch with unmodified smart contracts on today's Ethereum clients (as of this writing in May 2021) would fail because the unmodified smart contracts do not authorize the unmarshalled invocations sent from `Dispatcher` account.

Thus, iBatch's integration with a legacy Ethereum platform entails either rewriting DApps' smart contracts (e.g., to whitelist `Dispatcher`) or patching the Ethereum Virtual Machine (EVM) inside an Ethereum client. For the latter, we notice a recent Ethereum development EIP-3074 [8], which would allow `Dispatcher` to send an invocation with `msg.sender` remaining the original caller's account. This EIP, currently in progress, if made into the future EVM, would make it possible to directly integrate iBatch with an operation Ethereum network without rewriting smart contracts. We discuss details in Technical Report [40].

In this section, we focus on the current EVM platform and propose smart-contract rewriters. We describe a source-code rewriter in the next paragraph, while leaving the proposed bytecode rewriter to Technical Report [40].

Source-code rewriter: We rewrite the solidity code of a smart contract to whitelist the `Dispatcher` account, as follows. Given an application smart contract `bar`, we create a new contract say `barByD` to inherent contract `bar`. We rewrite each function that contains references to `msg.sender`: Given such a function `foo(type original_args)` in contract `bar`, we add in contract `barByD` a new function `fooByD(address from, type original_args)`. 1) In this new function, a new argument `from` is added in function `fooByD`. The function body in `fooByD()` is the same with `foo()`, except for three modifications: 2) References `msg.sender` in `foo()` are replaced by argument `from` in `fooByD()`. 3) The first code line in `fooByD()` asserts if the function caller is `Dispatcher`. 4) For any functions of `bar` that are called inside `foo`, the function invocation is rewritten to add a new argument `from`. In particular, this includes the case of modifier functions in solidity. Listing 1 illustrates the example of rewriting `transfer()` in an ERC20 token contract.

6 EVALUATION

This section presents the evaluation of iBatch. We report iBatch's performance (cost and delay) in comparison with the unbatched baseline under real workloads. We formulate two research questions (RQ1 and RQ2) that are respectively answered by our experiments in § 6.1 and § 6.2. We defer to Technical Report [40] other experiments that answer research questions comparing iBatch with batched baselines.

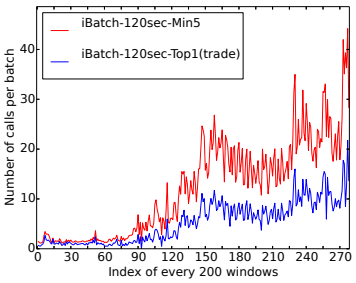
6.1 Evaluating Gas Cost

RQ1: How much Gas per invocation does iBatch result in, under different policies and in comparison with the unbatched baseline (B0), under real workloads?

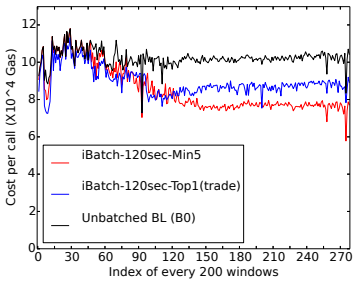
Motivation: Gas per invocation is the metric directly affected by iBatch. This metric shows certain aspects of iBatch's cost-effectiveness. iBatch's Gas per invocation is sensitive to different policies (described in § 4). It is also dependent on the actual workload (e.g., how frequent invocations are sent in a fixed period). We set up this RQ to explore the sensitivity to policies and real workloads.

Experiment methodology: First, we choose three representative and popular DApps, that is, IDEX (representing decentralized exchange), BNB token (representing ERC20 tokens), and Chainlink (representing data feeds). We collect the DApps' invocations by running an instrumented Geth node to join the Ethereum mainnet. During the (basic) node synchronization, the node is instrumented to intercept all the transactions (i.e., external calls) and internal calls and dump them onto a local log file.

Then, we prepare the collected trace to be replayable with accurate Gas cost. To do so, we replace the Ethereum addresses (i.e.,



(a) batch size



(b) gas cost per call

Figure 6: IDEX trace (5 months): 3 functions batch result

Traces	Policies	Gas per call (10k)
IDEX	iBatch-120sec-min5	7.78 (−23.68%)
	iBatch-120sec-top1	8.71 (−14.59%)
	Unbatched BL (B0)	10.20
BNB	iBatch-120sec-min5	2.14 (−59.13%)
	iBatch-120sec-top1	3.79 (−27.77%)
	Unbatched BL (B0)	5.25
Chainlink	iBatch-120sec-min5	9.53 (−17.62%)
	Unbatched BL (B0)	11.57

Figure 7: Average Gas cost per invocation

public keys of account holders) by new public keys that we generated. This allows us to know the secret keys of the addresses used in the trace and use them to unlock the accounts (and sign transactions) during the replay. In addition, for cost-accurate replaying, we collect the pre- and post-states of relevant smart contracts of the DApps (e.g., BNB token balances) on Ethereum by crawling the website https://oko.palkeo.com.

In the experiments, we first unlock all senders' accounts, then replay the invocations with mining turned off, and at last turn on the miner to obtain the transaction receipt and Gas cost. This procedure does not require us to wait for transaction receipts individually, and can greatly speed up the whole transaction-replaying process, especially in large-scale experiments. In this experiment, transactions/invocations in the original trace are replayed based on the block time, namely the block in which the transactions are originally included in real life. In the trace, only external calls are replayed and internal calls are used to cross-check the correctness of the replaying.

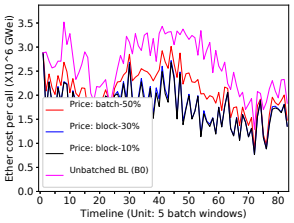
Experiment settings: We choose an IDEX trace that contains 664, 863 transactions calling three IDEX's functions: `deposit`, `trade` and `withdraw`. The trace represents Ethereum transactions submitted from Sep. 2017 to Feb. 2018 (5-month long). In the experiment, we replay the trace on our experiment platform, with and without iBatch. When running iBatch, we adopt two batching policies: 1) Batch all invocations in each 120-second window if there are more than $n_{min} = 5$ invocations in that window. The policy is denoted by 120sec-min5. 2) Batch all `trade` invocations in each 120-second window. The policy is denoted by 120sec-top1. Recall that given a time window, the top1 policy means batching only the invocations from the most popular caller in that window, which in this case is the IDEX2 or the caller of `trade`. Additionally, we set a maximal batch size to be 60 invocations, so that the Gas of batched transaction does not exceed the block Gas limit. In each experiment, we collect the resultant batch sizes and Gas cost of batched and unbatched transactions, from which we further calculate the Gas cost per call.

Results: Figure 6a shows the batch-size distribution over time. Each tick on the X axis is a time period of 200 windows (i.e., $200 \cdot 120$ seconds=400 minutes), and the Y value is the average size of the batches generated during that 200-window period. In the beginning, the generated batches are small, largely due to the fact that the distribution of calls is sparse. After the X index grows over 90, calls are more densely distributed and it generates larger batches. Comparing the two batching policies, the min5 policy generates batches that are 125% larger than those generated by the top1 policy. This can be explained by that min5 policy considers all three functions in a batch and top1 considers only `trade` function, thus the former generates larger batches.

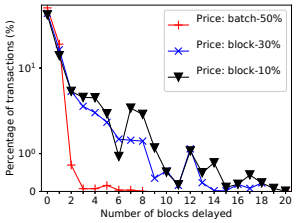
Figure 6b illustrates the average Gas per call over time. In the beginning, the two iBatch and the unbatched baseline B0 result in similar per-call costs, because of sparse call distribution over time and no chance of generating batches. After the X index grows over 90, it becomes clear that the iBatch under min5 results in the lowest Gas per call, which is 23.68% smaller than that of unbatched baseline (B0). The iBatch under top1 results in a Gas per call that is 14.59% lower than that of B0.

From these two figures, we summarize the average Gas per call in the first three rows of the table in Figure 7. We conducted similar experiments under the other DApps' trace and show the iBatch's performance in the rest of the table. Specifically, the BNB trace is from July 7, 2017 for 8 months, and the Chainlink trace is from Oct. 1, 2020 to Dec. 27, 2020. It can be seen that at the batch time window of 120 seconds, iBatch can generally save 14.59 ∼ 59.13% Gas cost per call compared with the unbatched baseline (B0).

6.2 Evaluating Ether Cost & Delay



(a) Ether cost over time (each point is a 75-second period).



(b) Block delay distribution (with $W = 15$-second windows).

Figure 8: 15 seconds window cost and delay

RQ2: How to characterize the Ether-delay tradeoff attained by different batching policies? And how much Ether cost per invocation can iBatch save while with minimal block delay (compared with unbatched baseline B0)?

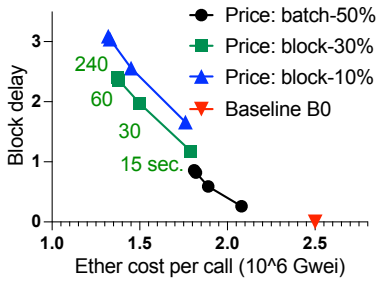


Figure 9: Tradeoff between Ether cost and block delay under varying Gas price and batch window

Motivation: On Ethereum, the cost metric that an end user (Ether owner) cares the most is the amount of Ether she needs to pay out of pocket for invocations. The Ether cost per invocation is the product of the Gas of an invocation and the Gas price of the (batch) transaction. RQ2 focuses on measuring the Ether cost per invocation.

Many DeFi applications are very sensitive to the timing of invocations, that is, when an invocation is included in the blockchain. Additional delay to the invocation may invite loss of financial opportunity (e.g., in an auction), increase exploitability under frontrunning attacks, et al. We mainly use the 1block online mechanism (in § 4.2) that causes minimal block delay to batched invocations.

Experiment methodology: We follow the same transaction-replaying method described before, with the only exception: To measure delays under 1block, we have to know each transaction's submission time. This is obtained by crawling the transaction's "pending" time from website etherscan.io (an example link is [14]). Then, a transaction's submission time is its block time minus the pending time.

Experiment settings: We collect a trace of 100, 000 Ethereum transactions, each invoking Tether's `transfer()` function [22]. In real life, these transactions were submitted in one day on Oct. 4, 2020. We did not collect more transactions as replaying 100, 000 transactions takes around 570 minutes, which is long enough for conducting our experiments.

We replay the transaction trace in the following manner: We apply a pre-configured batching policy to generate a batch transaction, say at time t. How a block is produced and which transactions will be included in a block are simulated in the following manner (an approach also used in [41]): Given a specified Gas price p, the batch transaction submitted at time t will be included in the first block produced after t which includes at least one transaction with Gas price lower than p.

Following the above method, we replay the trace with iBatch with 1block mechanism and under different batching policies.

Results: When replaying the trace, we use three pricing policies, namely batch-50%, block-30% and block-10%, as described in § 4.2. We measure each transaction's Gas and multiply it with its Gas price to obtain the transaction's Ether cost. By summing the Ether costs of the transactions in a unit time period and dividing it with the number of calls, we report the average Ether cost per call in Figure 8a where the unit period is 5 windows (or $5\times15 = 75$ seconds). The results show that iBatch of policy block-10% achieves the lowest Ether cost, which is 31.52% lower than that of the unbatched baseline (B0). By comparison, iBatch under the batch-50% policy saves 19.06% Ether per invocation than the baseline B0.

We also plot the block delays of iBatch under these three configurations in Figure 8b. The figure shows the distribution of batch transactions in their block delays. As can be seen, under the batch-50% policy, the majority of the batch transactions have a minimal delay under three blocks. In average, the delay of iBatch under the pricing of $batch - 50\%$ is 0.26 blocks, the delay under the price of 30 Gwei per Gas is 1.18 blocks, and the delay under the price of 10 Gwei per Gas is 1.66 blocks.

We then report the tradeoff between block delay and Ether cost per call under varying Gas prices of batch transaction and batch windows. The result is in Figure 9. It can be seen with the batch transaction of the same Gas price (i.e., block-30% in the figure), the block delay increases and Ether per call decreases as the batch window grows from 15 seconds through 240 seconds. The unbatched baseline B0 incurs 0 block delay and 2.5×10^6 Gwei per call. In comparison to the baseline, with the batch-50% policy and 15-second batch window, iBatch saves 19.06% cost at the expense of delaying invocations by 0.26 blocks. With the policy of block-10% and 15-second batch window, iBatch saves 31.52% cost at an average 1.66 block delay.

7 RELATED WORKS

Public blockchains are known to cause high costs and to have limited transaction throughput [27]. Reducing the cost of blockchain applications is crucial for real-world adoption and has been studied in the existing literature. **Layer-two protocols**: Another approach, dubbed layer-two designs [19, 26, 28, 38], focuses on designing add-on to a deployed blockchain system by designing extensions including smart contracts on-chain and services off-chain. The notable example is payment networks [19, 26, 38] that place most application logic of making a series of micro-payments off the blockchain while resorting to blockchain for control operations (e.g., opening and closing a channel) and error handling. In a sense, a payment channel "batches" multiple repeated micro-payments into minimally two transactions. State channels [28] generalize the idea to support the game-based execution of smart contracts. The batching in this line of work is orthogonal to that in iBatch: 1) iBatch is generally applicable to any smart contracts, while payment channel/network is specific to repeated micro-payments between a fixed pair of buyer and seller. 2) iBatch can further reduce the Gas of a payment channel. Specifically, the invocations to the smart contracts in a payment channel (namely HTLC) can be batched to amortize the transaction fee over multiple operations to open/close a channel [19]. **Ethereum Gas optimization**: GRuB [35] supports gas-efficient data feeds onto blockchains, for decentralized financial applications (DeFi). For Gas efficiency, it employs a novel technique that replicates data feeds adaptively to the workload. iBatch can complement GRuB's adaptive data-feed to achieve a higher level of Gas efficiency. Gasper [25] detects and fixes the "anti-patterns" in smart contracts that excessively cost Gas. While Gasper aims at reducing the on-chain computation in smart contracts, iBatch reduces the transaction fee in smart-contract invocations.

8 CONCLUSION

This paper presents iBatch, a security protocol and middleware system to batch smart-contract invocations over Ethereum. The design of iBatch addresses the tradeoff between security, cost effectiveness and delay. The result shows that compared with the baseline without batching, iBatch effectively saves cost per invocation with small block delay.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers and shepherd. The first five authors at Syracuse University are partially supported by the National Science Foundation under Grant CNS1815814 and DGE2104532. Xiapu Luo is partially supported by Hong Kong RGC Project 152223/20E and Hong Kong ITF Project GHP/052/19SZ. Ting Chen is partially supported by Project 2018YFB0804100 under National Key R&D Program of China and Project 61872057 under National Natural Science Foundation of China.

A APPENDIX: SECURITY ANALYSIS

Security against invocation-forging `Batcher`: Invocation forging refers to that given a caller A who did not send an invocation X, the `Batcher` forges the invocation X and falsely claims it is sent by caller A. In iBatch, the hardness of `Batcher` making `Dispatcher` accept a forged invocation can be reduced to the hardness of forging a digital signature (as in Protocol Step 3) in § 3.2), which is known to be with negligible probability.

Security against invocation-omitting `Batcher`: Invocation omission refers to that the `Batcher` omits an invocation in a batch while falsely acknowledging the victim client the inclusion of her invocation. In iBatch, an omitted invocation in a batch transaction included in the blockchain cannot be concealed from the victim client. To prove it, omitting an invocation and concealing it from the client requires producing a sufficient number of fake blocks (e.g., 6 blocks in Bitcoin) where one of the blocks includes a fake transaction that includes the omitted invocation. Thus, this is equivalent to mounting a successful double-spending attack on the underlying blockchain, which is assumed to be hard.

Beyond attack detectability, iBatch can be extended to prevent invocation omission, via an external incentive scheme (similar to IKP [36]) which punishes a misbehaving `Batcher` and disincentivizes her future omission of invocations. Recent work [34] examines the security of real-world transaction relay services (similar to `Batcher`) under denial of service attacks, from which the mitigation schemes can be applied to harden the DoS security of the `Batcher`.

Security against invocation-replaying `Batcher`: Invocation replaying refers to that the `Batcher` replays an invocation in a successful batch transaction without informing the victim client. There are different forms of replaying attacks, including R1) the `Batcher` replaying invocations twice (or multiple times) in the same batch transaction, R2) the `Batcher` replaying a batch transaction with the same nonce $\texttt{nonce}_\texttt{B}$, R3) the `Batcher` replaying a batch transaction's data twice with two different $\texttt{nonce}_\texttt{B}$, and R4) the `Batcher` intentionally generating smaller batches. Here, we don't consider the case of the `Batcher` replaying an invocation in two different batch transactions, in which one replayed copy must be a forged invocation to the caller and which can thus be prevented.

Overall, iBatch prevents invocation replaying in forms of R1, R2, R3 and R4. The following is the security analysis.

Consider R1 that a replayed invocation cannot appear in the first round message (`bmsg`), as the victim client can easily detect it and refuse to sign the joint message in the second round. If an invocation is replayed in the batch transaction, the `Batcher` has to modify the jointly signed message (`bmsg_sign`) and forge all the second-round signatures, known to be hard.

Consider Case R2 that the `Batcher` replays an entire batch transaction, that is, sending the batch transaction with the same nonce twice. Such a transaction-level replay will be prevented by Ethereum's native replay protection based on $\texttt{nonce}_\texttt{B}$.

Consider Case R3 that the `Batcher` replays a batch transaction with different $\texttt{nonce}_\texttt{B}$. The `Dispatcher`'s verification will fail because the original $\texttt{nonce}_\texttt{B}$ is signed by callers (recall Equation 3).

Consider Case R4 that the `Batcher` may intentionally generate small batches; for instance, instead of one batch of 10 invocations, it generates two smaller batches, each 5 invocations. This is not necessarily an attack as the batch transaction size is bounded by Ethereum's native block Gas limit. But it could be a protocol deviation and can be detected: It will result in two batch transactions included in Ethereum at a similar time (w.r.t., the batch time window). An auditing caller can detect the anomaly by inspecting the public Ethereum transaction and open disputes for further resolution.

Security against denial-of-service callers: iBatch can be extended to guarantee that a denial-of-service caller cannot delay the overall processing of a batch. In the extension, the `Batcher` enforces a timeout on waiting for callers' batch signatures. After the timeout, the `Batcher` generates the batch transactions, and `Dispatcher` does not forward to the callee smart contract an invocation whose batch signature is missing. With this extension, a denial-of-service caller who delays her batch signature after the timeout will be ignored and does not invoke the callee smart-contract function, while other invocations are not affected. The DoS caller can only cause the fee of batch transaction to increase, which can be further detected and blacklisted by the `Batcher`.

This work does assume that the `Batcher` is always available. In practice, we consider this is a reasonable assumption as such a service can be run on highly-available cloud platforms, and real-world transaction relay services such as infura.io that require clients to trust its availability are already operational and widely adopted. The `Batcher` service has incentives to protect its business and defend against external denial-of-service attacks.

Security against caller impersonator in collusion w. `Batcher`: Recall Figure 3 that normally, the `Batcher` sends to Caller 2 the batch message `bmsg` that includes Caller 1's public key PK_1 and her invocation $\texttt{call}_1$. Caller 2 simply verifies $\texttt{call}_1$ against the provided PK_1 and, if it passes, signs `bmsg` before returning it to `Batcher`. The malicious `Batcher` may include in `bmsg'` an impersonator's invocation, that is, $\texttt{call}'_1$ and her public key PK'_1. In this case, Caller 2 still verifies $\texttt{call}'_1$ in the `bmsg` against PK'_1, which passes and leads to Call 2's signature on `bmsg'`. Message `bmsg'` is returned to and signed by `Batcher`, is further verified successfully by `Dispatcher`, and gets $\texttt{call}'_1$ forwarded to the callee smart contract. The callee will handle the internal call sent from PK'_1 and leave the actual sender (i.e., PK_1) unharmed.

REFERENCES

[1] Retrieved June, 2021. An analysis of batching in Bitcoin. https://coinmetrics.io/batching/.

[2] Retrieved June, 2021. Binance Smart Chain, A Parallel Binance Chain to Enable Smart Contracts. https://www.binance.org/en/smartChain.

[3] Retrieved June, 2021. Blockchain Oracles for Connected Smart Contracts, Chainlink. https://chain.link/.

[4] Retrieved June, 2021. Contract address (Binance token) on Etherscan. https://bit.ly/3dHo7Pc.

[5] Retrieved June, 2021. Decentralized Ethereum Asset Exchange. https://idex.market/eth/idex.

[6] Retrieved June, 2021. EIP-2711: Sponsored, expiring and batch transactions. https://eips.ethereum.org/EIPS/eip-2711.

[7] Retrieved June, 2021. EIP-3005: Batched meta transaction. https://eips.ethereum.org/EIPS/eip-3005.

[8] Retrieved June, 2021. EIP-3074: AUTH and AUTHCALL opcodes. https://eips.ethereum.org/EIPS/eip-3074.

[9] Retrieved June, 2021. The EOA owning IDEX smart contract on etherscan. https://etherscan.io/address/0xa7a7899d944fe658c4b0a1803bab2f490bd3849e.

[10] Retrieved June, 2021. EOS: Blockchain software architecture. https://eos.io/.

[11] Retrieved June, 2021. Ethereum Blockchain Public Dataset (hosted by Google BigQuery). https://console.cloud.google.com/bigquery?project=bigquery-public-data&page=dataset&d=ethereum$_b$lockchain&p=bigquery-public-data&redirect$_f$rom$_c$lassic=true.

[12] Retrieved June, 2021. Ethereum Daily Transactions Chart. https://etherscan.io/chart/tx.

[13] Retrieved June, 2021. Ethereum project. https://www.ethereum.org/.

[14] Retrieved June, 2021. Ethereum transaction hash details; an example link. https://shorturl.at/huGH9.

[15] Retrieved June, 2021. Ethereum Transactions historical chart. https://bitinfocharts.com/comparison/ethereum-transactions.html.

[16] Retrieved June, 2021. Geth: the Go Client for Ethereum. https://www.ethereum.org/cli#geth.

[17] Retrieved June, 2021. Github repository: iosiro/airdropper. https://github.com/iosiro/airdropper/blob/master/contracts/Airdropper.sol.

[18] Retrieved June, 2021. The IDEX smart contract on etherscan. https://etherscan.io/address/0x2a0c0dbecc7e4d658f48e01e3fa353f44050c208.

[19] Retrieved June, 2021. Lightnning network, Scalable, Instant Bitcoin/Blockchain Transactions. https://lightning.network

[20] Retrieved June, 2021. Native Meta-Transaction Proposal Roundup. https://ethresear.ch/t/native-meta-transaction-proposal-roundup/7525/2.

[21] Retrieved June, 2021. SushiSwap, Fleeing Ethereum Fees, Is Now Live on Binance Smart Chain, Fantom, Others. https://www.coindesk.com/sushiswap-fleeing-ethereum-fees-is-now-live-on-binance-smart-chain-fantom-others.

[22] Retrieved June, 2021. Tether, stable digital cash on the blockchain. https://tether.to/.

[23] Retrieved June, 2021. Uniswap V2 Overview. https://uniswap.org/blog/uniswap-v2/.

[24] Joseph Bonneau, Arvind Narayanan, Andrew Miller, Jeremy Clark, Joshua A. Kroll, and Edward W. Felten. 2014. Mixcoin: Anonymity for Bitcoin with Accountable Mixes. In *Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*. 486–504. https://doi.org/10.1007/978-3-662-45472-5$_3$1

[25] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*. 442–446. https://doi.org/10.1109/SANER.2017.7884650

[26] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2018. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution. *CoRR* abs/1804.05141 (2018). arXiv:1804.05141 http://arxiv.org/abs/1804.05141

[27] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed E. Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. 2016. On Scaling Decentralized Blockchains - (A Position Paper). In *FC 2016 Workshops (Lecture Notes in Computer Science, Vol. 9604)*, Jeremy Clark, Sarah Meiklejohn, Peter Y. A. Ryan, Dan S. Wallach, Michael Brenner, and Kurt Rohloff (Eds.). Springer, 106–125. https://doi.org/10.1007/978-3-662-53357-4$_8$

[28] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. 2018. General State Channel Networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 949–966. https://doi.org/10.1145/3243734.3243856

[29] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert van Renesse. 2016. Bitcoin-NG: A Scalable Blockchain Protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, Katerina J. Argyraki and Rebecca Isaacs (Eds.). USENIX Association, 45–59. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eyal

[30] Michael Fröwis and Rainer Böhme. 2019. The Operational Cost of Ethereum Airdrops. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2019 International Workshops, DPM 2019 and CBT 2019, Luxembourg, September 26-27, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11737)*, Cristina Pérez-Solà, Guillermo Navarro-Arribas, Alex Biryukov, and Joaquín García-Alfaro (Eds.). Springer, 255–270. https://doi.org/10.1007/978-3-030-31500-9$_1$7

[31] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. 2017. TumbleBit: An Untrusted Bitcoin-Compatible Anonymous Payment Hub. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/tumblebit-untrusted-bitcoin-compatible-anonymous-payment-hub/

[32] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 583–598. https://doi.org/10.1109/SP.2018.000-5

[33] Aurora Labs. [n.d.]. IDEX: A Real-Time and High-Throughput Ethereum Smart Contract Exchange. https://idex.market/static/IDEX-Whitepaper-V0.7.6.pdf. ([n. d.]).

[34] Kai Li, Jiaqi Chen, Xianghong Liu, Yuzhe Tang, XiaoFeng Wang, and Xiapu Luo. 2021. As Strong As Its Weakest Link: How to Break Blockchain DApps at RPC Service. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/as-strong-as-its-weakest-link-how-to-break-blockchain-dapps-at-rpc-service/

[35] Kai Li, Yuzhe Tang, Jiaqi Chen, Zhehu Yuan, Cheng Xu, and Jianliang Xu. 2020. GRuB: Gas-Efficient Blockchain Storage via Workload-Adaptive Data Replication. *ACM/IFIP Middleware 2020* (2020). http://arxiv.org/abs/1911.04078

[36] Stephanos Matsumoto and Raphael M. Reischuk. 2017. IKP: Turning a PKI Around with Decentralized Automated Incentives. In *SP 2017*. IEEE Computer Society, 410–426. https://doi.org/10.1109/SP.2017.57

[37] Sarah Meiklejohn and Rebekah Mercer. 2018. Möbius: Trustless Tumbling for Transaction Privacy. *PoPETs* 2018, 2 (2018), 105–121. https://doi.org/10.1515/popets-2018-0015

[38] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. 2017. Sprites: Payment Channels that Go Faster than Lightning. *CoRR* abs/1702.05812 (2017). arXiv:1702.05812 http://arxiv.org/abs/1702.05812

[39] István András Seres, Dániel A. Nagy, Chris Buckland, and Péter Burcsi. 2019. MixEth: efficient, trustless coin mixing service for Ethereum. *IACR Cryptol. ePrint Arch.* 2019 (2019), 341. https://eprint.iacr.org/2019/341

[40] Yibo Wang, Qi Zhang, Kai Li, Yuzhe Tang, Xiapu Luo, and Ting Chen. 2021. iBatch: Saving Ethereum Fees via Secure and Cost-Effective Batching of Smart-Contract Invocations. *CoRR* abs/2106.08554 (2021). arXiv:2106.08554 https://arxiv.org/abs/2106.08554

[41] Sam M. Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William J. Knottenbelt. 2021. SoK: Decentralized Finance (DeFi). *CoRR* abs/2101.08778 (2021). arXiv:2101.08778 https://arxiv.org/abs/2101.08778