Adding Fairness to Order: Preventing Front-Running Attacks in BFT Protocols using TEEs

Chrysoula Stathakopoulou
IBM Research Europe - Zurich
ETH Zurich

Signe Rüsch*
TU Braunschweig

Marcus Brandenburger
IBM Research Europe - Zurich

Marko Vukolić*
Protocol Labs

***Abstract*—This paper presents Fairy: a modular system that augments any Total Order Broadcast (TOB) protocol with fairness properties, namely input (request) causality and sender obfuscation. With these properties, Fairy helps to prevent front-running attacks where an adversary can interfere with the order of requests depending on request payload and sender identity, allowing for substantial application-level consequences.**

Fairy leverages Trusted Execution Environments (TEEs) to implement fairness on top of a TOB-based ordering service. Fairy collocates TEEs with ordering service nodes effectively making them run as trusted proxies of actual TOB clients. TEEs help Fairy to achieve both input causality and sender obfuscation — previous related systems addressing only input causality.

We evaluate Fairy on top of a recent, efficient Byzantine Fault-Tolerant (BFT) TOB protocol and compare it to state-of-the-art BFT TOB with input causality. We show that Fairy improves state-of-the-art throughput by 66% and reduces latency by 50%.

***Index Terms*—total order broadcast, trusted execution, front-running attacks**

I. Introduction

Total order broadcast (TOB) is a fundamental building block for implementing replication in fault-tolerant distributed systems. Recently, research in Byzantine fault-tolerant (BFT) [24] total order protocols has been intensified, since they are a core technology in decentralized blockchain and distributed ledger systems, e.g., Hyperledger Fabric [4], responsible for maintaining identical replicas of the ledger of requests across all participants. In a nutshell, in a distributed setting, TOB guarantees that all nodes establish a total order in which requests will be processed and executed, i.e., applied to the replicated state. This is the basis of guaranteeing consistent state replication in state-machine replication (SMR), blockchain, and distributed ledger systems.

However, total order broadcast does not guarantee that the order of request execution is the same as the order in which requests are submitted, which can be critical for some applications. In particular, revealing the content of a request without establishing in advance the order of execution of the request opens a surface for front-running attacks [26]. Reiter and Birman [32] give such an example of a front-running attack for a stock trading service. Assume a distributed service that trades stocks, and two clients: Alice and Mallory. Alice issues a request to purchase shares of stock. After discovering the intended purchase, a corrupt node colludes with Mallory to issue a request for the same stock. If Mallory's request is executed first, the demand for the stock increases, inflating the value for the purchase of the stock for Alice's request.

In the example above, the invocation of Mallory's request depends on the content of Alice's request. We refer to such a dependency as a *causal dependency*. On a high level, causal TOB prevents a causal dependency of two requests before the total order of the requests has been established.

We extend the front-running attack definition to cover a wider range of attacks. Assume now that Mallory is aware that Alice will purchase stock of some company. Mallory now waits to see Alice's transaction with the stock service before acting as in the previous example to inflate its price. The information that Mallory used here is the identity of the sender of the request. To prevent such an attack, total order broadcast should hide the identity of the sender. We refer to this as *sender obfuscation*.

In traditional markets, such behavior is potentially punishable, e.g., as some form of insider trading. However, with the rise of blockchain systems, as well as digitization in general, opportunities for front-running attacks have increased. Examples include bidding for domain names, ICOs, gambling services, and decentralized exchange services [18]. In such use cases, the traditional centralized legal repercussions are not only incompatible with the decentralized nature of such systems but may also be impractical, since users are often pseudonymous or anonymous.

To that end, preventing front-running attacks is a more natural solution. Existing approaches use threshold encryption [11], [16] or commit-reveal schemes [17] to add input causality to total order broadcast. The former require a trusted key setup and expensive per transaction cryptographic operations while the latter prove prone to malicious clients who can block the system's progress.

In this work we leverage modern hardware-aided Trusted Execution Environment (TEE) technology, such as Intel Software Guard Extensions (SGX), to enhance BFT total order broadcast protocols with input causality and sender obfuscation in a modular and efficient way. This technology allows applications to run securely in an isolated environment and thereby protect the execution integrity and confidentiality of the application. With additional support for remote attestation, the use of TEEs enables trust to be established in an applica-

*Work done while at IBM Research Europe - Zurich tion executed remotely on a potentially misbehaving host.

Our contributions can be summarized as follows:

- We introduce *fair total order broadcast* (FTOB) abstraction which extends TOB with two fairness properties: input causality and sender obfuscation.
- We introduce Fairy protocol which implements FTOB in a modular way on top of TOB. Fairy can be built on top of any TOB protocol with external validity [10].
- We implement Fairy using Intel SGX as a TEE on top of state-of-the-art Mir-BFT [35]. We show that Fairy outperforms state-of-the-art causal TOB and introduces an overhead of approximately 20% to baseline TOB performance.

The rest of this paper is structured as follows. Section II introduces the system model and our assumptions, as well as TOB and FTOB abstractions. Section III discusses in detail front-running attacks and their mitigations. Section IV introduces our extended model with trusted execution environments and gives an overview of how we achieve FTOB with this setup. Fairy details are given in Section V and correctness arguments in Section VI. Section VII discusses how we implement Fairy with Intel SGX on top of Mir-BFT. Section VIII studies the performance of Fairy focusing on the overhead it introduces to the underlying TOB protocol and comparison to existing solutions. Section IX compares our work with existing literature. Section X concludes.

II. Model and Abstractions

Our system comprises a set $Nodes$ of n nodes and a non-intersecting set of client processes $Clients$ of arbitrary size. Each node hosts a TOB process. We assume a public key infrastructure (PKI) under which the processes (TOB and client processes) are identified by their public keys. We further assume that all TOB processes' identities are lexicographically ordered and we use a bijection set $I = [0 \dots n-1]$ to argue about identities.

We assume a dynamic adversary which in every execution may control up to f nodes, such that $f < \frac{n-1}{3}$, and any number of clients. The adversary may fully control the operating system of the nodes and thereby tamper with the execution and the memory of TOB processes and clients it controls. The adversary can also arbitrarily drop, delay, or modify messages sent from and to the nodes it controls. Finally, we assume a computationally bound adversary which cannot break cryptographic primitives.

The TOB processes implement a Byzantine fault-tolerant total order (atomic) broadcast service, which is extended with the property of external validity [11]. A process acting as a client to TOB invokes TOB for some request r by sending a message $\langle TOB-CAST, r\rangle$ to TOB processes. For simplicity we say the client *broadcasts* r. TOB for request r terminates for a process p once p outputs a message $\langle TOB-DELIVER, sn, r, \pi\rangle$, where sn is a monotonically increasing sequence number and π some proof. For simplicity, we say p *delivers* r with sequence number sn and proof π. A protocol solves TOB with predicate Q if it satisfies the following properties:

TOB1 **Integrity:** If all clients are correct and a correct process delivers r then some client broadcast r.
TOB2 **Agreement:** If two correct processes deliver requests r and r' with sequence number sn, then $r = r'$.
TOB3 **No-duplication:** If a correct process delivers request r with sequence numbers sn and sn', then $sn = sn'$.
TOB4 **Totality:** If a correct process delivers request r, then every correct process eventually delivers r.
TOB5 **Liveness:** If a correct client broadcasts request r, then some correct process p eventually delivers r.
TOB6 **External Validity:** If a correct process delivers a request r with a sequence number sn and a proof π, then $Q(r, \pi)$ holds.

Next we define *fair TOB* abstraction, or simply FTOB. A client process with identity id invokes FTOB for some request r by sending a message $\langle FTOB-CAST, F(id, r)\rangle$ to nodes in $Nodes$, where $F : \{0,1\}^{k_1} \times \{0,1\}^{k_2} \rightarrow \{0,1\}^{l}$ and where k_1 (resp., k_2) is the bit-length of the client's identity (resp., request) and where l is a security parameter. For simplicity we say the client *f-broadcasts* r.

FTOB for request r terminates for a node i in $Nodes$ once i outputs a message $\langle FTOB-DELIVER, sn, r\rangle$, where sn is a monotonically increasing sequence number. For simplicity we say i *f-delivers* r with sequence number sn. FTOB satisfies the properties of total order broadcast TOB1-5 (note External Validity is not needed) and further satisfies the properties of *input causality* and *sender obfuscation*. We define input causality similarly to the non-malleability property in [15] and sender obfuscation as the indistinguishability of invocations by correct clients. In the following, we first define *causal dependency* which we need to define input causality. Then we proceed with defining FTOB properties.

Definition 1 (Causal dependency)**.** *For a request r, let the adversary know $F(r, id)$. Then we say that a request r' causally depends on request r and symbolize $r \leftarrow r'$, if the adversary can choose r' as some function of r'' ($r' = f(r'')$), where $r'' = r$ with non-negligible probability.*

FTOB1-FTOB5: same as properties TOB1-TOB5.
FTOB6 **Input Causality:** Let some correct node f-deliver r and r' with sequence numbers sn and sn' respectively. If $r \leftarrow r'$ and if a correct client f-broadcast r, then $sn < sn'$.
FTOB7 **Sender Obfuscation:** Let two correct clients with identities id and id', $id \neq id'$, f-broadcast requests r and r' respectively such that some correct node i f-delivers r and r' with sequence numbers sn and sn' such that $sn < sn'$. Then the adversary cannot distinguish $c = F(r, id)$ from $c' = F(r', id')$, except with negligible probability, before r is f-delivered by any node.

Note that the adversary given function F can always distinguish an invocation $\langle FTOB-CAST, F(id, r)\rangle$ of a correct client id from an invocation $\langle FTOB-CAST, F(id', r')\rangle$ of a malicious client of whom the adversary knows id' and r'. FTOB only guarantees the obfuscation of a request from a correct client among other requests from correct clients.

III. The Problem of Front-Running Attacks

On a high level, with the term front-running attack we refer to the misuse of information about a request by an adversary in order to first create and execute some other request that displaces, suppresses, or precedes the original request [18]. In this section we define front-running attacks, discuss how front-running attacks are addressed in the literature, and justify why input causality and sender obfuscation properties of our FTOB are relevant to preventing front-running attacks.

We define a front-running attack as follows.

Definition 2 (Front-Running Attack)**.** *Let S be a service that executes client requests and let r be a request. An adversary successfully performs a front-running attack if it can create another request $r' = f(r)$, where f is some function, which is executed from S before r with non-negligible probability.*

We consider a service that implements a TOB protocol as introduced in the previous section, where clients invoke TOB by submitting their requests to the TOB nodes. Totality (TOB4) and liveness (TOB5) properties guarantee that the request from a correct client will eventually be delivered and executed by all correct nodes. Agreement (TOB2) property guarantees that all correct nodes will execute the request in the same order. However, TOB does not guarantee that the order in which clients submit their requests is the same as the order in which they are delivered. A malicious node can delay some request r of interest to favor some other request with a causal dependency on r, effectively performing a front-running attack. This is indistinguishable from the request r being delayed by the network and makes it impossible to deduce the order in which two different requests were sent to the system. Note that this differs in the crash fault-tolerant (CFT) model, where First-In-First-Out (FIFO) order, also referred to as CFT causality [23], can be achieved, for instance, by using a vector clock [19].

In the BFT model, as discussed in Section II, *input causality* (FTOB6) is a non-malleability property that requires the client to hide the request from the nodes of the service and the property is satisfied if the adversary who controls the Byzantine nodes cannot create some other request r' as a function of r before r is delivered with some sequence number. Input causality prevents front-running attacks as the adversary can only create a causal dependency on r after it is delivered with some sequence number sn. Thus, any other request r', such that $r \leftarrow r'$, can only obtain a sequence number $sn' > sn$, forcing r' to be executed after r.

Hiding the request from the nodes suggests a non-malleable cryptographic primitive. Reiter et al. [32] introduced causal TOB with threshold encryption, which was later refined in [11]. In a nutshell, the client encrypts the request with a public key of the service and invokes TOB with the encrypted request. Only after the encrypted request is delivered, do the nodes decrypt it with their key shares and combine the decryption shares to obtain the request.

While this scheme satisfies input causality, it requires a distributed key setup. Moreover, threshold decryption and decryption share combination per request are computationally expensive. These issues were addressed by Duan et al. [17] by replacing the threshold encryption with a non-malleable commit reveal scheme with associated data. The clients first invoke TOB for a commitment to the request along with a unique identifier linked to their identity and only after the commitment is delivered with the same unique identifier, do the clients invoke TOB for the decommitment of the request. However, in this scheme faulty clients can block progress. To preserve input causality it is necessary that requests are delivered in the same order as their commitments. Even if a single client crashes or is malicious and, thus, does not submit the decommitment, the protocol blocks. The authors suggest a periodic garbage collection of commitments whose decommitment is pending. This, however, requires strong synchrony assumptions; otherwise, it cannot be guaranteed that the nodes do not discard the commitment of some request that is delayed by the network.

In this work, we introduce a solution that is efficient, in the sense that it does not require threshold decryption nor extra rounds of TOB, such that it guarantees input causality even with Byzantine faulty clients and without imposing extra synchrony constraints.

Another aspect we examine w.r.t. front-running attacks is the sender's (client's) identity. As discussed in Section I, leaking the sender's identity, combined with external information, such as previous client's requests, can be used by an adversary to launch a front-running attack. We, therefore, extend Definition 2 with the sender's identity.

Definition 3 (Extended Front-Running Attack)**.** *Let S be a service that executes client requests and let r be a request from a client with identity id. An adversary successfully performs a front-running attack if it can create another request $r' = f(r, id)$, where f is some function, which is executed by service S before r without negligible probability.*

Previously mentioned mitigations of front-running attacks do not take this aspect into account. In fact, for the commit-reveal scheme in [17] verifying that client's identity matches the associated data of the commitment is integral to the protocol to prevent unauthorized parties flooding the system with invalid commitment messages. Similarly, the solution in [11] cannot authenticate the requests before combining the decryption shares unless the client's identity is attached to the request in cleartext.

In Section II we introduced the property of *sender obfuscation* (FTOB7) to build the FTOB abstraction, which combined with the input causality property prevents the extended front-running attack, since the adversary cannot distinguish the senders of different requests. Note that the probability of the adversary performing a successful extended front-running attack becomes negligible with the sender obfuscation property, assuming that the number of clients the adversary controls is bounded, which, in practice, constitutes a reasonable assumption. Moreover, anonymizing the client on the network level is out of the scope of this work. Complementary solutions can be deployed, e.g., communication over Tor [2]. Such a deployment has been demonstrated in Honeybadger BFT [30].

Our protocol, as we discuss in the next sections, implements FTOB by satisfying both input causality and sender obfuscation on top of TOB by leverage trusted execution environments (TEEs). In a nutshell, a request r, submitted to TOB, is encrypted in a way that it can only be decrypted by the TEE once r has been delivered. That is, the TEEs disclose (the plaintext contents of) r to the nodes of the system only after acquiring proof that the request is assigned a sequence number sn, such that r will be eventually delivered with sequence number sn by all correct nodes. We guarantee sender obfuscation by also encrypting information pertaining to the client's identity until proof of delivery is provided to the TEEs. Notably, Fairy can obfuscate clients' identity *without* the danger of them flooding the system with invalid requests. This is because the integrity and the authenticity of the requests are checked inside the TEE before TOB. Additionally, application-specific validity checks could be performed inside the TEE but this is beyond the scope of this work.

IV. Trusted Execution Environments

TEEs provide an isolated environment for executing applications securely. TEEs guarantee confidentiality and integrity to the application code and data within the execution context.

Remote attestation. In order to establish trust in a TEE-based application, the code must be initially inspected by the participants (processes), i.e., the clients and TOB nodes, and then compute a cryptographic hash ϕ over the application code, which is further used to identify the code running inside the TEE. During the lifetime of a TEE, the processes can run a remote attestation protocol, see Section V-A2, that attests the code the TEE is running and details about the execution environment. In particular, processes can verify that the TEE runs the code matching the code identity they have inspected initially. In other words, with remote attestation, processes can verify that the TEE which they are interacting with is unmodified and runs the expected software. Moreover, the attestation protocol also allows additional data to be embedded into the attestation, which is particularly useful when attaching dynamic data to the attestation, for instance, a public key that is generated during launching. This key generation may be part of the application running in a TEE. Since the memory of a TEE is protected and can only be accessed by the application code within the TEE, all code and data, including the secret keys, are protected against unauthorized access. Finally, TEEs provide access to a secure random number generator.

TEEs in our model. We extend our system model described in Section II to include TEEs as follows. Each node runs along with the untrusted TOB process and a trusted process inside a TEE. Like the other system processes (clients and TOB), processes running in the TEEs are also identified by a public key under a PKI.

We assume a typical fault model for TEEs: an adversary may completely control, including physical access to, the host system. This means the applications prior to TEE entry, the operating system, as well as the hypervisor are untrusted. However, the adversary does not have control of the execution and memory inside the TEE. The adversary may also have control of the communication to and from the trusted execution environment. Finally, TEEs run in user space and depend on the cooperation of the host system, which means a compromised host can launch denial-of-service attacks, e.g., by not starting, not entering, or stopping the trusted processes.

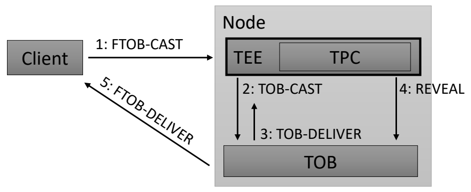

Fig. 1. Invocation of TOB and FTOB abstractions.

V. Fair TOB with Fairy

In this section, we describe Fairy distributed trusted setup and protocol. We design Fairy in a modular way so that it can be easily implemented on top of every protocol that satisfies the TOB properties we defined in Section II. The key to our modular design is that the process that runs in the TEE of each node acts as a trusted proxy to the clients of the system. For the rest of this work we will, therefore, refer to the process inside the TEE as *trusted client proxy*, or as TPC for short. TPCs act as clients to the underlying TOB protocol, invoking TOB on behalf of the clients, after decrypting and authenticating the encrypted clients' requests. TPCs are also responsible for disclosing client requests in cleartext to their hosting node, after receiving a proof that the request is delivered by TOB. Figure 1 illustrates how the processes interact using the TOB and FTOB interfaces.

A. Trusted setup

1) Trusted public keys and identities: We assume a standard PKI with a root certificate CR_0 signed by a trusted Certificate Authority (CA). We could also argue about multiple root certificates signed by multiple CAs by replacing CR_0 with a vector of certificates, but without loss of generality, we will assume only one for simplicity. Each process i in the system (i.e., TOB, client, and TPC) is equipped with a public-private key pair PK_i,SK_i used for authentication. The public key PK_i is also used as the identity of the process. We assume that each PK_i is issued with a certificate signed by CR_0. Each node, upon bootstrapping its TPC, provides the TPC with CR_0. Each TPC is equipped with a public-private key pair EK_i,DK_i for encrypted communication with the clients.

2) Remote Attestation: Any processes i can at any time perform remote attestation with any TPC j to verify that a genuine TEE protects the intended code (identified using ϕ) and its data. In detail, the attestation cryptographically binds the TPC state to the TEE, in particular, the public keys of the TPC and the root certificate provided during bootstrapping.

The attestation protocol briefly works as follows: (1) Process i computes h = Hash(CR_0), keeps h; picks a nonce z and sends to host message $\langle RA, z\rangle$. (2) The hosting node forwards $\langle RA, z\rangle$ to the TPC j. (3) TPC j computes h'=Hash(CR_0) of the root certificates it has stored locally. (4) TPC j creates an attestation including h', Hash(PK_j), Hash(EK_j), z, and its code identity ϕ'. (5) TPC j returns the attestation, PK_j, and EK_j to process i. (6) Process i verifies that the attestation is "correct", that is, it has been produced by a genuine TEE and that it corresponds to $h == h'$, Hash(PK_j), Hash(EK_j), and the nonce z initially sent. Moreover, the client checks that the attestation contains the expected TPC code identity ϕ.

Note that a malicious node may intercept and tamper with the communication between its TPC q and process p, but any modification on the attested data will be detected by q.

The trusted setup described here assumes static membership. To support dynamic configuration changes, Fairy can be easily extended. For instance, nodes can provide the updated root certificate(s) and a revocation list to their TPC and signal to the clients to re-perform attestation.

3) Client registration: Each client c needs to register a symmetric key $SymK_c$ with all TPCs in correct nodes. The client will use the symmetric key to encrypt the requests it submits to the TPCs (we use RSA-OAEP encryption). Each client further needs to register a one-time client' identity ($OTID_0$) with each TPC. The client sends $OTID_0$ in a future message in cleartext along with the first request r to identify with the TPC, since request r, which includes client's long term identity, i.e., the public key, is encrypted. $OTID_k$ with $k > 0$ is renewed by the client with every new request.

Client registration starts with the client asking for a remote attestation of the TPC public keys and root certificate the TPC uses with the protocol described in Section V-A2. Client c then encrypts the registration information $Reg = \langle SymK_c, OTID_0, PK_c, Cert_c, nonce\rangle$ and sends a message $\langle REGISTER, E(Reg)_{EK_i}\rangle_{\sigma_c}$ to each TPC i, where σ_c is the signature of the message under the public key PK_c of client c, $Cert_c$ is the certificate of PK_c signed by the root certificate, and $E(.)_{EK_i}$ is the encryption under the public key EK_i.

Upon receiving the registration, TPC i verifies that Pk_c corresponds to $Cert_c$, which must be signed by the root certificate CR_0. Next, it calculates a commitment z to the client's symmetric key $SymK_c$. We use a hash function (SHA-256) denoted by $H(.)$ to implement z as a pseudorandom function: $z = H(nonce||SymK_c)$. TPC i, then, signs z with its private key SK_i and sends it as a response to the client. The client waits and collects $2f + 1$ valid signed responses and broadcasts them again to the TPCs. TPCs verify that the signatures on the responses that the client collected are valid and also that the commitments match. Only then can TPCs register the client's symmetric key under $OTID_0$.

The $2f + 1$ matching responses indicate that the client registered the same symmetric key with $2f + 1$ TPCs, among which at least $f + 1$ are correct. This is crucial, as we will see in the next sections, to guarantee totality for FTOB even with faulty clients.

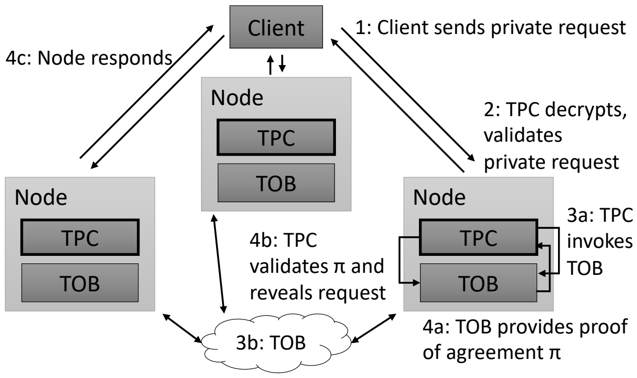
Client
1: Client sends private request
4c: Node responds
Node
TPC
TOB
2: TPC decrypts, validates private request
Node
TPC
TOB
Node
TPC
TOB
3a: TPC invokes TOB
4b: TPC validates π and reveals request
3b: TOB
4a: TOB provides proof of agreement π

Fig. 2. Overview of the four phases of Fairy protocol: (1) request submission, (2) request processing, (3) request ordering, and (4) request disclosure and delivery.

B. Fairy protocol execution

We split the protocol into four phases: (1) request submission, (2) request processing, (3) request ordering, and (4) request disclosure and delivery.

In the first phase, the client invokes FTOB with an encryption of request r under the symmetric key the client shares with TPCs. We refer to the encrypted request the client submits as *private request*, because it also hides the client's identity. In the second phase, the TPC, upon receiving the private request, decrypts it and verifies its authenticity. In the third phase, the TPC creates a new request r_{TPC} having the private client request as payload and invokes TOB for r_{TPC}. In the fourth phase, after some node i delivers r_{TPC} with proof π and sequence number sn to the TPC hosted by the same node, TPC verifies π and discloses (reveals) r to the node. Finally, the hosting node can f-deliver r with sn and respond to the client. Figure 2 summarizes the phases of Fairy protocol.

In the rest of this section we provide the protocol details of each phase. Algorithm 1 presents the client implementation of registration and FTOB invocation. Algorithm 2 presents client registration handling, request processing, and request disclosure implementation in TPC. Finally, Algorithm 3 presents the node's interface implementation to the TOB process and TPC.

1) Request submission: Client request r is represented as a message $\langle REQUEST, o, t, c, nonce\rangle$ where o is the request *payload* (operations to be executed), t is the client's *timestamp*, a monotonically increasing sequence number per client, $c = PK_c$ the client's identity and $nonce$ a random number used only once for entropy. Two requests r_1, r_2 are considered equal, and we write $r_1 = r_2$, if and only if $r_1.o = r_2.o \wedge r_1.t = r_2.t \wedge r_1.c = r_2.c$.

To invoke FTOB for a request r with timestamp t, client c first waits for $2f + 1$ responses that request $t - 1$ has been f-delivered. This, because of FTOB liveness property, guarantees that a request with timestamp $t-1$ will eventually be delivered by all correct nodes and therefore all correct nodes will be able to process the request with timestamp t. We say that the client submits requests in a close loop. The client wraps r in a private request $Pr(r, c) = AE(\langle r, OTID_{t+1}, \Pi\rangle_{SymK_c})$.

$OTID_k$ is the one-time client's ID for request with timestamp k and Π is the Merkle-tree root of the keys of the most recent configuration of TOB processes the client c is aware of. $AE(.)_{SymK_c}$ denotes authenticated encryption under the client's symmetric key. Pr implements the function F of FTOB abstraction defined in Section II. Then the client sends a message $\langle FTOB-CAST, Pr(r,c), OTID_t\rangle$ to all nodes.

2) Request processing: Upon receiving a $FTOB-CAST$ message, each correct node in $Nodes$ forwards the message to their TPC. TPC uses the $OTID_t$ field to retrieve the client's symmetric key and decrypt and authenticate the encrypted part of the message. TPC checks if a request r' of the same client with timestamp $r'.t = r.t - 1$ has already been f-delivered, otherwise it drops r.

TPC then checks if Π matches the identities of $Nodes$ it is aware of. If not, TPC drops r and notifies the client. The identities represented by Π are used in the delivery phase to validate the proof π of TOB. If Π does not match the membership in $Nodes$ the TPC knows about, either the client is not up to date, or the node has not updated their TPC.

3) Request ordering: For each client request r, each TPC i calculates PID which is a unique private identifier of request r. Then TPC i creates a new request message r_{TPC} represented by a message $\langle REQUEST, \hat{r}, t_i, PK_i\rangle_{\sigma_i}$ where $\hat{r} = \langle Pr(r, r.c), OTID, PID\rangle$, PK_i is the public key of i, σ_i a signature of the request under PK_i, and t_i is the timestamp of TPC i, a monotonically increasing sequence number per TPC, not to be confused with client's timestamp $r.t$.

The identifier PID should not reveal any information about request r but should identify r uniquely. We use a hash function (SHA-256) to implement PID as a pseudorandom function, with the nonce of request r as a secret: $PID = H(r.nonce||r.o||r.t||r.c)$.

Two requests r_1, r_2 are considered equal, and we write $r_1 = r_2$, for TOB if and only if $r_1.pid = r_2.pid$. We do not use t_i or PK_i to argue about request equality in TOB, since different TPCs can invoke TOB for the same request r.

TPC i invokes TOB for request r_{TPC} by sending a message $\langle TOB-CAST, r_{TPC}\rangle$ to the TOB process j which is hosted by the same node as i.

4) Request disclosure: Upon TOB process j is delivering request r_{TPC} with sequence number sn and proof π, j sends a message $\langle TOB-DELIVER, sn, r_{TPC}, \pi\rangle$ to the TPC k hosted by the same node as j. TPC verifies first that it has already disclosed sequence number $sn - 1$. Disclosing sequence numbers in order guarantees that TPC is up to date with the state of correct clients. TPC verifies proof π for the membership of $Nodes$ justified by Π. TPC k uses the $OTID$ in r_{TPC} to retrieve the symmetric key that decrypts the private request in r_{TPC}. TPC then asserts that the request timestamp t is the next timestamp of the previously delivered request for the same client and updates the $OTID$ that points to the client's symmetric key. The TPC reveals the decrypted request r to the node, only if π is valid. The node can now output $\langle FTOB - DELIVER, sn, r\rangle$, terminating FTOB for request r. Otherwise, the node outputs $\langle FTOB-DELIVER, sn, \perp\rangle$ and asks the client to resubmit the request.

Notice that if the system configuration has changed to a configuration with Merkle-tree root $\Pi' \neq \Pi$ between the request processing and request disclosure, TPC cannot validate π and aborts. It is important that the client sends the reference of the most recent configuration because the TPC cannot trust the node to provide it. A TPC with a non-valid view of the system configuration could be tricked into disclosing clients' requests with an invalid proof, such that the request will be eventually delivered, therefore, potentially violating the input causality property. We assume here, that a correct client is always aware of the correct system configuration and that the system configuration does not change faster than the clients learn about it. We also assume that all correct nodes have a consistent configuration.

If k cannot retrieve the symmetric key that decrypts the private request in r_{TPC} because no client has registered to k with the $otid$ in r_{TPC} or because the key the client registered with is not the same, the node hosting k has to fetch r from another node. Notice that in-order disclosure ensures that this inconsistency can only happen for the first request a client submits with a fresh symmetric key but the registration protocol ensures that there are always $f + 1$ TPCs on correct nodes that have the same symmetric key with TPC i.

VI. Fairy correctness arguments

In this section we sketch correctness arguments on Fairy satisfying FTOB properties defined in Section II. Line numbers refer to Fairy pseudocode.

Integrity (FTOB1) is achieved with client authentication. On each node, each client request, before being f-delivered, is authenticated by the TPC, using authenticated encryption in the request disclosure phase (Algorithm 2, line 39) and we trust TPC to be correct. The symmetric key used for authenticated encryption is authenticated by TPCs during registration with client's signature authentication. If TPC in node i does not have the client's symmetric key for authenticating and disclosing the request in the disclosure phase, node i will fetch the request from another node j, signed by TPC in node j, which we also trust to be correct.

Agreement (FTOB2) is inherited from the underlying TOB protocol. Namely, if a request is f-delivered with some sequence number sn, it has been delivered by TOB with the same sequence number.

No-duplication (FTOB3) is trivially achieved because of in-order disclosure of requests (see Algorithm 2, line 42), before f-delivery. Note, it is possible that TOB processes deliver the same clients request more than once wrapped in different TPC requests, since the client submits to more than one node for robustness. Let's assume that there is a request r with timestamp t that is f-delivered with sequence number sn and later r is again delivered by TOB with $sn' > sn$. No TPC will disclose the request for a second time, because they have already disclosed a request from this client with the same or a higher sequence number (see Algorithm 2, line 43).

Algorithm 1 Fairy implementation at the client C_i.

```
state
    $t$                                                    // timestamp, initially 0
    $otid_t$                                               // current one-time id
    $otid_{t+1}$                                           // next one-time id
    $k_i$                                                  // current symmetric key
    $(pk_i, sk_i)$                                         // client public/private key
    $(cert)$                                               // $pk_i$ certificate
    Nodes                                                  // the set of nodes
    TPCs                                                   // set of TPCs and their attestations
    $f$                                                    // fault tolerance
    RegResp                                                // set of registration responses
    reg                                                    // latest registration

struct Registration contains
    otid                                                   // initial one-time id
    key                                                    // symmetric key
    cert                                                   // public key certificate

struct Request contains
    o                                                      // payload
    t                                                      // timestamp
    c                                                      // client's public key
    nonce                                                  // random number used only once

function register()
    for $(\tau, attestation)$ in $TPCs$ do
        assert $verify_\tau(\tau, attestation)$            // verify TPC attestation
        $RegResp \leftarrow \emptyset$                     // initialize response set
        $ek_\tau \leftarrow attestation$                   // TPC public key
        $otid_t \leftarrow rand()$                         // initial one-time id
        $k_i \leftarrow key\text{-}gen()$                  // symmetric key
        $nonce \leftarrow rand()$
        $r \leftarrow Registration(otid_t, k_i, pk_i, nonce)$
        $er \leftarrow encrypt(r, ek_\tau)$                // public key encryption
        $\sigma \leftarrow sign(er, pk_i)$                 // clients signature
        $reg \leftarrow er$
        send [REGISTER, $er, \sigma, pk_i$] to $\tau$

upon receiving [REGISTER-RESPONSE, $resp$] do
    $RegResp \leftarrow RegResp \cup resp$

upon $|RegResp| = 2f + 1$ do
    send [REGISTER-PROOF, $reg, RegResp$] to $TPCs$

// FTOB invocation of operation $o$
function invoke($o$)
    $\Pi \leftarrow root(Nodes)$                           // Merkle-tree root
    $otid_{t+1} \leftarrow rand()$                         // next one-time id
    $nonce \leftarrow rand()$
    $r \leftarrow Request(o, t, pk_i, nonce)$
    $pr \leftarrow auth\text{-}encrypt(r, otid_{t+1}, \Pi, k_i)$   // authenticated encryption
    send [PROCESS, $pr, otid_t$] to $Nodes$

upon receiving ftob-deliver from $f + 1$ nodes in $Nodes$ do
    $t \leftarrow t + 1$                                   // update timestamp
    $otid_t \leftarrow otid_{t+1}$                         // update otid
```

Algorithm 2 Implementation of Fairy at the TPC T.

```
state
    $t$                                                    // TPC sequence number, initially 0
    $d$                                                    // last delivered sequence number
    $R$                    // registered client map $otid \rightarrow key \times timestamp$
    $(pk_T, sk_T)$                                         // TPC public/private key for signatures
    $(ek_T, dk_T)$                                         // TPC public/private key for encyption
    $f$                                                    // fault tolerance
    $C_0$                                                  // root certificate
    Nodes                                                  // the set of nodes

upon receiving [REGISTER, $er, \sigma, pk_c$] from $c$ do
    assert $verify(er, \sigma, pk_c)$                      // client signature verification
    $r \leftarrow decrypt(er, dk_T)$                       // decrypt encrypted registration
    assert $verifyCert(r.cert, C_0)$                       // verify clients certificate
    $\phi \leftarrow hash(r.nonce \| r.key)$
    $\sigma \leftarrow sign(\phi, pk_T)$
    send [REGISTER-RESPONSE, $\phi, \sigma$] to $c$

upon receiving [REGISTER-PROOF, $er, proofs$] from $c$ do
    $r \leftarrow decrypt(er, dk_T)$                       // decrypt encrypted registration
    assert $verify(r, proofs)$                             // $2f + 1$ matching proofs
    $R[r.otid] \leftarrow (r.key, 0)$                      // store client state

function process($pr, otid$)
    assert exists($R[otid]$)
    $(k_c, t_c) \leftarrow R[otid]$                        // retrieve client state
    $(r, otid_{next}, \Pi) \leftarrow auth\text{-}decrypt(pr, k_c)$   // decrypt the private request
    assert $r.t = t_c + 1$                                 // verify this is client's next request
    $PID \leftarrow hash(r.nonce \| r.o \| r.t \| r.c)$    // calculate unique identifier
    $r_{TPC} \leftarrow Request(\langle pr, otid, pid \rangle, t, pk_T, \perp)$
    $t \leftarrow t + 1$                                   // increase TPC sn
    $\sigma \leftarrow sign(r, sk_T)$
    return $(r, \sigma)$

function disclose($r_{TPC}, sn, \pi$)
    $(pr, otid, pid) \leftarrow r_{TPC}.o$   // retrieve the priv. req. and otid from the payload
    assert exists(($R[otid]$)
    $(k_c, t_c) \leftarrow R[otid]$                        // retrieve client state
    $(r, otid_{next}, \Pi) \leftarrow auth\text{-}decrypt(pr, k_c)$   // decrypt the private request
    assert $verifyRoot(\Pi, Nodes)$                        // verify Merkle tree root
    assert $verifyProof(req, sn, \pi, Nodes)$              // verify proof of external validity
    assert sn = d + 1                                      // assert in order disclosure
    assert $r.t = t_c + 1$                                 // verify this is client's next request
    $R[otid_{next}] \leftarrow (k_c, r.t)$                 // store updated client state
    $R[otid] \leftarrow \perp$                             // remove old client state
    $d \leftarrow sn$                                      // update protocol state
    return $r$
```

Algorithm 3 Implementation of Fairy at the nodes.

```
upon receiving [FTOB-CAST, $pr, otid$] do
    $(r_{TPC}, \sigma) \leftarrow process(pr, otid)$
    invoke TOB for [REQUEST, $r_{TPC}, \sigma$]

function deliver($r_{TPC}, sn, \pi$)
    $r \leftarrow disclose(r_{TPC}, sn, \pi)$
    send [FTOB-DELIVER, $sn, r$] to client with public key $r.c$
```

Totality (FTOB4) is achieved by TOB Totality and the guarantee that there always exist at least $2f + 1$ TPCs, among which at least $f + 1$ on correct nodes that have the symmetric key to decrypt the delivered request. See also Section V-A3.

Liveness (FTOB5) is guaranteed by a correct client submitting to all nodes. Among all nodes there exists at least $2f + 1$ who have delivered the client's previous request from the close-loop submission (if not they ask the client to resubmit), and among them at least $f + 1$ correct to invoke TOB. By TOB liveness, the request will eventually be delivered and by the client being correct, all on correct nodes can disclose the request.

Input Causality (FTOB6), in short, is achieved by hiding a client request until it has proof from TOB that it can be delivered by all correct nodes with some sequence number sn. Notably, we use symmetric AES-GCM authenticated encryption to hide the request. Even though AES is malleable, the authenticator guarantees that no tampered request can be valid and therefore accepted by TPCs.

Sender Obfuscation (FTOB7), is achieved by hiding the client's identity inside the encrypted private request. The one-time id and the unique request's identifier, the only two fields that are sent in plaintext, are pseudorandom, leaking nothing about the client's identity.

VII. Fairy implementation

A. High throughput Fairy implementation with Mir-BFT

Mir-BFT [35], or simply Mir, is a novel multi-leader Byzantine fault-tolerant TOB protocol which achieves high throughput in large deployments (evaluated up to 100 nodes) both in datacenters and WANs. Mir runs in parallel instances of the classical PBFT protocol [13] by sharding sequence numbers deterministically across multiple leaders, while multiplexing them in a single total order. Leveraging multiple leaders, Mir alleviates the network bottleneck which drives the performance of single leader protocols. Moreover, Mir avoids ordering duplicate requests by deterministically sharding requests in buckets that are uniformly distributed across all leaders, since eliminating duplicate requests before ordering proves critical for maintaining high performance for multileader protocols. The protocol maintains the liveness guarantees of TOB by periodically rotating the bucket assignment.

Mir assumes asynchronous communication for safety and eventual synchrony for liveness with optimal fault tolerance; a system with n nodes tolerates up to f faults s.t. $n \geq 3f+1$. Fairy implementation on top of Mir inherits the same fault-tolerance under the same synchrony assumptions.

We choose to implement Fairy on top of Mir because of its good performance. Our modular design allows us to easily implement Fairy on top of Mir-BFT codebase with minor modifications. In detail, we extend the Mir client by implementing Fairy client registration, wrapping Mir requests in the private FTOB client request, and enforcing close-loop request submission. We also implement our own back-end for Mir, which forwards client registrations and requests to the local TPC, invokes Mir with TPC requests wrapped in Mir client requests, forwards Mir delivered batches to TPC for the Fairy disclosure phase, and, finally, responds to Fairy client. Both, our client and back-end implementation are in Golang, same as Mir, but both TOB and FTOB interfaces expect messages serialized as Protocol buffers [1], which allows flexibility in the language of the implementation.

External Validity. To enhance Mir with the *External Validity* (TOB6) property from Section II, we extend Mir implementation so that each process signs the message of the commit phase. Mir liveness invariant, same as PBFT, is that if a correct process commits, and therefore delivers a request, eventually all correct processes will deliver the request. Collecting $f+1$ signatures as proof π guarantees that at least one of them is from a correct process, which in turn guarantees that the request will eventually be delivered with sequence number sn by all correct processes.

Duplication Prevention. To achieve a highly performant FTOB, we need to limit the impact of duplicate requests, since, as is demonstrated in [35], for multi-leader protocols duplication prevention is critical for performance. Mir eliminates ordering of duplicate requests; however, this translates to eliminating duplicate TPC requests, as TPCs invoke Mir as its clients. Since a Fairy client submits a private request to multiple TPCs for robustness, different TPCs may invoke Mir with their requests for the same client private request, which from Mir's point of view are different, since each TPC has its own public key and is viewed as a different client.

Duplicate Fairy requests are eliminated in disclosure phase. To maintain, though, the high throughput of Mir we eliminate duplicate Fairy requests also during ordering. To achieve this we modify the Mir request equality predicate for filtering duplicate requests as described in Section V-B3. Two requests r_1, r_2 are considered equal, if and only if $r_1.pid = r_2.pid$. We must point out that this equality predicate can be effectively used by Mir to prevent ordering of duplicate requests only with infinite memory, by keeping all request identifiers that have ever been proposed. Mir-BFT, as any practical protocol, periodically garbage-collects its state with a Checkpoint protocol. While we can trust that a TPC will not invoke Mir-BFT for a Fairy request that is already known to be delivered, a malicious node can withhold the delivery of a request from their TPC, or can reset the TPC forcing it to lose its state. This can trick the TPC into submitting the same request multiple times and if this occurs in different Checkpoint periods it will result in duplication. Section VII-B discusses how to avoid such attacks on the TPC state. Notice that this attack can introduce only one duplicate for each request within each Checkpoint period. More importantly, this performance attack is detectable, since, after revealing the content of the Fairy request, the correct processes can identify the duplicate requests proposed from Byzantine processes that are delivered out of order. It is also possible for a malicious client to submit the same request with a different $nonce$ value for each TPC, creating therefore different private identifiers for the same request. However, this performance attack is also detectable post ordering and malicious clients can be blacklisted.

Batching. Mir uses batching to amortize the cost of protocol operations per request, a common performance optimization technique. A leader groups incoming requests in a batch and orders the batch with the same sequence number. A deterministic order of the requests within the batch (e.g., lexicographical) results in a unique total order of requests. Since Fairy builds on top, it inherits the batching mechanism and maintains the requests' batching for the disclosure phase. This further helps Fairy amortize the cost of signature verification of the disclosure phase.

B. TPC implementation with Intel SGX

The trusted execution environment (TEE) used in this work is Intel's Software Guard Extensions (SGX) [28], though the general concept can be applied to other TEEs as well. Intel SGX is an instruction set extension of x86 processors which allows the creation of trusted compartments called *enclaves*. Intel SGX ensures the confidentiality of code and data via transparent memory encryption, i.e., code and data are only available in plaintext in the CPU package, while checksums are used for integrity checks. Enclaves can only be entered or exited using pre-defined entry points via *ecalls* into the enclave and *ocalls* to the outside. All other access, even from privileged software, is prevented, as only the code in the enclave is assumed to be trusted. Enclaves can run arbitrary code with the exception of system calls, which require input from the untrusted OS. There exist several SDKs to facilitate the enclave life-cycle management and memory management, and both C/C++ and Rust are supported programming languages. Fairy TPC implementation is written in Rust.

The amount of memory available to all enclaves in a system is 128 MB (92 MB in practice). As we discuss in Section V TPCs persist per client state to map client's one-time identifier (OTID) to their symmetric key and latest request sequence number. This imposes a limit on the number of clients the TPC can serve. For example, assuming both fields have a 32 bit representation, the enclave can serve up to 11.5 million clients. To server more clients, TPCs can garbage collect state pertaining to stale clients and force them to register again with a fresh symmetric key.

TEE technology, such as Intel SGX, helps to protect confidentiality and integrity; still, some attack vectors remain, and therefore even this technology must be used cautiously. In particular, Intel SGX is known to suffer from rollback and forking attacks on stateful applications with persistent storage [8], [27]. The key to a rollback attack is that when the enclave is stopped, the data inside is lost. Data can be persistently stored for later use by encrypting and storing them in memory outside of the enclave, a process called sealing. However, a malicious node may force the enclave to restart at any time and load its memory from some state that is not the most recent one. Forking attacks generalize rollback attacks in that the malicious node may spawn multiple enclave instances from the same sealed state and split the clients who interact with the enclave to interact with different instances.

While rollback and forking attacks seem relevant to Fairy, since TPCs store protocol and client state, Fairy is robust to such attacks. If TPCs in Fairy crash or are forced to restart, they do not need to use the locally persisted state to recover. Instead, client-related state can be re-established by asking the clients to re-register with a fresh symmetric key. Notice that FTOB should guarantee liveness and input causality only to correct clients, who provide their correct, most up-to-date state with their registration. Protocol state, i.e., the latest delivered sequence number, is relevant for updating the client state in the right order without losing liveness. However, Byzantine nodes are not expected to provide their enclaves with the correct protocol state. To support a crash fault recovery, protocol state can be recovered with a distributed storage protocol [27].

Other known attacks are denial-of-service (DoS) and side channel attacks [9] [34] [36]. The latter and their mitigation are orthogonal and outside the focus of this work. DoS is captured in our extended model (Section IV) by the adversary controlling the communication to and from the TPCs. This attack would only render the TPC non-responsive would only threaten the liveness of Fairy protocol if the number of faulty nodes exceeds f, i.e., the fault tolerance of our system.

VIII. Performance evaluation

In this section, we evaluate the performance of Fairy implementation with Mir, further referred to as *Fair-Mir*. In particular, we examine the overhead of Fairy by comparing Fair-Mir to plain Mir protocol implementation. We also compare Fair-Mir with the state-of-the-art non-malleable commit-reveal scheme [17], also implemented on top of Mir for a fair comparison, further referred to as CR-Mir. We do not compare Fair-Mir to threshold encryption protocols in [11] [16], since the evaluation in [17] shows that the commit-reveal scheme clearly outperforms threshold encryption. We evaluate all three protocols in a wide area network (WAN) of 16 client machines and up to 16 node machines and in a local area network (LAN) of 4 client and 4 node machines. Finally, we perform LAN micro-benchmarks to stress the performance of TPC under different parameters. Table I summarizes the used configuration parameters for our performance evaluation.

Max batch size	50 KB (100 requests)
Cut batch timeout	10 ms
Checkpoint period	16
Watermark window size	64
Bucket rotation period	64
Client authenticated Encryption	AES-GCM 256
TPC & batch signatures	256-bit ECDSA

TABLE I. Fairy configuration parameters used in evaluation.

LAN deployment We executed our evaluation in a local area network of 4 node and 4 client machines. Nodes run on Supermicro 5019-MR servers with a 3.4GHz 4-core E3-1230 V5 Intel CPU that provides SGX support. They are equipped with 32 GB of memory and run Ubuntu Linux 20.04 LTS Server. Client processes run on Haswell 2.4GHz 4-core, 8GB memory machines with Ubuntu Linux (kernel version 4.4.0-24). All machines have a 1 Gbps network connection. We use docker for seamless deployment across all servers.

WAN deployment For our wide area network evaluation we deployed 16 VMs for the clients and up to 16 VMs for the nodes. They all run on Xeon 2.4GHz 4-core, 32 GB machines with Ubuntu Linux (kernel version 4.15.0-118) and 1 Gbps network connection. All virtual machines are deployed on IBM Cloud, distributed across 16 distinct datacenters all over the world. In this deployment we ran our enclaves with SGX simulation mode as HW mode was not yet available. In a preliminary test we experience similar performance between hardware and software mode for our workload. Same as for the LAN deployment, we use docker for seamless deployment across all servers.

End-to-end evaluation. Clients submit requests in a close loop, that is, they send their requests as soon as they are notified by $2f+1$ nodes that their previous request is delivered. Request payload size is fixed to 500 bytes, which corresponds to the average Bitcoin transaction size [3]. With the term end-to-end latency, we refer to the latency form the moment the clients send the request to the network until they are notified by $2f+1$ nodes that the request is delivered. We do not include in end-to-end latency the latency of request preparation in the client, as this was done asynchronously to allow clients to submit requests fast enough to saturate the nodes.

We first evaluate latency in the WAN for an increasing number of nodes. When deployed on all 16 nodes Fair-Mir has an overhead of 4% compared to plain Mir. In particular, the latency of the additional phases of Fair-Mir, processing and disclosure, accounts in total only for 0.2% of end-to-end latency. On the other hand CR-Mir has more than double the latency of the other two protocols, since it requires extra communication rounds. Latency profiling for the WAN evaluation can be found in Table II.

In order to minimize the network latency and better observe the overhead of Fairy we focus on a LAN evaluation of 4 nodes for the rest of the section.

Figure 3 shows the average end-to-end latency and average throughput of Mir, Fair-Mir, and CR-Mir. We evaluate all protocols by increasing request load until they reach saturation. Load is increased by increasing the number of client instances per client process and the number of client processes. We observe that Fair-Mir reaches a maximum throughput of $7.3K$ req/s, which suggests an overhead of approximately 20% on Mir which achieves a peak throughput of $9.3K$ req/s, without significant overhead in latency. Latency profiling of Fair-Mir can be found in Table III. It is worth noticing that the latency of Fairy request processing and request disclosure phases accounts for only 2% of the end-to-end latency.

Fair-Mir outperforms by 66% CR-Mir in throughput, with the latter only reaching $4.4K$ req/s. Moreover, CR-Mir has double the latency of Mir and Fair-Mir. This is expected, since CR-Mir runs two rounds of TOB, unlike Fair-Mir and Mir that run only one TOB round.

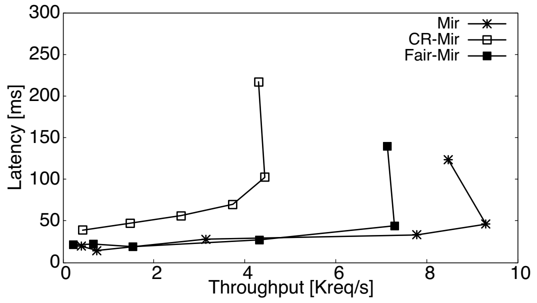

Fig. 3. Impact of Fairy on Mir and comparison to the commit-reveal (CR-Mir) protocol [17].

Impact of TPC request processing. We micro-benchmark the request processing phase (Section V-B2) to estimate when it becomes a bottleneck to Fair-Mir. In particular, for different payload sizes, we invoke request processing with increasing parallelism. As we can see in Figure 5 and Figure 4, increasing payload sizes has a negative impact on the performance of the request processing phase because the size of data that are copied in the enclave per operation grows. Up to 8 parallel invocations increase overall throughput. Higher parallelism does not help, since we need to lock access to the client state inside the TPC, which causes contention. To put that into perspective, we co-plot in Figure 5 the peak throughput in a LAN with 4 nodes. As we can see, even for larger payloads (tens of KB) TPC can process thousands r/s, and, with the selected payload size (500 B), Fairy does not suffer any performance limitations due to request processing.

Scalability micro-benchmark. Finally, to estimate the performance of Fair-Mir in larger deployments we isolate and micro-benchmark the request disclosure phase. Our evaluation focuses on the added complexity of Fairy, which is affected by the system scale only in the disclosure phase (Section V-B4) during which each TPC needs to verify $O(n)$ signatures per batch. In Figure 6 we plot the peak throughput request disclosure can achieve with an increasing number of simulated nodes, for different batch sizes. The number of nodes is simulated by increasing the signatures per batch. Indeed, we observe that throughput drops with an increasing number of nodes, but this is amortized with increasing batch size. To put this into perspective, we also plot Mir performance on a LAN of 32-core VMs with increasing number of nodes. We can see that, with similar batch sizes, Fairy disclosure phase imposes a limit of 57.3K req/s on 97 simulated nodes, which is approximately 70% of Mir throughput on the same scale.

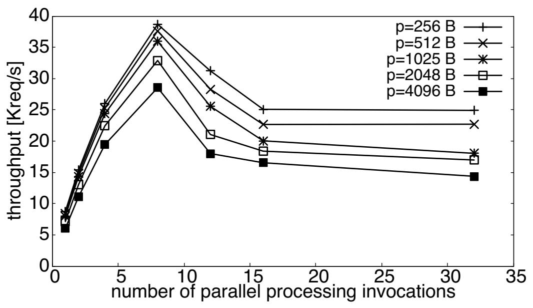

Fig. 4. Microbenchmarking of private request processing and validation for increasing parallel processing invocations with different payload sizes (p).

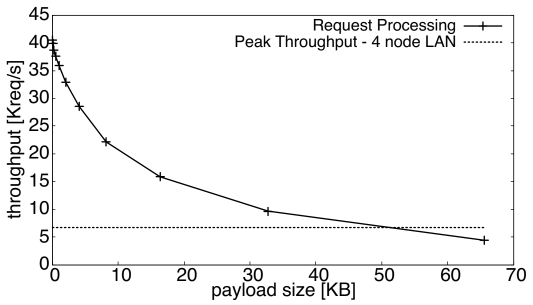

Fig. 5. Microbenchmarking of private request processing and validation for increasing payload size with 8 parallel processing invocations.

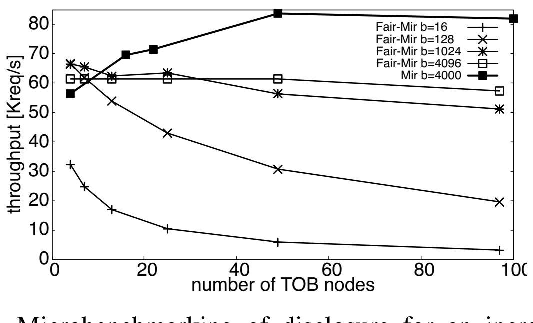

Fig. 6. Microbenchmarking of disclosure for an increasing number of TOB nodes with different batch sizes (b).

Protocol	Processing	Ordering	Disclosure	End-to-End	Nodes
Mir	-	353.23 ms	-	1411.56 ms	4
Fair-Mir	0.75 ms	354.63 ms	1.29 ms	1479.17 ms	4
CR-Mir	-	367.40 ms	-	2985.65 ms	4
Mir	-	235.04 ms	-	935.16 ms	7
Fair-Mir	0.81 ms	237.88 ms	1.42 ms	1023.46 ms	7
CR-Mir	-	245.54 ms	-	1882.59 ms	7
Mir	-	375.84 ms	-	1602.47 ms	16
Fair-Mir	0.70 ms	354.79 ms	1.91 ms	1669.47 ms	16
CR-Mir	-	347.11 ms	-	3776.67 ms	16

TABLE II. Latency profiling with 4, 7 and, 16 nodes in WAN for Fair-Mir with low load, below saturation.

	mean	95p	99p	pseudocode
Preparation	0.1 ms	0.1 ms	0.3 ms	Alg1 lines[46:52]
Processing	0.3 ms	0.6 ms	0.8 ms	Alg2 lines[24:33]
Ordering	15.4 ms	33.3 ms	52.8 ms	
Disclosure	1.0 ms	1.8 ms	2.0 ms	Alg2 lines[35:47]
End-to-End	19.4 ms	40.9 ms	51.7 ms	

TABLE III. Latency profiling with 4 nodes in LAN for Fair-Mir with 50% saturation load. Line numbers refer to Fairy pseudocode in Appendix VI.

IX. Related Work

Causality under the crash fault-tolerance assumption was introduced by Lamport [23]. Under Byzantine faults, causality was introduced by Reiter and Birman [32], who outline a protocol for causal total order broadcast in BFT systems by combining total order broadcast with a public key threshold cryptosystem, such that a client submits an encrypted request, which is only revealed by combining decryption shares in a round of communication after total order broadcast of the encrypted request. Later Cachin *et al.* [11] refined the definition of causal total order broadcast and provided a provably secure protocol based on labeled threshold cryptosystem with a two-step delivery. The protocol was implemented in SINTRA system [12]. More recently, Duan *et al.* proposed BEAT [16], a group of practical asynchronous TOB protocols, including one protocol which guarantees causality also using threshold encryption. Solutions based on threshold cryptosystems, however, require extra rounds of communication to combine the decryption shares. Fairy does not require any further communication rounds after total order broadcast. Moreover, threshold cryptosystems are computationally expensive and require a trusted setup of the threshold key shares. Such a setup requires either a trusted third party, compromising decentralization, or a distributed key-generation (DKG) protocol. The most efficient asynchronous DKG protocol to date, to the best of our knowledge, was introduced by Kokoris-Kogias *et. al* [22] with $O(n^3)$ communication and $O(n^4)$ bit complexity.

Causal total order broadcast is revisited by Duan *et al.* [17] who propose a group of protocols which replace threshold encryption with symmetric cryptography improving efficiency over [11]. However, two of the protocols in [17] only guarantee safety and liveness with benign, crash-fault clients and the third needs strong synchrony assumptions to maintain liveness with Byzantine clients. Fairy tolerates an arbitrary number of Byzantine clients, making it suitable for a broader set of applications, without imposing further synchrony assumptions to the underlying TOB protocol.

Sender obfuscation and anonymity have been extensively studied in the space of cryptocurrencies with a number of techniques such as zero-knowledge proofs [29], ring signatures [31], and decentralized mix services [33]. Sender obfuscation has not been addressed, however, so far in the context of causal total order broadcast.

Kelkar *et al.*, in their recent work [21], add fairness to ordering in a different context. Their protocol maintains the actual receiving order for a fraction of nodes in the output of total order broadcast, within a block of arriving requests. The protocol pays for this by requiring at least $4f + 1$ nodes to tolerate f faults. Similarly, Zhang *et al.* [37], maintain in the output the ordering preferences of correct nodes. Notably, though, neither of the two protocols can prevent front-running attacks when the network is controlled by the adversary.

Trusted execution environments have been explored in BFT systems for improving performance and fault-tolerance [5], [20], [25] in a hybrid BFT/TEE model. Note that our model does not mandate that a TOB process itself encapsulates a TEE. Our system is built in a modular way and allows the use of other TOB protocols including crash fault-tolerant and, as evaluated in this paper, classical BFT TOB protocols.

Trusted execution environments against front-running attacks have been introduced in Tesseract [6] in the context of cryptocurrency exchange, by running the exchange protocol confidentiality in a TEE. Similarly to Fairy, Tesseract uses authenticated encryption to establish a confidential communication between the TEE and the client, obfuscating also the client's identify. However, Tesseract is tailored to cryptocurrency exchange and does not implement TOB.

Ekiden [14] and Fabric Private Chaincode (FPC) [7] are follow-up works which use TEEs to confidentiality execute arbitrary smart contracts on dedicated nodes. Both Ekiden and FPC do not reveal the transaction itself, which is executed within the TEE, but, unlike Fairy, the do not fully implement TOB, since they only order the encrypted output of execution. Moreover, in FPC the client's identity is not hidden and Ekiden leaks transaction metadata by revealing the identity of the smart contract to the blockchain nodes. This is necessary so that consensus nodes can perform read write checks for each smart contract. Ekiden guarantees liveness via an interactive protocol, where the client needs to acknowledge that the TEE send them the encrypted transaction output. Fairy, on the other hand, does not rely on further interaction with the client after the client has submitted their request. It is worth noticing that Ekiden targets a stronger adversary which can compromise a subset of TEEs. In order to guarantee forward secrecy, a separate key committee runs a distributed key generation protocol to generate a fresh key for each transaction. This sub-protocol is orthogonal to Fairy.

X. Conclusion

This work has presented Fairy, a modular and efficient system that extends total order broadcast protocols with fairness properties, particularly, input causality and sender obfuscation. To achieve this, we proposed a system that leverages modern trusted execution technology to protect clients' requests and only reveals the contents once a request has been delivered to the nodes of the network. We showed the feasibility of our approach that uses Intel SGX and overcomes the limitations of existing solutions. Fairy evaluation on top of Mir-BFT has shown that the overhead of our approach is 20% throughput and, thus, opens a new class of applications for BFT protocols.

References

[1] Protocol buffers. https://developers.google.com/protocol-buffers.

[2] The Tor project. https://www.torproject.org/.

[3] Bitcoin visuals: Transaction sizes. https://bitcoinvisuals.com/chain-tx-size, 2019.

[4] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, and J. Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the 13th EuroSys Conference, 2018*.

[5] J. Behl, T. Distler, and R. Kapitza. Hybrids on steroids: Sgx-based high performance BFT. In *Proceedings of the 12th EuroSys Conference, 2017*.

[6] I. Bentov, Y. Ji, F. Zhang, L. Breidenbach, P. Daian, and A. Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019*.

[7] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti. Trusted computing meets blockchain: Rollback attacks and a solution for hyperledger fabric. In *38th Symposium on Reliable Distributed Systems (SRDS)*, 2019.

[8] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017*.

[9] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software grand exposure: SGX cache attacks are practical. In W. Enck and C. Mulliner, editors, *11th USENIX Workshop on Offensive Technologies, WOOT 2017*.

[10] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[11] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. *IACR Cryptol. ePrint Arch.*, 2001.

[12] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the internet. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks DSN 2002*.

[13] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, (4), Nov. 2002.

[14] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[15] D. Dolev, C. Dwork, and M. Naor. Nonmalleable cryptography. *SIAM Rev.*, 45(4).

[16] S. Duan, M. K. Reiter, and H. Zhang. BEAT: asynchronous BFT made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018*.

[17] S. Duan, M. K. Reiter, and H. Zhang. Secure causal atomic broadcast, revisited. In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017*.

[18] S. Eskandari, S. Moosavi, and J. Clark. Sok: Transparent dishonesty: front-running attacks on blockchain. In *Proceedings of the International Conference on Financial Cryptography and Data Security*, 2019.

[19] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. 1987.

[20] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: Resource-efficient Byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys 2012*.

[21] M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels. Order-fairness for byzantine consensus. In *Annual International Cryptology Conference*. Springer, 2020.

[22] E. Kokoris-Kogias, D. Malkhi, and A. Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In J. Ligatti, X. Ou, J. Katz, and G. Vigna, editors, *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, (7), July 1978.

[24] L. Lamport, R. E. Shostak, and M. C. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, (3), 1982.

[25] J. Liu, W. Li, G. O. Karame, and N. Asokan. Scalable byzantine consensus via hardware-assisted secret sharing. *IEEE Transactions on Computers*, 68(1), 2018.

[26] J. W. Markham. Front-running-insider trading under the commodity exchange act. *Cath. UL Rev.*, 38, 1988.

[27] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium*, 2017.

[28] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In R. B. Lee and W. Shi, editors, *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[29] I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *2013 IEEE Symposium on Security and Privacy*.

[30] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[31] S. Noether. Ring signature confidential transactions for monero. *IACR Cryptol. ePrint Arch.*, 2015.

[32] M. K. Reiter and K. P. Birman. How to securely replicate services. *ACM Trans. Program. Lang. Syst.*, 16(3), 1994.

[33] T. Ruffing, P. Moreno-Sanchez, and A. Kate. Coinshuffle: Practical decentralized coin mixing for bitcoin. In *European Symposium on Research in Computer Security*, 2014.

[34] M. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: eradicating controlled-channel attacks against enclave programs. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017*.

[35] C. Stathakopoulou, T. David, and M. Vukolic. Mir-bft: High-throughput BFT for blockchains. *CoRR*, abs/1906.05552.

[36] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015*.

[37] Y. Zhang, S. Setty, Q. Chen, L. Zhou, and L. Alvisi. Byzantine ordered consensus without byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.