

Adversary-Augmented Simulation for Fairness Evaluation and Defense in Hyperledger Fabric

Erwan Mahe , Rouwaida Abdallah , Sara Tucci-Piergiovanni 
Université Paris Saclay, CEA LIST
Palaiseau, France

Pierre-Yves Piriou 
EDF Lab, Dpt. PRISME
Palaiseau, France

Abstract—This paper presents an adversary model and a simulation framework specifically tailored for analyzing attacks on distributed systems composed of multiple distributed protocols, with a focus on assessing the security of blockchain networks. Our model classifies and constrains adversarial actions based on the assumptions of the target protocols—defined by failure models, communication models, and the fault tolerance thresholds of Byzantine Fault Tolerant (BFT) protocols. The goal is to study not only the intended effects of adversarial strategies but also their unintended side effects on critical system properties. We apply this framework to analyze fairness properties in a Hyperledger Fabric (HF) blockchain network. Our focus is on novel fairness attacks that involve coordinated adversarial actions across various HF services. Simulations show that even a constrained adversary can violate fairness with respect to specific clients (*client fairness*) and impact related guarantees (*order fairness*), which relate the reception order of transactions to their final order in the blockchain. This paper significantly extends our previous work by introducing and evaluating a mitigation mechanism specifically designed to counter transaction reordering attacks. We implement and integrate this defense into our simulation environment, demonstrating its effectiveness under diverse conditions.

Index Terms—Adversary model, Distributed Systems, Cybersecurity, Multi-Agent Simulation, Hyperledger Fabric, Tendermint, Order Fairness,

I. INTRODUCTION

Distributed Systems (DS), by virtue of their decentralized nature, complexity and scale, present a unique set of security challenges. While decentralization might favor fault tolerance, it also introduces vulnerabilities. Indeed, in addition of providing a greater surface of attack, most DS require specific global properties to hold (e.g., for blockchains a common property is the coherence of the replicated state, expressed as the absence of forks). Each sub-system, each connection linking them, and even properties of communication protocols in use are potential targets for malicious entities. How then can we ensure the security of DS?

Cybersecurity often involves perimeter-based defense [1], ensuring that external threats are kept at bay. However, with DS, where there might not always be a clear “inside” or “outside”, these approaches might fall short. The alternative, which we pursue, is that of modeling the adversary as an agent that is an integral part of the DS. Adversary modeling [2] has been initially introduced to reason about cryptographic protocols [3] but has since been extended to various fields in computer science and security research [4]. The use of

adversary models can facilitate the evaluation of security properties. Indeed, like any other model, an adversary model, provided it has a well-defined semantics, can be used in formal verification (e.g., model checking) or in testing (e.g., via simulation).

In this paper, we propose an adversary model that builds upon the notions of assumptions, goals, and capabilities, as introduced in [4], while being specifically tailored to assess the security of distributed systems at the protocol level. Importantly, our model targets attacks on distributed systems that combine multiple distributed protocols.

In this context, adversary goals capture the attacker’s intent, while capabilities define the set of actions available to achieve those goals. Our focus is on adversaries aiming to compromise classical properties of distributed protocols, such as safety and liveness [5], [6]. Adversary capabilities are modeled as atomic actions that can be executed at any time during system execution. These capabilities are bounded by a set of assumptions reflecting the adversary’s environment and resources. In contrast to the all-powerful network adversaries commonly used in cryptographic protocol verification [3], our model considers realistic, resource-limited adversaries constrained by the assumptions of the distributed protocols under study. These constraints stem from the underlying communication model [7], [8], failure model [9], and fault tolerance thresholds, as explored in the resource-limited adversary framework [10].

In this paper, we apply our model to demonstrate the feasibility of attacks on a distributed network of blockchain nodes implementing HyperLedger Fabric (HF) [11]. Our choice of HF is motivated by two factors. On one hand, HF stands out as a popular choice for industrial blockchain applications. On the other hand, its structure that combines two services (an endorsing service for transaction validation and an ordering service for sequencing and appending transactions to the blockchain) enables us to demonstrate our approach by combining attacks on the different protocols implementing these services in a non-trivial manner.

Let us note that although HF is a permissioned blockchain, which means that participants must be authorized to be part of the system, HF nodes can be deployed in a wide-area network (e.g., the Internet) as opposed to a private intranet. As a result, HF is vulnerable to attacks [12], [13], [14] that can consist in either or both the adversary taking control of some of its constituting nodes, or the adversary otherwise manipulating

exchanges between these nodes (e.g., increasing transmission delays via e.g., having control over routers, or via performing Denial of Service [12]).

In terms of attacks, we focus on a specific class of liveness attacks known as fairness attacks. Fairness refers to ensuring that all users, or clients, have equally fair access to the blockchain, meaning that no client’s transactions are systematically favored over others. In consortium blockchains, censorship attacks targeting specific participants are a genuine concern. As such, evaluating the robustness of distributed protocols against fairness attacks is a critical aspect of ensuring system security. More specifically, we examine four fairness properties: an application-specific form of client fairness, whose violation is the adversary’s primary goal, and three types of order fairness [6], [15], which relate the order in which transactions are received by individual nodes to the order in which they are ultimately committed to the blockchain.

In this paper, we show that even when the adversary operates within the tolerance thresholds of the underlying protocols — such as the proportion of compromised participants and assumptions about the communication and failure models — it is still possible to violate the client-fairness property in HF. Moreover, the different strategies used to achieve this violation have varying impacts on the associated order-fairness properties.

In [16], we introduced our adversary model and implemented it within the MAX [17] multi-agent simulation tool. We then defined several attacks on HF and used our simulator to demonstrate that even a weak adversary can negatively impact the fairness of a system based on HF.

Our framework also enabled the quantification of order-fairness violations, allowing us to assess the effects of our attacks on both the ordering and endorsing services of HF. To the best of our knowledge, this was the first time a blockchain simulator was augmented with a programmatic adversary [18], that these specific attacks on HF were described, and that an empirical evaluation of order-fairness violations was explored.

This paper extends our previous work [16] in five key ways: **(1)** we broaden the discussion of related work, particularly regarding order fairness; **(2)** we enhance our initial simulation framework by incorporating more realistic network assumptions—especially in terms of probabilistic delay distributions—and by collecting and analyzing additional metrics; **(3)** we introduce a mitigation mechanism aimed at improving HF’s resilience to transaction reordering attacks; **(4)** we observe and evaluate the effects of a specific implementation of this mitigation mechanism within our simulations; and **(5)** we vary additional simulation parameters—such as the number of peers, orderers, and clients—to demonstrate the generalizability of our findings.

This paper is organized as follows. Sec.II introduce preliminary notions and discusses related works. In Sec.III, we recall in details the adversary model from [16]. Sec.IV presents the HF system on which we apply our approach as well as the specific fairness properties that we consider. The attacks that can be performed by the adversary are defined in Sec.V,

and simulated in Sec.VI. In Sec.VII, we define our mitigation mechanism, and then observe its impact in Sec.VIII-A. Finally, after studying the impact of the network composition in Sec.VIII-B, we conclude in Sec.IX.

II. PRELIMINARIES AND RELATED WORKS

A. Communication and failure models

Distributed protocols specify patterns of *communications* between distant systems with the aim of *performing* a service. These services are characterized by *properties* often related to *safety* and *liveness* [5]. Here, *communications* involve message passing between sub-systems of a Distributed System (DS) over a network. Three distinct *communication models* [8] define assumptions that hold over message passing. In the *synchronous model* [7], there is a finite time bound Δ (that can be known by the involved parties) s.t., if a message is sent at time t , it must be received before $t + \Delta$. By contrast, the *asynchronous model* [7] allows an arbitrary delay between emission and reception. With the *eventually synchronous model* [8], communications are initially asynchronous, but there is an unknown Global Stabilization Time (GST) after which they become synchronous. Another equivalent definition of the *eventually synchronous model* (without GST) is to consider that the bound Δ is not known by any of the involved parties [8], [19].

Distributed protocols are deployed in an environment consisting of a DS with various sub-systems, each corresponding to a running process. The individual failure of such processes may negatively impact the service performed by the protocol (i.e., the associated properties may not be upheld). *Failure models* [9] (see Fig.1) define assumptions on the types of failures that may occur.

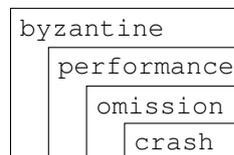


Fig. 1: Failures

A *crash* failure involves a process terminating prematurely. An *omission* failure occurs when it never delivers an event (e.g., never sends a message it is expected to send). As illustrated on Fig.1, a crash is a specific omission where, after a certain time, all subsequent events are never delivered.

With the *performance* failure model, only expected events (considering the protocols that nodes implement) occur, but the time of their occurrence may be overdue. Omission failures are infinitely late performance failures. Finally, *Byzantine* failures authorize any arbitrary behavior.

Some distributed protocols are designed to uphold specific properties, even in cases where a number of processes (sub-systems) fail. These *Fault Tolerant* (FT) protocols [9] are characterized by the nature of the failures they can withstand (i.e., a failure model) and a threshold (usually a proportion of involved processes) of failures below which they maintain their properties. For instance, Tendermint [20] is a Byzantine FT (BFT) consensus algorithm (the property it upholds being that of maintaining consensus agreement) that can withstand lessup to one-third of faulty processes.

B. Ledgers & Transaction Reordering

A Distributed Ledger is a network of replicated state machines that maintain a consistent state. By its decentralized nature, it has no single point of control/failure, which improves security. The replication of the data it carries also improves transparency and auditability. Blockchains are a means to implement a Distributed Ledger. In a Blockchain, individual machines are referred to as “nodes” and changes in the state machine occur when a transaction is “delivered”. We call “clients” the entities that submit these transactions to the nodes. By agreeing on the initial state and the order in which transactions are delivered, the nodes maintain a consensus on the current state. To improve throughput (i.e., the rate at which transactions can be delivered), transactions are batched into consecutive blocks. The order of delivery of transactions then corresponds to the order in which they appear on and within blocks.

The properties of distributed ledgers which are mostly studied can be categorized as either “*consistency properties*”, “*liveness properties*” or “*fairness properties*”. While the first may correspond to consensus agreement and validity which are safety properties [5], the second may refer to various distinct notions such as starvation-freedom, local progress or consensus wait-freedom [21]. Fairness, in the context of Blockchains [22], [23] may refer to various notions, including the fairness in committee selection [22], rewarding [24] or the ability to take decisions [23], [25] w.r.t. individual nodes’ voting power. Consistency (1), liveness (2) and the aforementioned fairness properties (in e.g., committee selection or rewards) (3) may resp. enforce that (1) all nodes deliver the exact same transactions in the same order, (2) it is always possible for an honest client to submit a transactions that will eventually be delivered or (3) the rules for nodes to participate in the consensus process are fair and they are rewarded according to their participation.

However, as explained in [6], these properties enforce no constraint on the agreed upon order of transactions. In that case, even though the involved algorithms may be BFT, an adversary may still be able to manipulate the order of transactions. A malicious adversary which causes the order of transactions to change (i.e., the consensus yields a different total order as the one which would have occurred without the intervention of the adversary) is said to perform a “transaction reordering attack” [26]. A typical example is that of front-running : if the adversary is aware that a transaction x has been submitted but not yet delivered, it may submit a new transaction x' and make so that x' is delivered before x (e.g., by leveraging reward mechanisms [27] or by coordinating Byzantine nodes that it controls [16]). In decentralized finance [27], front-running can be leveraged to extract profits via manipulating the value of financial assets. Maximum Extractable Value (MEV) bots can continuously monitor a distributed ledger and perform such attacks opportunistically [27], [26]. It is estimated that MEV bots have extracted around ~ 675 million \$ on the Ethereum blockchain alone in the span between January 2020

and September 2022, which underlines the importance of being concerned with transaction ordering.

C. Order Fairness

The rise in potential applications of transaction reordering attacks lead to an increased interest in properties related to the fairness with which the order of transactions is decided. In the context of decentralized finance, in [26], “fairness” is achieved whenever participants cannot include, exclude or front-run a transaction after having seen its content. Formal definitions of similar concepts exist in the literature. Using an older notion of “order” [28] in State Machine Replication, [29] defines “serializability” in ledger consensus. This “serializability” coincides¹ with the “*receive-order fairness*” of [6] which is defined as follows : for any two transactions x and x' , if a majority of nodes receive x before x' then x is delivered before x' . In the following, we will use the terminology *order-fairness* (OF) to refer to such properties. OF properties relate the order in which transactions are delivered to some partial orders on communication events occurring in the distributed system. *receive-order fairness* is a particular case of OF property that is defined with regards to reception events.

a) *OF w.r.t. receptions events.*: As per [6], *receive-order fairness* is impossible to achieve. Indeed, one might notice that for three transactions x_1 , x_2 and x_3 , it occur that a majority of nodes receive x_1 before x_2 , x_2 before x_3 and x_3 before x_1 , hence forming a Condorcet cycle [30].

As a result, [6] proposes a weaker property of “*block-order fairness*” and introduces the “Aequitas” consensus protocol that upholds it. Block-order fairness is defined as follows : for any two transactions x and x' , if a majority of nodes receive x before x' then no honest node can deliver x in a block after the block in which x' is delivered. However, in [6], the meaning of these blocks is more specific than the generic notion of block in a Blockchain. Indeed, they rather correspond to batching together exactly the transactions that are in the same Condorcet cycle (i.e., transactions that are not in a Condorcet cycle together must not be put in the same block). [19] clarifies this via defining “ γ -(all)-batch-order fairness” as follows : for $\gamma > 1/2$, if a proportion of nodes greater than γ receives x before x' and if x and x' are not in a Condorcet cycle, then x must be delivered before x' . [19] also introduces the Themis algorithm that upholds γ -(all)-batch-order fairness.

[15] introduces “*differential-order fairness*” and a “quick order-fair atomic broadcast” protocol that upholds it. This property is defined as follows: for any two transactions x and x' , if the number of honest nodes that receive x first exceeds by more than $2 * f$ (where $n = 3 * f + 1$ is the total number of nodes) the number of nodes that receive x' first, then x must be delivered before x' .

These properties encode an intuitive notion of fairness in consensus ordering, as the delivery order, if “fair”, should

¹in [29], the “majority” is defined as the set of all honest nodes and “reception” of a transaction x by a node signifies x enters the node’s local view of the mempool

mimic the “reception” order. Protocols that uphold such OF properties are called Algorithmic Committee Ordering algorithms in [26].

However, most existing consensus algorithms and Blockchain systems were not designed with these properties in mind. As a result, assessing the propensity of an existing Blockchain system to violate such OF properties can be difficult. In particular, the notion of “reception” may be ambiguous. For instance, it may be so that clients do not broadcast their transactions to all the nodes (then the instants of “reception” do not necessarily exist). In Blockchains using the execute-order-validate architecture, different subsets of nodes might perform different tasks and thus do not receive transactions in the same manner (in HyperLedger Fabric [31], peers receive transactions before orderers do).

b) *Limitations of OF defined w.r.t. receptions.*: If a protocol upholds an order-fairness property that is defined w.r.t. reception events, it only implies that the delivery order cannot be tampered with given certain reception orders (one per participating node). However, an adversary may actually tamper with the reception orders themselves, in order to influence the delivery order. For instance, if the adversary benefits from a quicker network connection than honest nodes and clients (lower latency when the adversary receives and sends transactions), it may listen to an incoming transaction x and front-run it via submitting x' and ensuring that most nodes receive x' before x [26]. In that case, upholding receive-order-fairness guarantees the front-running will succeed.

But even without the reduced latency requirement, an adversary may still succeed. Let us consider a consensus algorithm that upholds a form of γ -(all)-batch-order fairness. Then, as described in [30] the adversary may artificially create Condorcet cycles by submitting transactions in a specific manner : for any transaction x'' , it immediately (instant (1)) sends x'' to half of the nodes and then waits before (instant (2)) sending x'' again to the other half of the nodes. Indeed, it is then likely that x'' will form a Condorcet cycle, “trapping” a number of transactions that were submitted between instants (1) and (2). Considering our previous front-running example, if both x and x' are trapped in the Condorcet cycle formed by x'' , it is more likely for the front-running to succeed (given that no constraint is enforced on the ordering within a block and that x and x' are, as a result, in the same block).

As a result, as shown in [30], existing Algorithmic Committee Ordering algorithms are not robust against all forms of transaction reordering attacks.

c) *Blind-order fairness.*: [32], [33] (called “on-chain commit and reveal” in [26]) solves the problem of transaction order manipulation by hiding the content of the transactions until their delivery order has been decided. As the adversary ignores the content of the transactions, it cannot easily identify and choose the transactions it wants to reorder (except through some leakage of metadata or if it just wants to front-run blindly [33]). Blind-order fairness can be achieved in various ways. Time-lock puzzles [34] or delay encryption [35] enable time-based implementations (i.e., decryption is only possible after

an amount of time has passed, e.g., that of solving a puzzle). Verifiable secret sharing and threshold encryption [36] enable communication-based implementations (i.e., knowledge of the key for decrypting transactions is split in shares that are only gathered after delivery). In any case, blind-order fairness requires significant overhead (computational and communication costs) [33] and may negatively impact latency and throughput [26]. Moreover, even though it may limit the ability of an adversary to perform transaction reordering attacks, blind-order fairness does not reduce the number of order-fairness violations that may naturally occur (due to e.g., network delays and non-deterministic communications) in the system. In other words, implementing blind-order fairness does not make the system “fair” but only prevents an adversary to make it “more unfair” than it already is (see also the notion of “bounded unfairness” in [37]).

d) *Send-order fairness.*: Instead of considering the orders with which nodes receive transactions, “send-order fairness” [6] relates the orders in which transaction are emitted (by clients) to the order in which they are delivered. Unlike the variants of *receive-order fairness*, upholding *send-order fairness* may prevent an adversary with a better network connection to perform front-running as it cannot possibly make so, after receiving a transactions x and sending x' to front-run it, that the time of emission of x' is lower than that of x . However, actually implementing a system that upholds *send-order fairness* remains an unsolved problem [6]. Indeed, it requires keeping track of the global order in which emission events occur across all the clients. Because distant machines have uncorrelated local clocks, and because, in any case, Byzantine clients may falsify timestamps, one cannot simply rely on trusting clients to report the time at which they send transactions. A potential solution would be to use a Byzantine Fault Tolerant vector clock algorithm. However, such algorithms do not exist (it is impossible in an asynchronous network as per [38], at most, we have Crash Fault Tolerance for vector clocks [39]). A partial solution would be to combine Trusted Execution Environments and a Clock Synchronization algorithm [40] such as NTP [41]. In addition of the communications overhead they incur, such algorithms only guarantee that at any given time, the difference in the readings of any pair of local clocks do not exceed a certain maximal skew δ [42]. This is problematic in cases where the difference between the timestamps of events is smaller than δ . Still, the notion of *send-order fairness* is particularly pertinent as it is the most accurate description of order fairness. Moreover, in a controlled environment (e.g., in a simulator), it can actually be monitored.

e) *Fairness to clients.*: *Send-order fairness* is, intrinsically, a form of “fairness to clients”. More generally, *client-fairness* refers to nodes treating clients fairly, which includes the order with which transactions coming from distinct clients should be handled [43], [44]. Jain’s fairness index [45] quantifies fairness between m client sharing resources. Jain’s index is 1 if all clients receive the same allocation and k/m (for any $k \in [1, m]$) if $m - k$ clients receive no allocation and the k

others equally share what remains. Jain’s index represents a global notion of fairness which may also be used to quantify “fair access to transactions from all clients”. In that context, value 1 would mean that transactions from all clients are treated equally and value 0 that the transactions coming from one specific client are always prioritized over the others. The “client-fairness score” from [16] describes a similar notion. However, whereas Jain’s fairness index is global, this score is defined on a “per client” basis, as it evaluates the fairness towards a specific client.

f) Bounded unfairness.: The “strongest” (i.e., closest to fairness to the clients) notion of order-fairness is send-order fairness. However, we have seen that it is not achievable due to lacking mechanisms to maintain a BFT global clock across distant clients. The second strongest is receive-order fairness which is also impossible to uphold due to the Condorcet paradox [6]. The variants of batch-order fairness [19], [15] simply ignore the problem by ignoring pairs of transactions that violate receive-order fairness if they are in the same Condorcet cycle. However, in practice, distributed ledgers must output a total order on transactions (i.e., transactions from the same Condorcet cycle cannot remain unordered). We have also seen that an adversary may perform Condorcet attacks [30] to force sets of transactions into the same cycle. In turn, this render protocols upholding batch-order fairness powerless as they only guarantee order-fairness between transactions that are not in the same cycle. As for blind-order fairness [33], it may only prevent an adversary from making the system more unfair than it already is, provided there is no metadata leakage and at the cost of computation and/or communication overheads. As a result, there are no ideal solution to the problem of providing an order-fair total order of transactions. Still, instead of trying to provide a fair order, one can approximate a solution by minimizing unfairness in the order.

This is what is proposed in [37] via the notion of “*bounded unfairness*”. Given a threshold $\gamma > 1/2$ and a bound $\Omega \in \mathbb{N}$, a ledger satisfies “ γ - Ω -bounded-unfairness” if, for any pair of transaction x and x' , if a proportion of nodes greater than γ receives x before x' then x cannot be delivered at an index that is greater than Ω plus the index at which x' is delivered. We can see that when $\Omega = 0$, bounded-unfairness coincides with the classical formulation of *receive-order fairness* and is therefore impossible to uphold. [37] proves that there exists a lower bound for the value of Ω from which it becomes possible to guarantee γ - Ω -bounded-unfairness. [37] also shows that the Algorithmic Committee Ordering algorithms from [6], [19], [15] (resp. Aequitas, Themis and the quick-order-fair broadcast) as well as other timestamp-based methods (Pompe from [46] and Wendy from [47]) do not uphold γ - Ω -bounded-unfairness with this minimal bound.

Moreover, [37] proves that calculating this bound (which depends on the actual transactions that are considered and their dependencies i.e., whether or not a γ -fraction of nodes has received one before the other) and a total order of transactions such that γ - Ω -bounded-unfairness is upheld corresponds to solving a NP-hard problem. Thus, any protocol that would

guarantee the minimal unfairness in its ordering would not satisfy liveness. [37] details one such protocol called *Taxis_{WL}* (*WL* standing for Weak Liveness). To conclude, any attempt at providing order-fairness in transaction ordering either **(1)** only solves part of the problem (via batch-order fairness, which only provides a partial instead of a total order), **(2)** only focuses on making the order impervious to manipulations (via blind-order fairness) or **(3)** yields a protocol that does not satisfy liveness.

Reconciling order-fairness and liveness necessary requires upholding weaker forms of order-fairness. This means that the total order that is output is not necessarily the “best” but is chosen according to a certain heuristics. In the literature, the most common mechanisms used to implement such heuristics involve reasoning on timestamps. The Pompe algorithm from [46] uses the median timestamps of the times at which nodes in a quorum receive a transaction to determine its delivery order. Because there are at most f Byzantine nodes (which may maliciously provide exceedingly low or high values), using the median value guarantees that the retained value is greater or equal than a timestamp send by an honest node and also lower or equal than a timestamp send by another honest node. In the live version of *Taxis_{WL}* (simply called *Taxis*) [37], a timeout mechanism is used to break long cycles of dependencies between transactions and transactions within the same Condorcet cycle are sorted according to the median timestamp. In [30], the Themis algorithm from [19] is extended with a method to sort transactions within Condorcet cycles. Instead of relying on a median timestamp, it uses Tideman’s Ranked Pairs algorithm.

D. Adversary models

To assess the robustness of a DS, it is a common practice (from Cybersecurity) to consider an attacker actively trying to harm it. Adversary models describe such attackers [4]. The level of abstraction of these models may vary from natural language to concrete algorithms and implementations. Historically, adversary models such as the symbolic Dolev-Yao [2] model were central to the design of provably-secure cryptographic schemes. In formal verification of (cryptographic) security protocols, whether a symbolic or computational model is used, the adversary is all powerful on the network (i.e., it can intercept all messages and compute and send new messages to any node) but has no power outside of it (i.e., side channels attacks or code injection are not modeled) [3]. However, in other fields of computer sciences, in which the use of adversary models remain limited [4], these assumptions on the power of the adversary may not be desirable.

In [4], a more generic description of adversary models according to three aspects is discussed. These correspond to the adversary’s **(1)** assumptions, **(2)** goals and **(3)** capabilities. Assumptions involve the conditions under which the adversary may act. Goals correspond to the adversary’s intentions (which are related to information retrieval in most of the literature on cryptography). Capabilities synthesize all the actions the adversary may perform. A passive attacker may only eaves-

drop on message passing without any tampering while an active attacker may, among other things, intercept and modify messages (Man-In-The-Middle attack).

Certain assumptions may bind the capabilities of adversaries. Adaptability [48] refers to the ability of the adversary to update its plan i.e., the choice of its victims and of which adversarial actions to perform. While static adversaries have a fixed plan (established before the execution of the system), adaptive adversaries may make choices at runtime. Threshold cryptography [49] was introduced as a means to share a secret securely among a fixed set of participants, a threshold number of which being required to access it. Hence, adversaries attacking such protocols within its assumptions must not be able to infect more participants than the threshold, thus bounding their power. By extension, adversarial actions can be limited by a corresponding resource as in [10] (bounded resource threshold adversaries), or via a more abstract notion of budget.

Various adversary models have been designed specifically for Blockchain systems. For instance, that from [50] focuses on network connectivity (i.e., the adversary only performs network related actions) and how this can be exploited to impact the consensus mechanism. The limitation of adversary models to only represent network related actions stem from their origin in cryptographic protocol verification [3], in which the adversary has a total control over the network.

The adversary model from [16] (which this present paper extends) allows modeling an adversary which manipulates the network. However, it may additionally target individual sub-systems (thus modeling e.g., side channel or code injection). Moreover, it incorporates notions of communication models [8], failure models [9] and Fault Tolerance thresholds to further limit the capabilities of the adversary [10], which enables parameterizing attacks depending on assumptions under which involved distributed protocols are to be used. This approach also allows observing the side-effects of attacks (whether or not the adversary succeeds) on various metrics. If the side effects of an attack are known, their observation might allow detecting the attack.

E. Simulation

Validating systems can either involve formal verification or testing which are two orthogonal approaches [9]. The former involves techniques such as model checking or automated theorem proving which do not scale well with the complexity of verified systems and properties. Indeed, in complex and dynamic DS involving several protocols we must then consider all possible correct as well as incorrect behaviors (due to failures up to the relevant failure model) combined with all possible interleaving and side effects between these behaviors (due to communications delays up to the relevant communication model and/or performance failures). Although techniques can be used to negate (e.g., universal composability) or mitigate (e.g., partial order reduction or symbolic execution) this complexity, this is not always possible. In this context,

empirical security evaluations, in the form of testing, is more adapted.

Tests can be performed against a concrete implementation of the DS. However, it may involve unexpected side-effects due to executing the whole implementation and hardware dependent protocol stack. Similarly to software integration tests being performed via code isolation using mockups, one can focus on and isolate specific aspects of the DS via the use of a simulator in which parts of the protocol stack are abstracted away. Additionally, this allows a finer control over communications because they occur within the simulator and not on a network on which control is lacking. Simulation also has the added advantage of being able to closely and easily monitor the simulated system e.g., by collecting metrics (which is more difficult on a live network or testnet deployed on a WAN).

As highlighted by the recent review [18], blockchain simulators are important tools for understanding these complex systems. Yet, to our knowledge, none have been fitted with a programmatic adversary to simulate adversarial attacks. Most simulators address specific blockchain systems (Bitcoin, Ethereum) and/or are oriented towards performance evaluations (latency & throughput) rather than security aspects. Shadow-Bitcoin [51] allows replicating Bitcoin networks on a large scale, but lacks flexibility for non-Bitcoin blockchains. While lacking an adversary, some simulators can be adapted for security analyses such as eVIBES [52] for Ethereum networks.

Multi-Agent-Systems (MAS) is an agent-oriented modeling paradigm which is particularly adapted to DS with large numbers of agents. The behavior of each agent can be proactive (following a specific plan regardless of their environment) and reactive (reacting to stimuli e.g., incoming messages). Agent Group Role (AGR) [53] is a MAS framework which focuses on the interactions agents have by playing roles within groups. MAX [17] is a simulator based on AGR that leverages MAS for blockchain networks.

F. HyperLedger Fabric

[16] demonstrates an approach to computational studies of order-fairness via adversary-augmented simulation on a specific Blockchain system : HyperLedger Fabric (HF) [11]. Although generally secure, HF has some known vulnerabilities. If the addresses of peers are known by malicious entities, DoS [31] might occur. To mitigate this, [31] recommends anonymizing peers (e.g., using random verifiable functions and pseudonyms). HF chaincode is vulnerable to (smart contract) programming errors [54] which can be mitigated by formal verification of smart contracts [31] and updating deployed contracts [55]. HF, like any other permissioned blockchain, is vulnerable to the compromise of the Membership Service Provider (MSP) [54]. Potential solutions include using secure hardware for registration and transactions [31], and monitoring requests to detect potential attacks [55]. HF's flexible consensus protocols have distinct strengths and weaknesses. [55] points out the vulnerability to Network Partitioning from internal attackers affecting network routing, identifiable via

methods like BGP hijacking and DNS attacks. Its Gossip protocol, essential for block delivery, is susceptible to Eclipse attacks [54].

The attacks on HF defined in [16] have not been described previously [56] and correspond to transaction reordering attacks [26].

III. OUR ADVERSARY MODEL

In this section we define our adversary model. Fig.2 illustrates it following the approach from [4].

In our context, the adversary's environment is the DS it aims to harm. We formalize it as a set S of sub-systems. At any given time, each sub-system $s \in S$ has a certain state (defined by e.g., the current values of its internal state variables) in a state space Γ_s . The state space of the overall system, which is the product of its sub-systems', is $\Gamma = \prod_{s \in S} \Gamma_s$. At any given time during its execution, the state of the DS is a certain $\eta \in \Gamma$.

Assumptions	Goals	Capabilities
Environment (system & assumptions): - Communication Model - Failure Model Resources (binding capabilities): - Knowledge - Fault-Tolerance thresholds	property violation	- <u>process discovery</u> - <u>adaptation</u> - adversarial actions

Fig. 2: Our adversary model (adaptive adversaries underlined)

The goals of the adversary must be clearly defined so that the success or failure of attacks can be ascertained. In the following, we consider that goal to be to invalidate a property ϕ of the system, and we denote it as a First Order Logic [57] formula. Given a state $\eta \in \Gamma$ of the system, the property can be either satisfied (i.e. $\eta \models \phi$) or not satisfied (i.e. $\eta \not\models \phi$). Thus, the goal of the adversary is to lead the system to a state η s.t. $\eta \not\models \phi$.

For instance, let us consider a DS S with two sub-systems s_1 and s_2 which must agree on a value x stored as x_1 in s_1 and x_2 in s_2 . Before agreement is reached, the value of x is undefined which we may denote as $x = \emptyset$. After consensus, the values of x_1 and x_2 must be the same. This safety property of correct consensus can be described using $\phi = (x_1 = \emptyset) \vee (x_2 = \emptyset) \vee (x_1 = x_2)$. Given a state $\eta \in \Gamma$, in order to check whether or not ϕ holds it suffices to verify that $\eta(\phi)$ (i.e., $(\eta(x_1) = \emptyset) \vee (\eta(x_2) = \emptyset) \vee (\eta(x_1) = \eta(x_2)))$) holds.

The expressiveness of this approach is only limited by the expressiveness of the language that is used to define ϕ and the state variables of Γ . Both global and local variables can be used. If the adversary has several goals we can use disjunctions (resp. conjunctions) to signify that it suffices for one of these to be (resp. requires that all of these are) fulfilled.

A. Adversarial actions

We propose in Fig.3a a novel classification of adversarial actions. We name 7 types of actions, each one being illustrated by a diagram on Fig.3. The process target of the action is

represented on the left, the other processes of the DS on the right, and the adversary below them. The black horizontal arrows represent message passing. The effects of the action is represented in red.

Actions of type `reveal` and `listen` are passive actions. While `reveal` allows the adversary to read an internal state variable of a target process (e.g., the x variable on Fig.3c), `listen` only allows reading incoming and/or outgoing messages (red arrows on Fig.3b). Because message buffers are a specific kind of state variables, a `listen` action can be performed via a `reveal` action. Hence, on Fig.3a, `listen` is a subtype of `reveal`. `listen` actions can be further specialized depending on the nature of the messages that are observed e.g., whether they correspond to inputs, outputs or both (as indicated by I/O/IO on Fig.3a).

In the real world, `listen` actions correspond to network eavesdropping (also called sniffing or snooping), a common vulnerability in open networks, particularly wireless ones as discussed in [58]. `reveal` actions can mean access with read permissions. It may also involve passive side-channels attacks [14] where a process, despite being software secure, leaks information (e.g., memory footprint, power consumption etc.), or more active tampering with certain types of memory scanning attacks [13] in which an attacker reads and interprets memory addresses associated with a process.

While actions of type `listen` and `reveal` are passive (i.e., have no direct impact on system execution), those of types `send`, `delay`, `skip`, `stop` and `inject` are active. `send` allows the adversary to send messages to a target process (see Fig.3d), which, combined with specific knowledge (see resources on Fig.2), can be used to impersonate third parties (with e.g., knowledge of private keys). With `stop`, the adversary forces a process to crash (terminate prematurely). With `skip`, it prevents message exchange between the target and the rest of the system. If `skip` concerns every messages, it is equivalent (from the point of view of the system) to `stop` (hence on Fig.3a, `skip` contains `stop`). `delay` makes so that message exchanges with the target are slowed down. As a result, it delays the reception of the messages that it receives and emits. If the added delay is infinite, then `delay` is equivalent to `skip`. In the real world, `delay` may be implemented via Denial of Service [12]. `inject` modifies the behavior of the target process either by forcing it to express a given behavior at a given time or by changing the manner with which it reacts to events (e.g., to incoming messages). This may realistically correspond to code injection attacks [59] or the adversary having user access to the target's information system.

Network actions (hatched on Fig.3a) include actions of types `listen`, `send`, `stop`, `skip` and `delay` because they may only require tampering with the network environment and not the with the target sub-system hardware or software directly.

B. Capabilities binding assumptions

The adversary's assumptions (Fig.2) include a communication model and a failure model for individual processes. These models bind the capabilities of the adversary as they do not

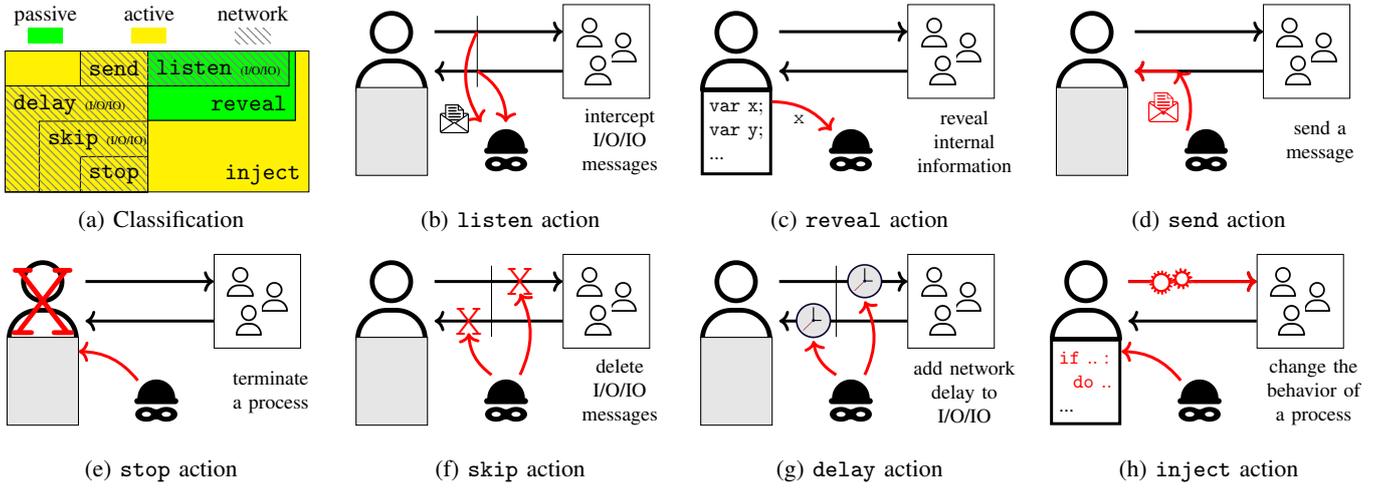


Fig. 3: Adversarial actions

allow certain classes of adversarial actions. Fig.4 summarizes these limitations.

It is always possible to perform `reveal` (and thus `listen`) actions. The asynchronous communication model always enables the unrestricted use of `delay` actions. While `skip` is allowed under the omission failure model, only `stop` is available under crash failures. Under both failure models and with the synchronous communication model, the use of `delay` actions is limited to the addition of a maximum delay δ so that the total retransmission time (i.e., between the output o and the input i) of the affected message does not exceed a certain Δ time. Given t the retransmission time without intervention from the adversary we hence have $i - o = t + \delta < \Delta$. Under the eventually synchronous communication model, this condition is only required after the GST (hence $o \geq GST$ on Fig.4).

Fail. \ Comm.	Synch.	Async.	Event. Synch.
	Crash	reveal stop delay $t + \delta < \Delta$	reveal delay
Omission	reveal skip delay $t + \delta < \Delta$	reveal delay	reveal skip delay $o \geq GST \Rightarrow t + \delta < \Delta$
Performance	reveal delay	reveal delay	reveal delay
Byzantine	inject	inject	inject

Fig. 4: Enabled actions w.r.t. assumptions

The adversary's assumptions also include its knowledge, which represents the information it possesses about the system. This includes it being aware of the existence of the various sub-systems that are part of the DS. In the case of an adaptive adversary, which may update its plan of actions according to new information, its capabilities can include process discovery. Knowledge can directly bind adversarial capabilities when certain action requires specific knowledge (e.g., authentication).

We use the concept of resource limited adversary [10] to bind the capabilities of the adversary w.r.t. FT thresholds. We abstract away adversarial actions as a set A . Each action $a \in A$ has a target sub-system $s(a) \in S$, and a baseline cost $\kappa(a) \in \mathbb{K}$, where \mathbb{K} is the ordered vector space in which the budget of the adversary is represented. The adversary is bound by a certain initial budget $B \in \mathbb{K}$ which limits its capabilities. For instance, let us suppose the initial budget of the adversary is the vector (f_x, f_y) , representing the maximal number of nodes it can infect on protocol x and resp. y . Then, if an action a_x involves sabotaging a node participating in protocol x , we have $\kappa(a_x) = (1, 0)$ and the remaining budget is $(f_x - 1, f_y)$ after performing a_x . Because it might cost less to target a sub-system that has already been victim of a previous action, we consider a protection level function $\psi \in \mathbb{K}^S$ (which may vary during the simulation) to modulate this cost. Then, given a current budget $b \leq B$, the adversary can perform an action $a \in A$ if the associated cost is within its budget i.e., iff $\kappa(a) \odot \psi(s(a)) \leq b$, with \odot the Hadamard product (element-wise product). After performing this action, the remaining budget is then $b - \kappa(a) \odot \psi(s(a))$. For instance, in our previous example we have an initial protection level $\psi(s(a_x)) = (1, 1)$ and therefore $\kappa(a_x) \odot \psi(s(a_x)) = (1, 0) \odot (1, 1) = (1, 0)$. After having performed a_x , the protection level for $s(a_x)$ becomes $\psi'(s(a_x)) = (0, 1)$ and therefore performing another action a'_x on that same node w.r.t. protocol x (i.e., s.t., $s(a'_x) = s(a_x)$) has no cost (i.e., $\kappa(a'_x) \odot \psi(s(a'_x)) = (1, 0) \odot (0, 1) = (0, 0)$).

C. System simulation and success of attack

Combining a model of the DS and of the adversary, we can simulate attacks and test whether or not the adversary's goal is met. The simulation's state at any time is given by a tuple (η, b, ψ) where $\eta \in \Gamma$ gives the current state of the system, $b \in \mathbb{K}$ correspond to the remaining budget of the adversary and $\psi \in \mathbb{K}^S$ gives the current protection levels of sub-systems (for each type of resource and each target sub-system). Protection levels of individual sub-systems may vary during the simulation for several reasons. It might cost less

to target a process that has already been victim of an action. Inversely, the system might heal and reset the sub-systems' protection level or apply countermeasures to increase it further.

We distinguish between two kinds of events: adversarial actions in A and system events in E (which correspond to the system acting spontaneously). Let us consider a relation $\rightarrow_E \subseteq (\Gamma \times \mathbb{K}^S) \times E \times (\Gamma \times \mathbb{K}^S)$ s.t., for any $(\eta, \psi) \xrightarrow{e} (\eta', \psi')$, η' and ψ' describe the state and protection levels of the system after the occurrence of $e \in E$. Similarly, let us consider $\rightarrow_A \subseteq (\Gamma \times \mathbb{K}^S) \times A \times (\Gamma \times \mathbb{K}^S)$. Then, the space of simulations is the graph with vertices in $\mathbb{G} = \Gamma \times \mathbb{K} \times \mathbb{K}^S$ and edges defined by the transition relation $\rightsquigarrow \subseteq \mathbb{G}^2$ s.t.:

$$\text{attack} \frac{\kappa(a) \odot \psi(s(a)) \leq b \quad (\eta, \psi) \xrightarrow{a} (\eta', \psi')}{(\eta, b, \psi) \rightsquigarrow (\eta', b - \kappa(a) \odot \psi(s(a)), \psi')}$$

$$\text{exec} \frac{(\eta, \psi) \xrightarrow{e} (\eta', \psi')}{(\eta, b, \psi) \rightsquigarrow (\eta', b, \psi')}$$

Any simulation **(1)** starts from a node (η_0, B, ψ_0) where η_0 is the initial state of the system, B is the initial budget of the adversary and ψ_0 gives the initial protection level of sub-systems and **(2)** corresponds to a finite path $(\eta_0, B, \psi_0) \rightsquigarrow^* (\eta_j, b_j, \psi_j)$ in graph \mathbb{G} , its length j being related to the duration of the simulation. To assess the success of the adversary, we then check if and how property ϕ is invalidated in that path.

IV. USE-CASE AND FAIRNESS PROPERTIES

A. Hyperledger Fabric system

A system using HyperLedger Fabric consists of a set of subsystems $S = S_c \cup S_p \cup S_o$ deployed over a network, where:

- S_c is a set of clients², with $|S_c| = m$
- S_p is a set of peers, with $|S_p| = n$
- S_o is a set of orderers, with $|S_o| = n'$

Fig.5 roughly describes the behavior of such a system. Clients may send transactions (denoted as x) to the peers, which are tasked with endorsing them. Upon receiving a transaction each peer p_i may send a corresponding endorsement with a cryptographic signature s_i . Once it has collected “enough” endorsements (which depends on the HF system’s endorsing policy parameterization), a client may then forward its endorsed transaction to the orderers (with the corresponding set of cryptographic signatures attesting its endorsement)

The orderers run a consensus algorithm to agree on the order of endorsed transactions. It is possible to use Crash Fault Tolerant algorithms in cases where the network is deemed safe or Byzantine Fault Tolerant ones if this is not the case. In any case, this allows orderers to order transactions in batches that are called blocks.

For the Tendermint [20] BFT consensus algorithms, which we will consider in the remainder of the paper, orderers may successively take the role of a proposer, which proposes a value for the next block of transactions ($[x_1, \dots, x_u]$ on Fig.5).

²for the sake of simplicity, we do not distinguish between clients and the applications through which they interact with HF

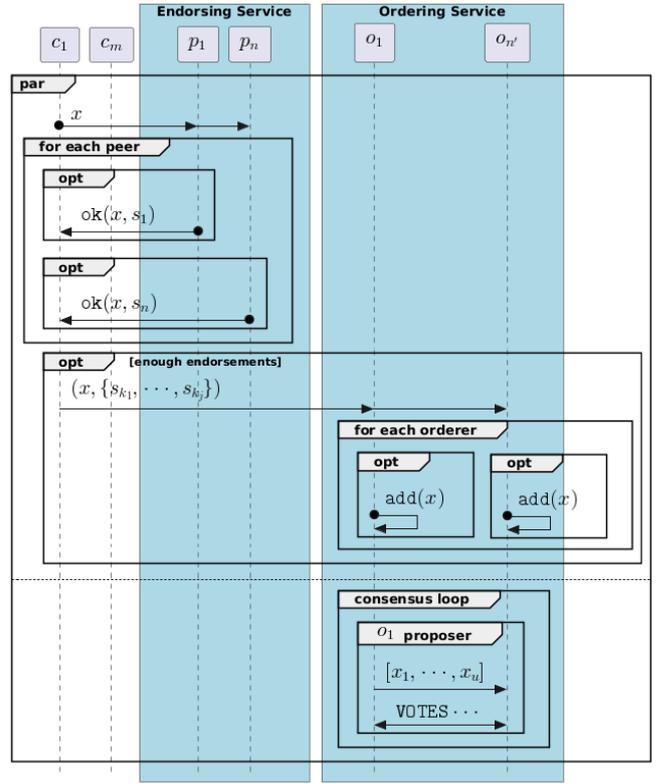


Fig. 5: Schematic behavior of HF

For honest proposers, the content of that block corresponds to the set of endorsed transactions it has received up to that moment and which have not already been put in a previous block. As for the order of transactions within the block, it corresponds to the local order with which that orderer received them.

Hyperledger Fabric (HF) thus implements a Distributed Ledger, and, more specifically, a non-revocable permissioned Blockchain. We say that a transaction is delivered once it has been put in a block. The delivery order of transactions corresponds to the order with which they are delivered on and within blocks. We may refer to the set of peers (resp. orderers) as the “endorsing service” (resp. “ordering service”).

B. OF properties as evaluation metrics

In the following, we define several metrics that can be used to monitor the effect of transaction reordering attacks on our HF system. These metrics correspond to counting the number of pairs of transactions (x, x') for which specific OF properties are violated.

In the case of HF, there is no single notion of “node” because there is a distinction between peers and orderers. Thus, the notion of “reception” can be interpreted in two manners. Indeed, we can either consider the receptions of not-yet endorsed transactions by the peers, or the receptions of endorsed transactions by the orderers.

Let us consider properties OF_α^β of the form: for any transactions x and x' , **if** $\alpha(x, x')$ **then** $\beta(x, x')$.

The premise of the property, which involves partial orders between communication events, is represented by the α predicate. We consider three variants : $\alpha \in \{S_{ND}, E_{DS}, O_{RD}\}$. If we have $S_{ND}(x, x')$, then the emission of x precedes that of x' . If we have $E_{DS}(x, x')$ then, a majority of peers receive an endorsement request for x before they do for x' . Finally, if $O_{RD}(x, x')$ then, a majority of orderers receive x with sufficient endorsements before they do for x' .

The property's right-hand side, represented by the β , involves partial orders between the delivery events. We consider two variants : $\beta \in \{D_{LV}, B_{LC}\}$. $D_{LV}(x, x')$ (resp. $B_{LC}(x, x')$) corresponds to x being delivered before x' (resp. x being delivered in a block that is either before that or the same as the one in which x' is delivered).

Let us remark that $OF_{S_{ND}}^{D_{LV}}$ corresponds to *send-order fairness*, while $OF_{E_{DS}}^{D_{LV}}$ and resp. $OF_{O_{RD}}^{D_{LV}}$ correspond to *receive-order fairness* from the perspective of the peers and resp. the orderers. Because blocks in HF do not coincide with Condorcet cycles, $OF_{E_{DS}}^{B_{LC}}$ and resp. $OF_{O_{RD}}^{B_{LC}}$ do not correspond to the γ -batch-order fairness from [19] (which is a generalization of the *block-order fairness* from [6]). Still, counting the violations of $OF_{S_{ND}}^{B_{LC}}$, $OF_{E_{DS}}^{B_{LC}}$ and $OF_{O_{RD}}^{B_{LC}}$ is pertinent as, if one observes an increase of these numbers as the adversarial power increases, this means that the adversary pushes transactions x it front-runs into blocks after the one in which the corresponding x' is delivered.

C. Application layer & client fairness

Let us consider that several clients repeatedly compete to solve puzzles. For a given puzzle k , a client c wins iff the first delivered transaction that contains the solution of k was sent by c . All clients having the same aptitude, the game is *client-fair* iff every client has the same likelihood of winning, which is $1/m$ where m is the number of clients. Let us denote by g the total number of resolved puzzle competitions.

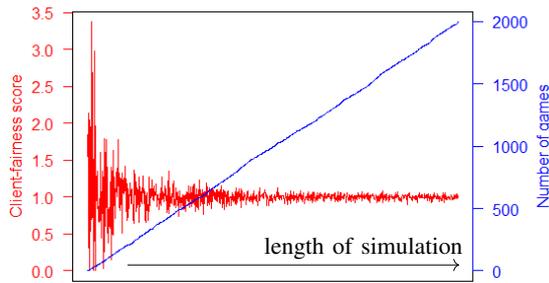


Fig. 6: Convergence of client-fairness score towards 1

As blocks are regularly delivered, g increases with the length of the simulation, as illustrated on Fig.6 (in blue on the right axis). For a client c , we denote by $\%g(c)$ the percentage of games it has won during a simulation. The game is *client-fair* if, for all clients c , the longer the simulation is, the more $\%g(c)$ is close to $1/m$. Via defining a client-fairness score $\text{score}(c) = \%g(c) * m$, we obtain an independent metric, that

converges towards 1, as illustrated on Fig.6 (in red, on the left axis) if the game is client-fair.

We consider the goal of the adversary to be to diminish the likelihood for a target client $c \in S_c$ to win puzzles. We formalize this as the *client-fairness* property: $\phi(c) = (g > 5000) \wedge (\text{score}(c) < 0.75)$. This signifies that, the adversary wins if, after more than 5000 competitions have been resolved, the client's score is less than 0.75.

Using this specific application use-case provides us with an additional client-fairness score metric as well as a clear goal for the adversary, which success can thus be measured. Additionally, it is particularly convenient for quantifying the number of *OF* violations. Indeed, counting the violations amounts to comparing, for each pair of delivered transactions x and x' , their delivery order and either the order of their emission or the order of their reception in each of the peers or orderers. If we consider X to be the total number of delivered transactions, there are $X * (X - 1)/2$ such pairs to consider. Moreover, for our results to be statistically significant, we need a high number k of puzzle games, which yields a large number of transactions $X = m * k$ having m (the number of clients) transactions per game. Thus, comparing reception orders on every pair of transactions on every peer amounts to $n * m * k * (m * k - 1)/2$ comparisons, which quickly become intractable. Thanks to our puzzle use-case, the order between two transactions only matters if they concern the same puzzle game as they may commute otherwise. Hence, we only need to consider $n * k * m * (m - 1)/2$ comparisons.

D. Parameterization of the system

For the experiments, we consider the network described on Fig.7. The peers (p_1 to p_n on the left) exclusively communicate with the clients (c_1 to c_m in the middle). The orderers (o_1 to $o_{n'}$ on the right) communicate with the clients and among themselves. Peer to peer communications is simulated with no loss and delays for the transmission of individual messages are sampled from a probability distribution denoted as \ominus . As recommended in [60], we consider hypoexponential distributions for these delays and we bound the maximal value a delay may take with a bound Δ , which corresponds to that of a partially synchronous communication model [8] (in its formulation without the GST, with Δ not known by any of the involved parties).

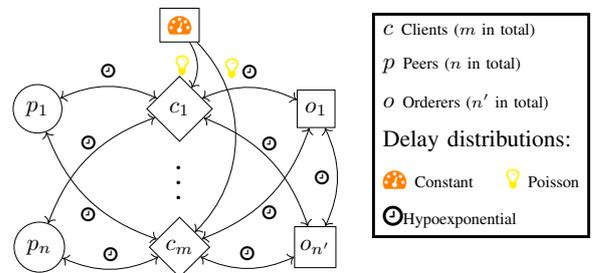


Fig. 7: Fabric network with delays

We suppose that puzzles are revealed at regular intervals, represented by 🧩 on Fig.7, and that all clients are aware of the puzzles at the same time. Any client can solve a puzzle in a certain amount of time that is sampled from a Poisson distribution, represented by 💡 on Fig.7. We use the same 💡 distribution for all clients so that they all have the same ability to solve puzzles.

Upon solving a puzzle, a client submits the corresponding transaction to HF (via the peers and then the orderers). In our use-case, the HF system also receives third party transactions (that do not correspond to puzzle solutions) at regular intervals. These heartbeat transactions simulate a concurrent usage of the HF system by other applications (besides the puzzle game).

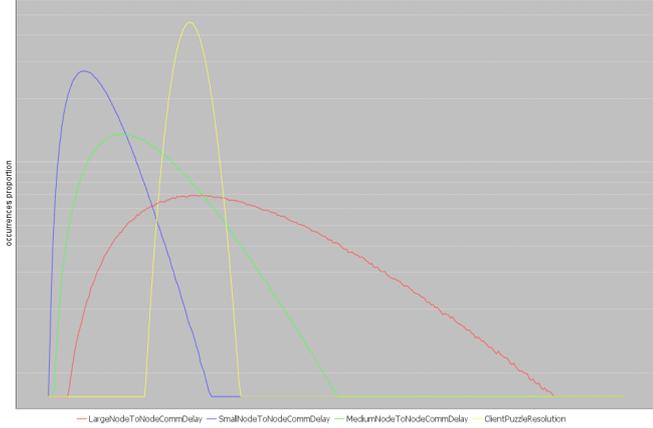


Fig. 8: The delay distributions

We consider three variants of the 🧩 delay distribution: one with small delays denoted as ⚡ (in blue), another with medium delays denoted as 🟢 (in green) and a third with large delays denoted as 🔴 (in red). Fig.8 represents these distributions in logarithmic scale.

The endorsing policy of HF is parameterized as follows: for a transaction to be endorsed and forwarded to the ordering service, it suffices to obtain at least $n/2$ endorsements from distinct peers, n being the total number of peers.

Finally, we parameterize the cost of actions and the budget B of the adversary so that it cannot apply adversarial actions to more than $(n' - 1)/3$ orderers (where n' is the total number of orderers) and $n - n/2$ peers. As a result, for the ordering service, which runs Tendermint, we remain in the hypotheses of Byzantine Fault Tolerance (below the one third threshold). Likewise, for the endorsing service, the adversary cannot guarantee censorship of the transactions.

V. BASIC ATTACK SCENARIOS

We consider several basic attack scenarios that can be combined by the adversary to harm the target client c .

A. Peer sabotage

Peer sabotage consists in applying an inject action to a peer so that it never endorses transactions from c . If the adversary were to sabotage more than $n/2$ peers, it would

guarantee that no transactions from c are ever delivered because there are n peers and at least $n/2$ distinct endorsements are required. However, sabotaging fewer peers still has an effect, particularly in a slow network.

Let us indeed denote by t the time required for c to receive an endorsement (for a given transaction) from any given peer $p \in S_p$. We represent the probability of receiving the endorsement from p before a certain timestamp z using $\mathcal{P}(t < z)$. If we suppose all events to be independent (i.e., we have i.i.d. variables) and have the same likelihood (i.e., peer to peer channels of communications have equally probable delays), for any honest peer p we may denote by X this probability $\mathcal{P}(t < z)$. On the contrary, the endorsement from p being received after t has a probability $\mathcal{P}(t \geq z) = 1 - X$.

Among n trials, the probability of having exactly $k \leq n$ peers endorsing the transaction before timestamp z is:

$$\mathcal{P}(k \text{ endorsement} < z) = \binom{n}{k} * X^k * (1 - X)^{n-k}$$

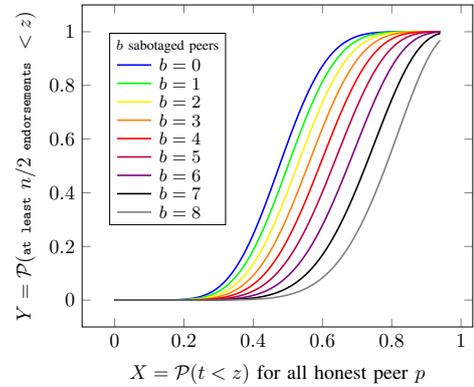


Fig. 9: Theoretical effect of peer sabotage (for $n = 20$)

Sabotaged peers never endorse transactions (we always have $X = 0$ for any timestamp z) and can therefore be ignored when counting the numbers of endorsements. Therefore, given $b \leq n/2$ the number of sabotaged peers, the probability Y of having at least $n/2$ endorsements from distinct peers before z is:

$$Y = \sum_{k=n/2}^{n-b} \binom{n-b}{k} * X^k * (1 - X)^{n-b-k}$$

On Fig.9, we plot this probability Y w.r.t. X which corresponds to the probability $\mathcal{P}(t < z)$ for honest peers. On this plot, we consider a system with $n = 20$ nodes, $n/2 = 10$ endorsements being required. We can see that the more peers are sabotaged, the smaller is the probability of collecting enough endorsements before timestamp z . We conclude that, infecting a minority of peers statistically delays the endorsement of transactions from c . This delay might in turn be sufficient to force these transactions into later blocks in comparison to transactions from other clients emitted at the same time.

B. Orderer sabotage

Winning a puzzle requires a solution-carrying transaction to be ordered in a new block. For this purpose, Tendermint [20] consensus instances are regularly executed by the orderers. Tendermint is based on rounds of communications, each one corresponding to an attempt to reach consensus. These attempts rely on a proposer to PROPOSE a new block, which will then be voted upon.

The adversary can sabotage an orderer via an inject action to force it not to include transactions from c whenever it proposes a new block. Because there are unknown delays between emissions and corresponding receptions (which might be arbitrary in the asynchronous communication model, or bounded in the synchronous and partially synchronous communication models), and because some messages might even be lost (depending on the failure model) it is impossible for the other orderers to know whether these transactions were omitted on purpose or because they have not been received at the moment of the proposal (guaranteeing the discretion of the attack).

If there are sabotaged orderers, the likelihood of transactions from c to be included in the next block diminishes, thus negatively impacting its client-fairness score. However, because the proposer generally isn't the same from one round to the next, infecting less than $f' = (n' - 1)/3$ orderers cannot reduce the score to 0 i.e., total censorship is not possible. Yet, because orderers (as Tendermint [20] is used) require $2 * f' + 1$ PRECOMMIT messages to order a block, if the adversary sabotages more than f' orderers and makes so that these orderers do not PREVOTE and PRECOMMIT blocks containing transactions from c , then, total censorship is possible.

VI. SIMULATIONS OF THE ATTACKS

Using MAX [17], we have simulated attack scenarios which are combinations of basic attacks from Sec.V. In these attacks, the adversary attempts to reduce the fairness score (as defined in Sec.IV) of a specific client. To do so, it infects a minority of peers and/or orderers.

For the simulations of this section, we consider $m = 3$ clients, $n = 25$ peers and $n' = 25$ orderers (in Sec.VIII-B, we consider additional simulations, varying m , n and n'). To be endorsed, a transaction requires 12 endorsements from distinct peers. We use an arbitrary unit of time denoted as "tick" in our discrete time simulations. Heartbeat transactions are submitted every 20 ticks. A new puzzle is revealed every 200 ticks and the Poisson distribution which determines the time required by a client to solve a puzzle has a mean of 75 ticks. The Tendermint consensus is parameterized so that empty blocks can be emitted and the timeout for each phase is set to 1500 ticks. In every simulation 5000 puzzles are revealed and the duration of the simulation is 1 200 000 ticks (we wait 200 000 ticks after the last puzzle is revealed so as to guarantee every puzzle to be solved). The details of the experiments and the means to reproduce them are available at [61].

We vary the following parameters : (1) the Delay distribution, which can either be small (Small in blue), medium (Medium in green) or large (Large in red), (2) the proportion of infected peers (between 0% and 50%) and (3) the proportion of infected orderers (between 0% and 33%).

For each simulation, we measure 9 metrics : the numbers of OF_{SND}^{DLV} , OF_{SND}^{BLC} , OF_{EDS}^{DLV} , OF_{EDS}^{BLC} , OF_{ORD}^{DLV} and OF_{ORD}^{BLC} violations, the client-fairness $\text{score}(c)$ of the target client c , the total number of blocks that is delivered and a measure on the observed distribution of the sizes of the blocks (the third quartile).

A. Analysis of the results

a) *Diagrams presentation.*: Fig.10 focuses on the effect of the attack on the endorsing service (i.e., the effect of the adversary infecting increasingly more peers). The 9 diagrams on Fig.10 plot our 9 metrics on the vertical axis, with, on the horizontal axis, the proportion of peers that are infected by the adversary. In each diagram, we have 9 curves, which correspond to 9 combinations of parameters. Each curve has 13 data points, which correspond to infecting between 0 and 12 peers. Thus, Fig.10 reports metrics collected from 117 distinct simulations.

As indicated in the legend, the three blue (resp. green and red) curves correspond to simulations in which the small (resp. medium and large) delay distribution Small (resp. Medium and Large) is considered. The shape of the points and style of the lines correspond to proportions of infected orderers (0%, 16% or 32%), highlighting the effect of combining peer and orderer sabotage.

The six diagrams on the left correspond to the number of violations of our six OF properties. These six diagram share the same scale on the vertical axis. The 3 diagrams on the right report on the $\text{score}(c)$ metric of the targeted client c as well as the total number of blocks that have been delivered and their size (we report the third quartile of the observed distribution of the size of the blocks as it is more stable and is not too much biased by the presence of empty blocks).

b) *Discussion on peer sabotage.*: We observe that, when no peers and no orderers are infected (at horizontal position 0, for the plain line curves), the score of the target client (i.e. the proportion of puzzles it has won, relative to the total number of puzzles (~ 5000) and the total number of clients) stays around 1 which is expected as per Fig.6.

As discussed in Sec.V-A, peer sabotage statistically delays the endorsement of transactions from the target client. Because collecting sufficiently many endorsements for these transactions is delayed, their delivery is likely to be delayed as well, resulting in other clients being more likely to win puzzle games. Experimentally, this effect is observed on Fig.10, especially on the diagrams corresponding to the score and OF_{EDS}^{DLV} metrics. Indeed, we observe that, as the proportion of infected peers increase, the score decreases and the number of receive-order fairness violations w.r.t. the instants at which peers receive the transactions increases. The attack is quite successful, as the score almost reaches 0 (meaning the target

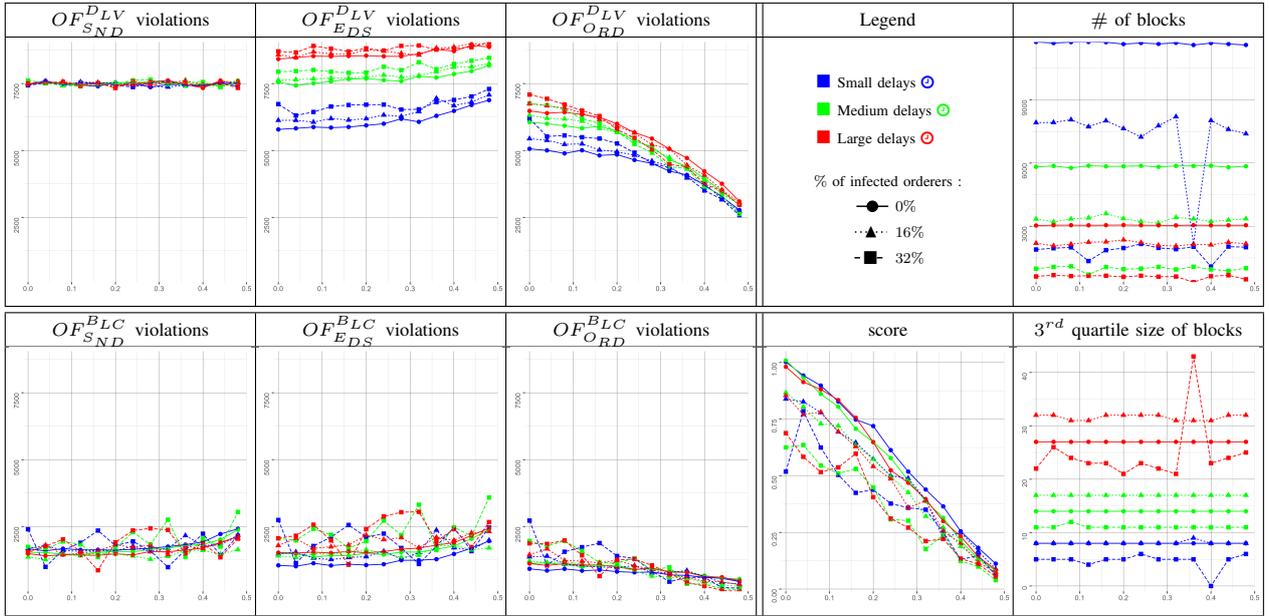


Fig. 10: Varying the number of infected Peers

client almost never wins) when the proportion of infected peers comes closer to the threshold (which is 50% in our use-case, having a majority endorsing policy).

This statistical delay, because it is correlated to Y on Fig.9, depends on the distribution of the communication delays between the peers and the clients, which corresponds to either the blue \odot (small delays), green \ominus (medium delays) or red $\omin�$ (large delays) from Fig.8 and is correlated to X on Fig.9. Indeed, with higher communication delays, the likelihood that an honest peer receives an endorsement before a certain timestamp, which corresponds X on Fig.9, decreases. At lower values of X , the values of Y are also lowered.

On Fig.10 this translates into having, at a fixed proportion of infected peers (i.e., horizontal position), a lower score for larger delays. Indeed, if we look at, e.g., the top three curves on the score diagram, we observe that the score is higher in the case of the small delays \odot . For OF_{EDS}^{DLV} , we observe that the baseline number of OF_{EDS}^{DLV} violations (whether or not peers are infected) is higher at higher values of delays. This is due to the fact that larger randomly sampled delays are more likely to cause endorsed transactions to be mis-ordered due to overtaking in communications from the peers to the clients and then from the clients to the orderers. Also, the number of violations tends to increase as more peers are infected and this increase is sharper for the smaller delays (at larger delays there are already many naturally occurring violations so the effect of the adversary is less noticeable). Similar remarks may be made w.r.t. OF_{EDS}^{BLC} .

As for OF_{ORD}^{DLV} , we also observe a higher number of violations at higher delays. However, an increase in the proportion of infected peers results in a decrease in the number of OF_{ORD}^{DLV} violations. Indeed, unlike OF_{EDS}^{DLV} which is defined w.r.t. the instants at which peers receive transactions, OF_{ORD}^{DLV} is defined w.r.t. the instants at which orderers receive

endorsed transactions. Infecting peers results in tampering with the delivery order but also with the order with which orderers receive endorsed transactions (the attack targeting the endorsing service which is upstream w.r.t. the orderers). Thus, the attack does not yield an increase in OF_{ORD}^{DLV} violations. On the contrary, it makes so that there is less competition between transactions from the target client and the other clients. Indeed, because delayed transactions are more likely to be both received and ordered after those of the other clients for the same puzzle, there are less risks of OF_{ORD}^{DLV} violations, having less pairs of transactions susceptible to cause such violations. Interestingly, in our simulations we have three clients, and, after neutralizing the target client on the endorsing service, the number of OF_{ORD}^{DLV} violations roughly decreases by two thirds. Similar remarks may be made w.r.t. OF_{ORD}^{BLC} .

Concerning OF_{SND}^{DLV} , the effect of the attack is not noticeable. It may be so that the non-determinism of the network already saturates the number of violations. However, even if there still is some noise, we can observe an increase in OF_{SND}^{BLC} violations with the proportion of infected peers. The statistical delay caused by peer sabotage is indeed likely to force transactions from the target client into later blocks, which is observable when considering OF_{SND}^{BLC} .

Finally, at higher delays, there are less blocks but these blocks are bigger. Infecting peers do not have a noticeable effect on the number and size of blocks, as the ordering service is not directly tampered with.

c) *Discussion on orderer sabotage.*: Fig.11 reports on 81 simulations (having 9 curves, and 9 data-points per curve) that focus on the effect of infecting orderers. The diagrams of Fig.11 are similar than those of Fig.10, the main difference being that the horizontal axes rather correspond to the proportion of infected orderers. Also, the shape of points and style of the lines correspond here to proportions of infected peers

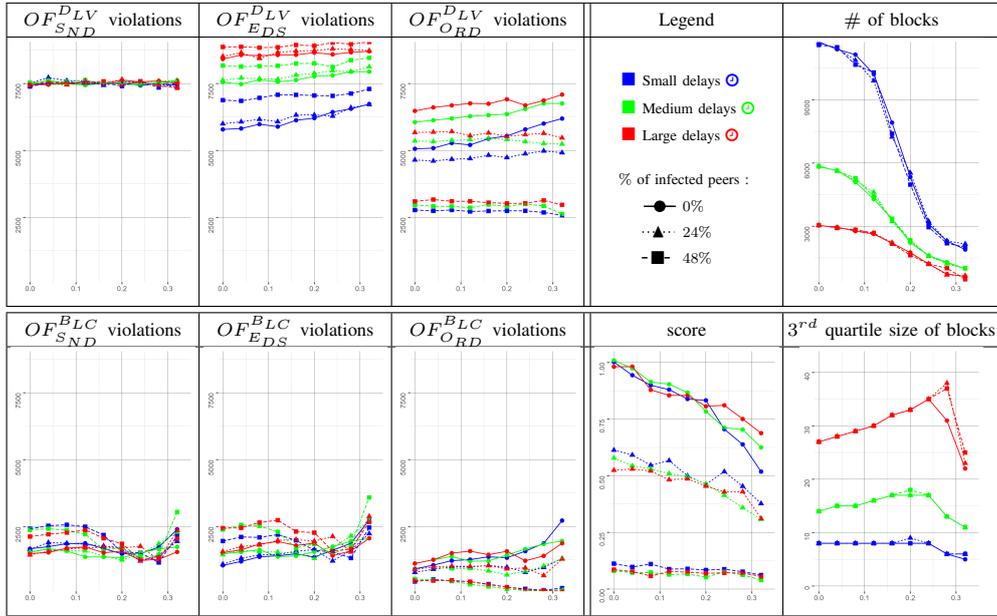


Fig. 11: Varying the number of infected Orderers

(0%, 24% and 48%), highlighting the combined effect of both attacks.

Staying below the Byzantine threshold, we observe a moderate but still significant impact the more orderers are infected. When no peers are infected (plain curves on Fig.11), the diminution of the score is directly correlated to the proportion of infected orderers (e.g., 30% of infected orderers yields a diminution of the score of around 30%). This is expected as it corresponds to the likelihood that an infected orderer is chosen as proposer. In contrast to peer sabotage, we observe that the violations of both OF_{EDS}^{DLV} and OF_{ORD}^{DLV} increase with the number of infected orderers. Indeed, this attack tamper with the delivery order but has no effect on the order of reception of transactions, w.r.t. neither the peers nor the orderers. Therefore its effect can be observed on violations of receive-order fairness w.r.t. both the peers and the orderers.

We also remark that unlike peer sabotage, which has no noticeable effect on the number and size of blocks, this is not the case for orderer sabotage. When there are more infected orderers, the number of blocks diminishes non-linearly in a S-shape sigmoid (top right diagram of Fig.11). Also, the likelihood that some blocks are quite big increases before decreasing again (bottom right diagram of Fig.11). This can be explained as follows : when there is a small proportion of Byzantine orderers, the mean time required for consensus to be reached increases as these orderers will not send PREVOTE and PRECOMMIT messages for blocks that contain transactions from the target client. Because a majority of nodes are honest, a majority of these blocks are likely to contain such transactions. As a result, the overall rate at which blocks are delivered diminishes. In turn, this lead to bigger blocks as transactions accumulate in each orderer’s local mempool. In turn, these bigger blocks are more likely to contain transactions from the target client (simply because they contain more

transactions, collected during a larger span of time). Thus the proportion of blocks which agreement are slowed by the Byzantine nodes increases, which further increases the overall diminution in the rate at which blocks are delivered. This explains the sharp decrease part of the non-linear effect (center of the sigmoid). As the proportion of infected orderers further increases, this increase in the likelihood that new block proposals contain transactions from the target client is counterbalanced by the fact that proposers are more likely to be Byzantine themselves and therefore that their proposal do not include such transactions. This then explains the last part of the non-linear effect (end of the sigmoid). This discussion also explains what is observed w.r.t. the 3rd quartile in the size of blocks. Indeed, when most block proposals include transactions from the target client, the rate at which blocks are delivered diminishes, which results in bigger blocks, more transactions accumulating in the orderers’ mempools. Once the proportion of infected orderers is past a certain point, the likelihood that block proposals contain transactions from the target client diminishes again, leading to smaller blocks.

Now that we have understood the side effect orderer sabotage has on the number and size of blocks, we can take a look at the number of OF_{SND}^{BLC} , OF_{EDS}^{BLC} and OF_{ORD}^{BLC} violations. Byzantine orderers do not include transactions from the target client in their block proposals. This trivially causes an increase in the number of OF_{SND}^{BLC} , OF_{EDS}^{BLC} and OF_{ORD}^{BLC} violations. On Fig.11, this is particularly noticeable for OF_{ORD}^{BLC} (see plain curves, with no infected peers). However, the aforementioned side effect on the size of blocks creates an inflection point in the curves (this is especially visible for OF_{SND}^{BLC}). As the proportion of infected orderers increases, the average size of the blocks increases before decreasing again. With bigger blocks, the likelihood of violating OF defined w.r.t. the partial order in which transactions are included into blocks decreases.

This explains why both OF_{SND}^{BLC} and OF_{EDS}^{BLC} decrease before increasing again.

d) *Combining peer and orderer sabotage.*: On both Fig.10 and Fig.11, we have information related to the combined effect of peer and orderer sabotage in the form of the dotted and dashed curves. We observe that the more there are infected peers and orderers, the more the score decreases. However, there are diminishing returns between the number of infected peers and orderers. Indeed, on Fig.10, when the proportion of infected peers is low, the infection of orderers has a significant effect on the score. However, closer to 50% of infected peers, we can see that there is few to no advantage in infecting additional orderers below the BFT threshold. Yet, overall, it seems that infecting peers is more efficient than infecting orderers, especially when network delays are large. This is particularly visible on the score diagram of Fig.11.

VII. A MITIGATION MECHANISM

The experiments presented in Sec.VI suggest that HF is particularly vulnerable to transaction reordering attacks. In this section, we propose a simple mitigation mechanism to make HF more robust to such attacks. In this context, improving robustness amounts to modifying the protocol so as to reduce the ability of the adversary to increase the number of OF violations (for our specific use-case, this would also limit the ability of the adversary to reduce the score).

In the following, we focus on reducing the number of OF_{EDS}^{DLV} violations. Indeed, it is not actually possible to consider a total order of send events across distant clients (as we have seen in Sec.II, upholding send-order fairness remains an open problem [6]) and reducing OF_{EDS}^{DLV} violations would be more efficient than reducing OF_{ORD}^{DLV} (as the reception orders on peers are likely closer to the send order than the reception orders on the orderers are).

Let us recall that receive-order fairness (which is the strongest OF defined w.r.t. receptions) is impossible to uphold [6]. As a result, our mitigation mechanism may, at most, reduce the number of violations (it cannot prevent all pairs of transactions to violate OF_{EDS}^{DLV} , as HF must output a total order on transactions).

Let us also recall that batch-order fairness [19] (block-order fairness in [6]), ignores part of the problem by excluding pairs of transactions that are in the same Condorcet cycle. Because we are considering OF_{EDS}^{DLV} , and because a total order must be agreed upon, mechanisms adapted from algorithms that uphold a form of batch-order fairness (see [6], [19], [15]) may not be adapted.

Our goal of “reducing the number of OF_{EDS}^{DLV} violations” is rather related to the notion of “bounded unfairness” from [37]. However, we have seen in Sec.II-C that finding a total order that minimizes the unfairness amounts to a NP-Hard problem. Practical solutions to that problem can only be approximate heuristics. Our mitigation mechanism is one such practical solution.

A. Description of the mechanism

Let us recall that, as described on Fig.5, the endorsement of a transaction corresponds to collecting a quorum of signatures (denotes as $\{s_{k_1}, \dots, s_{k_j}\}$ on Fig.5) from the peers. Upon receiving a sufficient number of signatures, a client may forward its transaction to the orderers. Concurrently, the orderers participate in repeated instances of consensus. Upon receiving an endorsed transactions, an orderer puts it in its local mempool. As a result, transactions can be progressively delivered.

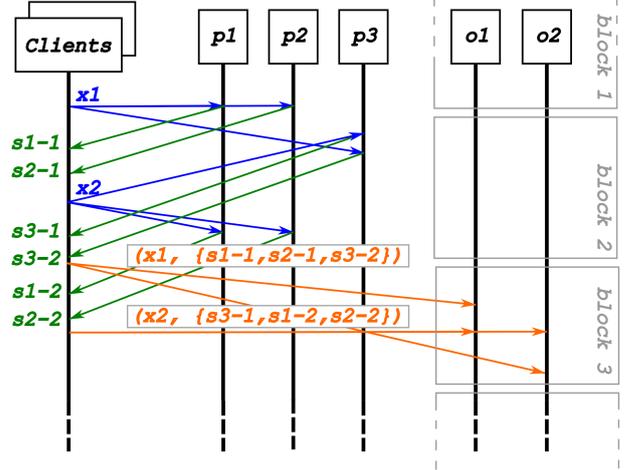


Fig. 12: Example scenario

Fig.12 describes a simple scenario in which we focus on the lifecycle of two transactions x_1 and x_2 . In this simplified system we consider that there are three peers and that the endorsement policy is such that all of them must endorse a transaction for it to be ordered. In our scenario, x_1 is sent before x_2 . However, while p_1 and p_2 do receive x_1 before x_2 , p_3 receives x_2 before x_1 . Once the corresponding has received the 3 endorsements (represented by the signatures s_{1-1} , s_{2-1} and s_{3-2}) for x_1 , it forwards it to the orderer. The same thing occurs for x_2 . There are a number of orderers that participate in repeated instances of (Tendermint) consensus. On Fig.12 only two of them : o_1 and o_2 are represented. The Tendermint protocol is continuously executed concurrently w.r.t. the endorsing process. On the right of Fig.12, the grey squares roughly represents the spans of time taken to resolve an instance of consensus. On Fig.12, we can see that both o_1 and o_2 (and we suppose this is the case for all the other orderers) receive the endorsed x_1 and x_2 during the vote of block 3. Supposing the vote is resolved at round 0, neither x_1 nor x_2 will be included in block 3. Later on, when the consensus instance for block 4 starts, the proposer selected for the first round of that instance may then include x_1 and x_2 in its proposal for block 4.

Let us then suppose that the proposer for the first round of the consensus instance for block 4 is either o_1 or o_2 . Let us also suppose that both of them are honest and, in particular, that they have no interest in manipulating the relative delivery

order of x_1 and x_2 . The default mempool implementation in Tendermint is a thread-safe FIFO queue³. As a result, if o_1 is the proposer, it will propose a block in which x_1 is ordered before x_2 , having received the endorsed x_1 before x_2 . However, if o_2 is the proposer, its block will order x_2 first (for the same reason). Thus, if o_2 is the proposer, OF_{EDS}^{DLV} will be violated for the pair (x_1, x_2) , a majority of peers having received x_1 before x_2 .

Our mitigation mechanism prevents such scenarios from occurring without any communication overhead and a slight computation overhead. We propose that each peer maintains a local counter that keeps track of the total number of transactions it has endorsed. When endorsing a transaction, a peer piggybacks the current value of its local counter in its endorsement message (the signature authenticating this value) and increments the counter. Then, when the client has received sufficiently many endorsements, the set of signatures ($\{s_{k_1}, \dots, s_{k_j}\}$ on Fig.5) that it sends to the orderers includes information about the local orders in which individual peers endorsed it.

As a result, whenever an orderer is selected to propose a block and has to propose a new value for that block, it may use this information as a guide in order to determine the order of the transactions within that block. In our case, let us consider that o_2 is the proposer and has exactly two transactions : x_1 and x_2 in its mempool at the moment it is selected to propose a block. o_1 knows that x_1 has $\{s_{1-1}, s_{2-1}, s_{3-2}\}$ as set of endorsements and also knows that s_{1-1} (resp. , s_{2-1} and s_{3-2}) implies that x_1 is the first (resp. first and second) transaction endorsed by p_1 (resp. p_2 and p_3). Similarly, o_2 infers that x_2 is the second (resp. second and first) transaction endorsed by p_1 (resp. p_2 and p_3). As a result, the orderer can infer a set of 3 ballots, one for each peer, that indicate their ranked preferences as for the ordering of the transactions.

```

Type Key ;                               /* public key*/
Type Tx ;                                 /* endorsed transaction*/
Input peers : Set<Key>;                   /* peers' keys set*/
Input pool : Set<Tx>;                     /* orderer's local mempool*/
localOrders : Map<Key, Map<Int, Tx>> ← {};
for x ∈ pool do
  | for s ∈ sigs(x) do
  | | localOrders.addOrInsert(s.key, s.idx, x);
  | end
end
ballots : Set<List<Tx>> ← {};
for (p, loc) ∈ localOrders do
  | ballot ← loc.values().sortByKeys();
  | ballots.add(ballot);
end
return ballots;
Algorithm 1: Extracting ballots from order metadata

```

Alg.1 details the process of extracting ranked preferences in the ordering of transactions from the orderer's mempool

for each peer. These preferences are extracted from the order-related metadata that is piggybacked on the endorsements of each transaction. This yields a set of ballots (one per peer) that may then be used to determine a total order between the transactions that are currently present in the orderers' mempool. Each ballot consists of a list of transactions in the order in which the corresponding peer has endorsed them. In our example from Fig.12 the ballots for p_1 and p_2 are both $[x_1, x_2]$ and the one for p_3 is $[x_2, x_1]$. However, let us remark that, depending on the endorsing policy, each transaction may only be endorsed by a fraction of the nodes. As a result, ballots may be of different lengths. In our example, if we had a fourth peer p_4 , its ballot would be the empty list as it endorsed neither x_1 nor x_2 .

In any case, this set of ballots may then be used by the orderer to compute a total order. A wide variety of voting algorithms may be applied to that end. Our only two requirements for choosing one such algorithm is that it is compatible with ranked ballots [62] and that voting allows establishing an ordered list of winners.

B. Selection of the voting algorithm

Algorithms that satisfy our two requirements include the following three families: positional voting (e.g., Borda count, Dowdall system), Condorcet voting (e.g., Copeland's method, Tideman's Ranked Pairs, Kemeny-Young method) and runoff voting (e.g., Nanson's and Baldwin's methods) [62].

Positional voting would consist here in computing scores for each transaction based on their indices in each ballot and ordering them according to these scores. Positional voting algorithms generally have a $O(n * X)$ time complexity where n is the number of peers (and therefore of ballots) and X is the number of transactions (and therefore the size of the block). These algorithms generally uphold monotonicity (increasing the rank of a transaction in some ballots should not cause a decrease in its overall ranking).

Condorcet voting would consist here in reasoning on each transaction's pairwise wins and losses, a transaction winning against another if a majority of peers ranks it higher. Such algorithms have at least a $O(n^2 * X)$ time complexity. They also uphold the Condorcet criterion, which, in our case, is particularly interesting to minimize OF_{EDS}^{DLV} violations.

Runoff voting consists in determining a single winner or loser, removing it from the set of candidates and repeating the process until no candidate is left. Some runoff voting algorithms satisfy the Condorcet criterion (e.g., Nanson's and Baldwin's methods which eliminate candidates below the average Borda score) and others do not (e.g., Bucklin, which reasons on the highest median ranking). Their time complexity also varies.

There are many parameters one ought to consider when choosing a specific voting algorithm for the orderers. Voting criteria such as the Condorcet criterion or the Majority criterion might be considered so as to reduce the probability to violate OF_{EDS}^{DLV} . However, this is not the only parameter to consider. Indeed, proposing a new block is in the critical path

³see <https://github.com/tendermint/tendermint/discussions/6295>

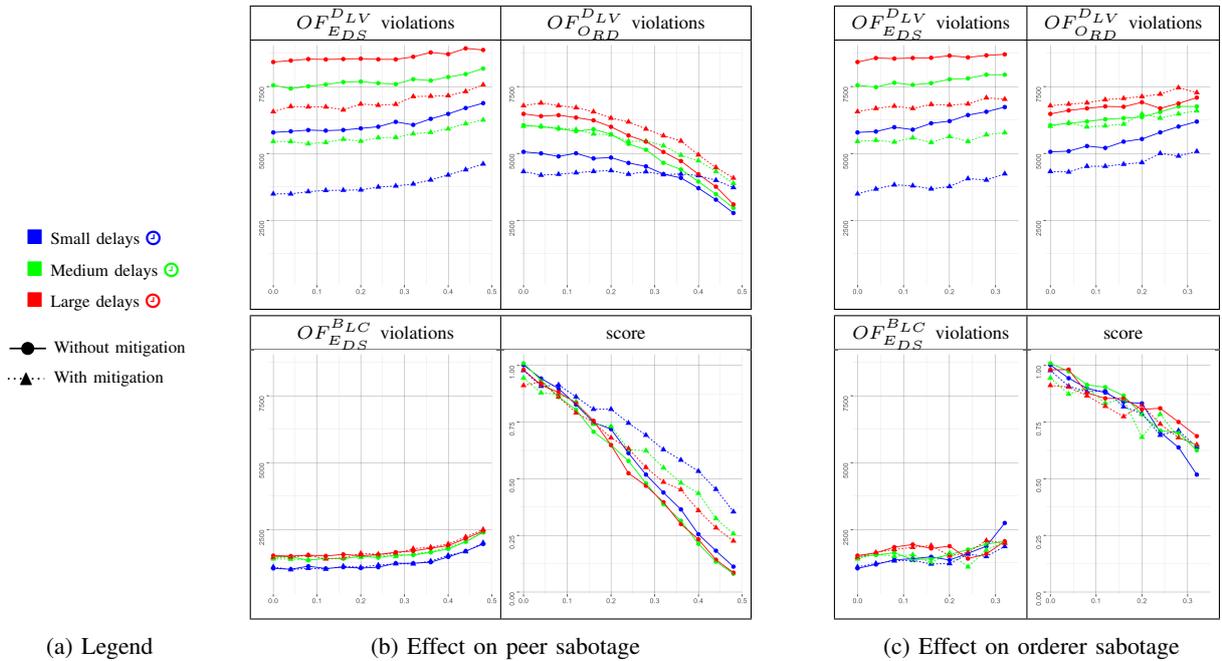


Fig. 13: Effects of the mitigation mechanism

of the HF process. Choosing an algorithm with a high time complexity could negatively impact latency. Given a block h of a certain size, determining the order of transactions within the block takes more time with one such algorithm. Moreover as the consensus for block h takes a long time, the next block proposal for block $h + 1$ may likely contain numerous transactions (which have accumulated in the next proposer’s mempool during the time it took to reach consensus for block h). As a result, the consensus for block $h + 1$ may take even more time and so on.

Another important aspect to consider is that of the manipulability of the algorithm [62], [63]. Indeed, an adversary may attempt to change the outcome of the election (and therefore the order of transactions within the next block) by coordinating the peers and clients it controls so as to manipulate the content of some ballots. For instance, the peers it controls could assign high (resp. low) indices to transactions that the adversary wants ordered first (resp. last). As for the clients it controls, they could carefully select the subsets of required endorsements that are forwarded to the ordering service (thus manipulating the ballots more directly). However, to do so, the adversary must, in a timely manner, (1) predict which transactions are likely to be part of the next block (i.e., predict the set of candidates), (2) compute a “manipulation” that would yield an order it wants to favor (the term “manipulation” having a formal meaning in e.g., [63]), (3) predict how to coordinate the peers and clients it controls so as to perform this manipulation of the ballots and (4) communicate with these peers and clients in a timely manner to coordinate them. Fortunately, as discussed in [63], the problem of computing a “manipulation” is NP-Hard for a variety of positional and runoff voting algorithms (Borda, Nanson, Baldwin). As a result, it is highly unlikely that an adversary may succeed except

if there is very few ballots (i.e., the endorsing service has only a few peers and/or the endorsing policy only requires a few endorsements) and candidates (the block proposal contains few transactions) or if it has complete control over the network under an asynchronous communication model.

In this paper, we do not recommend any specific voting algorithm. Still, simple positional voting algorithms such as Borda and Dowdall are interesting as they have a low time complexity and are difficult to manipulate in a timely manner (as per [63]). Let us also remark that, in another context, [30] uses Tideman’s Ranked Pairs to order transactions within Condorcet cycles. However, the $\mathcal{O}(n^3)$ time complexity of Ranked Pairs may be problematic for latency.

VIII. ADDITIONAL SIMULATIONS

A. Effect of the mitigation mechanism

We use MAX [17] to evaluate the effectiveness of the mitigation mechanism described in Sec.VII in making our HF system more robust to the attacks performed by the adversary. For our experiments, the orderers use the Dowdall positional voting algorithm to order transactions. We otherwise keep the same experimental setting as that of Sec.VI.

Fig.13 summarizes our experimental results. In each diagram, the plain (resp. dotted) curves correspond to simulations without using (resp. that use) the mitigation mechanism. Overall, the number of OF_{EDS}^{DLV} violations is smaller when using the mechanism (at any proportion of infected peers and orderers and under any network condition), which was its purported goal (it being defined as a vote on the ordering preferences of the peers).

However, the mechanism might increase the number of OF_{ORD}^{DLV} violations in certain cases. On Fig.11, using the mechanism causes a decrease under the smaller delays \odot

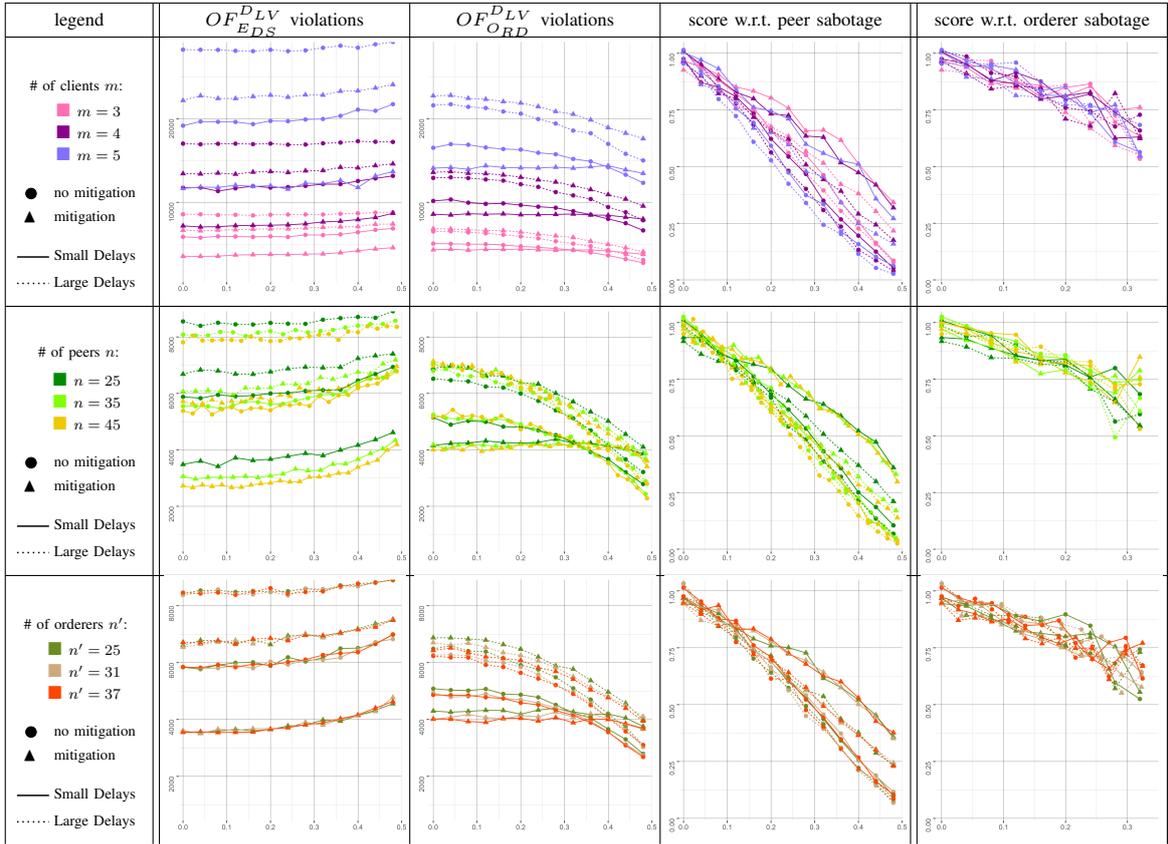


Fig. 14: Effects of varying the number of clients, peers, and orderers

in blue but an increase under the larger delays \ominus in red. Indeed, when network delays are particularly high and random, the order with which orderers receive endorsed transactions might not be correlated to the order with which peers receive transactions. Using the mechanism replaces the default FIFO order on the orderers' local mempools by an order that is computed according to the preferences of the peers. If we recall our example from Fig.12, under high network delays it may be so that a majority of orderers receive x_2 before x_1 , even though a majority of peers received x_1 before x_2 . As a result, using the default FIFO order would result in a violation of OF_{EDS}^{DLV} and complying with OF_{ORD}^{DLV} while using the mitigation mechanism would result in complying with OF_{EDS}^{DLV} and a violation of OF_{ORD}^{DLV} .

On Fig.13b, we remark that the mitigation mechanism has a protective effect against the attack on the endorsing service, especially under smaller network delays. Indeed, when using the mitigation, the decrease of the score with the number of infected peers is less pronounced. This protection is less visible against the ordering service attack except under smaller delays. Still, as peer sabotage is more efficient than orderer sabotage, overall, using the mechanism should prove advantageous.

B. Varying the numbers of sub-systems

In Sec.VI and Sec.VIII-A, we considered $n = 25$ peers, $n' = 25$ orderers and $m = 3$ clients and we varied the number

of Byzantine peers (resp. orderers) between 0 and 12 (resp. 0 and 8) so that their proportion varies between 0% and 50% (resp. 0% and 33%). As we need long simulations to make the score converge, considering higher numbers of participants involves costly simulations. These arbitrary values for n , n' and m were chosen so as to make the experiments reproducible without a prohibitive computational cost. In the following, we show that our results and remarks still hold when varying n , n' and m .

The first line of Fig.14 summarizes results from 264 distinct simulations, varying the number of clients m from 3 to 5. While the three diagrams on the left correspond to 156 simulations, varying the number of infected peers between 0 and 12 on the horizontal axis, the one on the right corresponds to 108 simulation, varying the number of infected orderers between 0 and 8 on the horizontal axis. We observe that the number of violations increases with m , as there are more pairs of non commutative transactions to consider. Still, we observe the same effect as in Sec.VI w.r.t. the power of the adversary : increasing the proportion of infected peers increases the number of OF_{EDS}^{DLV} violations and decreases the number of OF_{ORD}^{DLV} violations. Also, the effectiveness of the attack increases w.r.t. the number of clients m , which is expected, as slowing down the delivery of the transactions from the target clients is more likely to make that client loose if it has more competitors. As previously, we also remark that larger delays make the system more vulnerable while

the mitigation mechanism makes it more robust. Similarly, modifying m does not significantly change the effect of orderer sabotage.

The second line of Fig.14 summarizes results from 324 distinct simulations with the number of peers n being either 25, 35 or 45. We remark that the number of OF_{EDS}^{DLV} violations decreases with n . Indeed, increasing n reduces the variance of the time taken to obtain $n/2$ endorsements. However, increasing the proportion of infected peers still lead to an increase in OF_{EDS}^{DLV} violations. Overall the effect of peer sabotage and orderer sabotage is not impacted by modifying n . Also, the effect of the delay distribution and the mitigation system does not depend on n .

Finally, the third line of Fig.14 reports results of 288 simulations with the number of orderers n' being either 25, 31 or 37. Likewise, modifying n' does not change the effect of peers sabotage, orderer sabotage, of the distribution of delays or of the mitigation system.

IX. CONCLUSION

In [16], we have introduced a novel adversary model tailored to distributed systems and blockchain technologies. The adversary operates by performing actions that are bound by a failure and a communication model. Chaining such actions, it executes attacks within predefined fault-tolerance thresholds. This approach facilitates a more direct and fine-grained integration of adversarial behavior in practical scenarios. Furthermore, by integrating this adversary model into a multi-agent-based simulator, we enable the simulation of realistic adversarial attacks. We applied our approach on an HF-based blockchain system, simulating several attacks on a client-fairness property while evaluating their side effects on several order-fairness properties.

This present paper extends the study from [16] via considering more realistic network assumptions (e.g., hypoexponential delay distributions [60]) and a wider range of metrics. We also propose a novel mitigation mechanism (with no overhead) to make HF more robust to transaction reordering attacks [26]. we extend our simulation study to this mechanisms to demonstrate its protective effect, in particular against peer sabotage. Finally, we vary additional simulation parameters to comfort our results. Varying the number of clients, peers and orderers do not call into question the effect of the attacks from [16] and of our mitigation mechanism.

Our study thus highlights the vulnerability of HF to transaction reordering attacks. Although achieving order fairness is not realistic (as discussed in Sec.II-C and in [37]), simple mechanisms can be put in place to mitigate this vulnerability.

REFERENCES

- [1] S. Chen and Q. Song, "Perimeter-based defense against high bandwidth ddos attacks," *TPDS*, 2005.
- [2] D. Dolev and A. Yao, "On the security of public key protocols," *Transactions on information theory*, 1983.
- [3] Barbosa *et al.*, "Sok: Computer-aided cryptography," in *Symposium on Security and Privacy*, 2021.
- [4] Q. Do, B. Martini, and K.-K. R. Choo, "The role of the adversary model in applied security research," *Computers and Security*, 2019.
- [5] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distrib. Comput.*, 1987.
- [6] M. Kelkar *et al.*, "Order-fairness for byzantine consensus," in *CRYPTO*, 2020.
- [7] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," 1985.
- [8] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," 1988.
- [9] F. B. Schneider, *What Good Are Models and What Models Are Good?* ACM Press, 1993.
- [10] A. Teixeira *et al.*, "A secure control framework for resource-limited adversaries," *Automatica*, 2015.
- [11] T. Guggenberger *et al.*, "An in-depth investigation of the performance characteristics of hyperledger fabric," *Computers & Industrial Engineering*, 2022.
- [12] A. Hussain, J. Heidemann, and C. Papadopoulos, "A framework for classifying denial of service attacks," in *SIGCOMM*, 2003.
- [13] M. Shimamura and K. Kono, "Yataglass: Network-level code emulation for analyzing memory-scanning attacks," in *DIMVA*, 2009.
- [14] R. Spreitzer *et al.*, "Systematic classification of side-channel attacks: A case study for mobile devices," *COMST*, 2018.
- [15] C. Cachin *et al.*, "Quick order fairness," in *Financial Cryptography and Data Security*, 2022.
- [16] E. Mahe, R. Abdallah, S. Tucci-Piergiovanni, and P.-Y. Piriou, "Adversary-augmented simulation to evaluate order-fairness on hyperledger fabric," in *13th Latin-American Symposium on Dependable and Secure Computing (LADC24)*, 2024.
- [17] CEA LICIA, "Multi-Agent eXperimenter (MAX)," cea-licia.gitlab.io/max/max.gitlab.io/, 2022.
- [18] R. Paulavicius, S. Grigaitis, and E. Filatovas, "A systematic review and empirical analysis of blockchain simulators," in *Access*. IEEE, 2021.
- [19] M. Kelkar, S. Deb, S. Long, A. Juels, and S. Kannan, "Themis: Fast, strong order-fairness in byzantine consensus," in *2023 ACM SIGSAC Conference on Computer and Communications Security (CCS23)*, 2023.
- [20] Y. Amoussou-Guenou *et al.*, "Dissecting tendermint," in *NETYS*, 2019.
- [21] V. Bushkov and R. Guerraoui, "Safety-liveness exclusion in distributed computing," in *2015 ACM Symposium on Principles of Distributed Computing (PODC15)*, 2015.
- [22] Y. Amoussou-Guenou, A. D. Pozzo, M. Potop-Butucaru, and S. Tucci Piergiovanni, "On fairness in committee-based blockchains," in *2nd International Conference on Blockchain Economics, Security and Protocols, Tokenomics 2020*, 2020.
- [23] S. Müller, A. Penzkofer, D. Camargo, and O. Saa, "On fairness in voting consensus protocols," in *Intelligent Computing*, K. Arai, Ed., 2021.
- [24] Y. Huang, J. Tang, Q. Cong, A. Lim, and J. Xu, "Do the rich get richer? fairness analysis for blockchain incentives," in *2021 International Conference on Management of Data (SIGMOD21)*, 2021.
- [25] K. Lev-Ari, A. Spiegelman, I. Keidar, and D. Malkhi, "FairLedger: A Fair Blockchain Protocol for Financial Institutions," in *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, 2020.
- [26] L. Heimbach and R. Wattenhofer, "Sok: Preventing transaction reordering manipulations in decentralized finance," in *4th ACM Conference on Advances in Financial Technologies (AFT22)*, 2023.
- [27] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.
- [28] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, 1990.
- [29] J. Garay and A. Kiayias, "Sok: A consensus taxonomy in the blockchain era," in *Topics in Cryptology (CT-RSA 2020)*, 2020.
- [30] M. A. Vafadar and M. Khabbaziyan, "Condorcet Attack Against Fair Transaction Ordering," in *5th Conference on Advances in Financial Technologies (AFT 2023)*, 2023.
- [31] N. Andola *et al.*, "Vulnerabilities on hyperledger fabric," *Pervasive and Mobile Computing*, 2019.
- [32] D. Malkhi and P. Szalachowski, "Maximal Extractable Value (MEV) Protection on a DAG," in *4th International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2022)*, 2023.
- [33] P. Momeni, S. Gorbunov, and B. Zhang, "Fairblock: Preventing blockchain front-running with minimal overheads," in *Security and Privacy in Communication Networks*, 2023.

- [34] R. L. Rivest, A. Shamir, and D. A. Wagner, "Time-lock puzzles and timed-release crypto," MIT, Tech. Rep., 1996.
- [35] J. Burdges and L. De Feo, "Delay encryption," in *40th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT2021)*, 2021.
- [36] A. Shamir, "How to share a secret," *Commun. ACM*, 1979.
- [37] A. Kiayias, N. Leonardos, and Y. Shen, "Ordering transactions with bounded unfairness: Definitions, complexity and constructions," in *Advances in Cryptology (EUROCRYPT 2024)*, 2024.
- [38] A. Misra and A. D. Kshemkalyani, "Byzantine fault-tolerant causal ordering," in *24th International Conference on Distributed Computing and Networking (ICDCN23)*, 2023.
- [39] O. Damani and V. Garg, "How to recover efficiently and asynchronously when optimism fails," in *16th International Conference on Distributed Computing Systems*, 1996.
- [40] B. Simons, "An overview of clock synchronization," in *Fault-Tolerant Distributed Computing*, 1990.
- [41] J. Martin, J. Burbank, W. Kasch, and D. Mills, "Network Time Protocol Version 4: Protocol and Algorithms Specification," RFC 5905, 2010.
- [42] R. Ganguly, A. Momtaz, and B. Bonakdarpour, "Distributed Runtime Verification Under Partial Synchrony," in *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*, 2021.
- [43] S. Duan, K. N. Levitt, H. Meling, S. Peisert, and H. Zhang, "Byzid: Byzantine fault tolerance from intrusion detection," in *33rd IEEE International Symposium on Reliable Distributed Systems, SRDS 2014*, 2014.
- [44] P. Aublin, S. B. Mokhtar, and V. Quéma, "RBFT: redundant byzantine fault tolerance," in *IEEE 33rd International Conference on Distributed Computing Systems, ICDCS 2013*, 2013.
- [45] R. Jain, D. Chiu, and W. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems," 1984.
- [46] 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 20)*, 2020.
- [47] K. Kursawe, "Wendy, the good little fairness widget: Achieving order fairness for blockchains," in *2nd ACM Conference on Advances in Financial Technologies (AFT20)*, 2020.
- [48] R. Cramer *et al.*, "Efficient multiparty computations secure against an adaptive adversary," in *EUROCRYPT*, 1999.
- [49] A. De Santis *et al.*, "How to share a function securely," in *STOC*, 1994.
- [50] Y. Xiao *et al.*, "Modeling the impact of network connectivity on consensus security of proof-of-work blockchain," in *INFOCOM*, 2020.
- [51] A. Miller and R. Jansen, "Shadow-Bitcoin: Scalable simulation via direct execution of Multi-Threaded applications," in *CSET. USENIX*, 2015.
- [52] A. Deshpande, P. Nasirifard, and H.-A. Jacobsen, "evibes: Configurable and interactive ethereum blockchain simulation framework," in *Middle-ware. ACM*, 2018.
- [53] J. Ferber, O. Gutknecht, and F. Michel, "From agents to organizations: An organizational view of multi-agent systems," in *Agent-Oriented Software Engineering IV*, 2004.
- [54] A. Dabholkar and V. Saraswat, "Ripping the fabric: Attacks and mitigations on hyperledger fabric," in *ATIS*, 2019.
- [55] B. Putz and G. Pernul, "Detecting blockchain security threats," in *IEEE International Conference on Blockchain*, 2020.
- [56] S. D. Angelis *et al.*, "Evaluating blockchain systems: A comprehensive study of security and dependability attributes," in *DLT at ITASEC*, 2022.
- [57] J. Barwise, "An introduction to first-order logic," 1977.
- [58] N. Borisov, I. Goldberg, and D. Wagner, "Intercepting mobile communications: The insecurity of 802.11," in *MobiCom*, 2001.
- [59] D. Ray and J. Ligatti, "Defining code-injection attacks," in *POPL*, 2012.
- [60] R. Wallace, X. G. Andrade, P. Kayser, Z. Luo, H. Mukherjee, R. Nunes, and M. Warrior, "Models of network delay," in *Developments in Statistical Modelling*, 2024.
- [61] E. Mahe, "Extended order fairness experiments on hyperledger fabric & tendermint," gitlab.com/cea-licia/max/models/experiments/max.model.experiment.fabric_tendermint_of_exp_with_mitigation, 04 2025.
- [62] F. Brandt, V. Conitzer, U. Endriss, J. Lang, and A. D. Procaccia, *Handbook of Computational Social Choice*, 1st ed. USA: Cambridge University Press, 2016.
- [63] J. Davies, G. Katsirelos, N. Narodytska, T. Walsh, and L. Xia, "Complexity of and algorithms for the manipulation of borda, nanson's and baldwin's voting rules," *Artificial Intelligence*, 2014.