Secure Consistency Verification for Untrusted Cloud Storage by Public Blockchains

Regular Paper

ABSTRACT

This work presents ContractChecker, a Blockchain-based security protocol for verifying the storage consistency between mutually distrusting cloud provider and clients. Unlike existing approaches, the ContractChecker uniquely delegates log auditing to the Blockchain, and has the advantages in reducing client cost and lowering requirements on client availability, lending itself to emerging security applications.

New attacks are proposed that exploits the limitation of practical Blockchain systems including write unavailability, contract race conditions, and Blockchain forks. ContractChecker leverages proposed countermeasures to close the attack vectors and ensure the correctness of consistency assertion under attacks.

We build a consistency verification service based on ContractChecker which charges service fee for running a smart contract on Ethereum. We evaluate the client cost of ContractChecker which is lower than the prior state-of-the-art by orders of magnitude. While our techniques improve the cost-effectiveness of Blockchain, the current system is practical mainly for low-frequency data storage.

1. INTRODUCTION

In today's cloud storage services (e.g., Dropbox [4] and Amazon S3 [1]), storage consistency is a pressingly important security property, especially in an age that cloud incidents become the norm. The consistency of a data-storage service dictates how reads/writes should be ordered and whether a read should reflect the latest write (i.e., freshness). This property can be easily exploited by a malicious cloud provider and leads to severe security consequences. For instance, when the cloud storage hosts a pubic-key directory (as in certificate-transparency schemes [2]), a malicious cloud violating storage consistency can return to the user a stale key (i.e., a revoked public key), which leads to consequences like impersonation attacks and unauthorized data access. Assured storage consistency is fundamental and critical to supporting many security infrastructures in the cloud (e.g., DNS server, Github/DockerHub, personal data hosting).

The problem of asserting remote storage consistency entails two logical steps: W1) establishing a globally consistent view of operation log across clients (i.e., operation log attestation), and W2) auditing the log for checking consistency conditions (i.e., log auditing). To the best of our knowledge, all existing approaches including Caelus [26], CloudProof [34], and Catena [37], rely on the clients to collectively conduct the log auditing (step W2). More specifically, they rely on a trusted third-party log attester (step W1), and sends the attested global log to individual clients, each of which audits the global log against her local operations. Catena [37], particularly, leverages the Blockchain as the TTP log attester, and uses clients as the log auditor. These client-based log auditing schemes incur requirements on high client availability (i.e., all clients have to participate in the protocol execution) and high client cost (i.e., a client needs to store the global operation log of all other clients), rendering them ill-suited for the target applications with a large number of ad-hoc clients; see § A.1 for a list of target applications.

We propose ContractChecker, a security protocol for Blockchain-based consistency verification. Distinct from existing works, ContractChecker uniquely delegates both log attestation (W1) and log auditing (W2) to the Blockchain. Concretely, ContractChecker runs a program on the Blockchain (i.e., the so-called smart contracts) to collect log attestations from clients and the server and to audit the log for consistency assertion. Comparing existing works, our new approach, by delegating log auditing to Blockchain, has the advantage in lowering client availability requirement (i.e., only active clients who interacts the cloud are required to participate in the protocol) and in minimizing client overhead (i.e., a client only maintains her own operations for a limited period of time).

Designing the security protocol of ContractChecker against 1) mutually untrusted clients and server (detailed in § 5.2) and 2) exploitable Blockchain systems in the real world (detailed in § 5.3) is non-trivial. First, a ContractChecker party, be it a client or the server, can be malicious and forge operations in her log attestation in order to trick ContractChecker to make an incorrect assertion about the operation consistency. We propose to cross-check the server attestation and client attestations on Blockchain to both detect and mitigate the attacks from malicious clients and the server, such that the ContractChecker makes a correct assertion in presence of the attacks.

Second, real-world Blockchain systems are known to be limited in terms of E1) write unavailability (i.e., a valid transaction could be dropped by the Blockchain), E2) exploitable smart-contract race conditions (i.e., incorrect contract logic can be triggered by running contract code concurrently), and E3) Blockchain forks (i.e., a Blockchain network or data structure can be forked to multiple instances). We propose attacks against the use of Blockchain in the context of ContractChecker. These new attacks systematically exploit the Blockchain limitations mentioned above, and they are: 1) the selective-omission attack exploiting Blockchain write unavailability (E1), forking attacks exploiting smart-contract race conditions (E2), and another version of forking attack exploiting Blockchain forks (E3).

We propose countermeasures to defend the Blockchain-oriented ContractChecker attacks. To prevent write unavailability (E1), we propose a client mechanism to retry transaction submission with increasing transaction fees, such that the transaction submission is bound to succeed within a certain period of time. To prevent the race conditions on ContractChecker (E2), we define a small critical section in the contract to secure the contract execution yet without losing the support of concurrent server and client attestations. We prevent forking through Blockchain forks (E3) by ensuring all clients and server be aware of the presence of all Blockchain forks.

We build a prototype materializing the ContractChecker protocol with Ethereum [5]. For the on-chain part, we write a Solidity program that collects the log attestations from clients and server, crosschecks them and audits the log by checking consistency conditions. During the log auditing, the on-chain contract accesses the historical operations stored off-chain and data authenticity is assured by a Merkle tree maintained by the server.

We conduct cost analysis and experimental analysis on our prototype. We evaluate the client cost of the protocol and service under YCSB workloads [20]. Our results show that on Ethereum, ContractChecker results in significant cost saving on the client side, with reasonable Gas cost on running smart contract. The off-chain placement saves up to 80% of the cost comparing an on-chain placement. ContractChecker can support 613710 operations at the cost of \$100 at a real Ether price.

The contributions of this work are:

1. **New use of Blockchain**: This work proposes a new use of Blockchain for cloud-storage verification. Unlike existing research, the work delegates both log attestation and log auditing on Blockchain which reduces the client cost for consistency verification (§ 2.1).

2. **Security protocol**: New security attacks are identified due to the new use of Blockchain, and new security protocols that schedule the on-chain events for consistency checking are proposed (§ 4).

3. **Prototyping and evaluation**: We implement a ContractChecker prototype on Ethereum. Through cost analysis and YCSB-based experiments, we show the cost-effectiveness of the ContractChecker protocol (§ 6).

2. PRELIMINARIES

2.1 Secure Consistency Verification Protocols

Verifying the consistency of untrusted storage entails two steps: log attestation and log auditing. In the step of log attestation, the untrusted server attests to the global log against a trusted attester and the clients verify the correctness of log attestation. The purpose of having the attester is to ensure there is a single view of the global log and to prevent the server from mounting the forking attack [31]. Once a global log (without fork[1]) is constructed, the second step can take place which is to check the consistency conditions over the established global history.

SUNDR [28] is a secure log attestation scheme without trusted third party. SUNDR directly sends the server attestation to individual clients, which can be seamlessly and elegantly integrated to the data plane of serving reads/writes. SUNDR achieves the level of fork consistency that any forking attack will be eventually detected. Caelus [26] is a log attestation scheme that rotates (or time-share) the role of attester among a group of personal devices. Caelus runs an epoch-based protocol for log attestation. Caelus audits the log and checks a variety of consistency models, by requiring each client to download the global operation history. Similarly, CloudProof [34] relies on trusted data owners to attest to the server's claim of log and to audit the operation history. Catena [37] is a log-attestation scheme based on Blockchain. Catena prevents the server forking attack by running an auditing pass on clients. In Catena, the security of preventing forking attacks is reduced to the Blockchain's security in preventing double-spending attacks.

All existing works rely on trusted clients (or data owners) to audit the log and check consistency conditions, as illustrated in Table 1. In this respect, ContractChecker is the first scheme that delegates both log attestation and log auditing to the trusted-third party Blockchain.

Table 1: Distinction of the ContractChecker design: ✗ means the log operation is not supported by Blockchain, but instead by clients.

Solutions	Attestation by Blockchain	Auditing by Blockchain
SUNDR [28], Caelus [26], CloudProof [34]	✗	✗
Catena [37]	✓	✗
ContractChecker	✓	✓

2.2 Blockchain and Smart Contract

Blockchain is the backend technology in today's cryptocurrencies, such as Bitcoin and Ethereum. This work considers the public Blockchain over the Internet, and do not consider permissioned Blockchains over a private "consortium" of players. Physically, a public Blockchain runs in a large-scale peer-to-peer network, where peer nodes, called miners, collectively maintain Blockchain data structures. The P2P network is open-membership in that anyone on the Internet can join.

We now present the background of Blockchain system from two perspectives that are most relevant to our work: Blockchain as a distributed ledger and Blockchain as a smart-contract execution platform.

Distributed ledger: Blockchain is a distributed ledger that stores the history of "transactions" for cryptocurrency payment. The Blockchain ledger can be viewed as storage that is publicly readable and is writable by decentralized authorities namely miners. In particular, writing a transaction to Blockchain goes through the following pipeline: 1) The transaction is broadcast to all miners, pending in their memory pools. 2) Pending transactions are selected (e.g., by transaction fee) to be validated based on some prescribed rules (e.g., no double spending transactions). 3) All Blockchain nodes run a consensus protocol to decide which valid transactions are to be included next in the Blockchain ledger. Following this pipeline, every "block time" B, the Blockchain is expected to produce a new block of accepted transactions.

Writing a transaction to a Blockchain is *asynchronous* in the sense that it takes a long delay to confirm if the transaction is finally included in Blockchain. More precisely, only after there are F blocks produced after the transaction, the transaction is considered finalized on the Blockchain. That is, it will be hard to change the transaction state on Blockchain. Writing a transaction to a Blockchain can *fail* in the sense that the Blockchain can drop valid transactions in some circumstances. For example, when the transaction throughput is larger than Blockchain's throughput limits[2] or transaction fee is too low, Blockchain will drop transactions.

[1]Note that the forking attack differs the Blockchain forks caused by software updates.

[2]Public Blockchains are considered as low-throughput systems that

In other words, Blockchain may have low write availability in the above conditions.

Contract execution: Blockchain is also an execution platform for smart contact. Smart contract is a program that can be executed in modern Blockchains, such as Ethereum and that is originally proposed to support financial applications over Blockchain. To execute a smart contract, the contract author writes a program, compiles it to bytecode and deploys it to the Blockchain by encoding the compiled bytecode in a special Blockchain transaction. Once the contract is deployed on all Blockchain miners, a party off-chain will be able to trigger the contract execution by sending another transaction encoding the runtime arguments. Internally, a smart contract is executed as a replicated state machine on Blockchain miners. That is, the execution instances are spawn and replicated across all miner nodes. Running a contract function is a state transition to the Blockchain where the begin and end contract states are encoded in two blocks in Blockchain and all miners running the contract race to include the execution result in the end block.

In public Blockchain, a fundamental security assumption is that the majority of miners are honest nodes in the sense that they execute the Blockchain software by honestly following the prescribed protocol. In the case of 51% attacks where the majority of miners are malicious, many Blockchain security properties are broken. In practice, compromising 51% of miners in a large-scale Blockchain network is extremely hard and we believe this is reasonable to assume an honest majority. There are more sophisticated attacks, such as selfish mining [23], that require only 33% malicious miners, but they never happen in practice and we do not consider them in this paper.

Blockchain forks: This work focuses on practical forks in real-world public Blockchains, and excludes theoretic Blockchain forks (due to 51% attacks, selfish mining, etc.). There are two types of Blockchain forks: a transient fork among miners on recently found blocks, and a permanent fork among different Blockchain networks. The transient fork (the former) in a public Blockchain is eventually resolved after the finality delay when all miners reach consensus on one fork while other forks are orphaned. The permanent fork (the latter) is caused by the "hard fork" in Blockchain software updates (e.g., the case of bitcoin and bitcoin cash), launching an Alt coin (i.e., by running Blockchain software among a group of friends), etc. It is found common in practice.

Blockchain cost includes the fee to send a transaction and the cost of running smart-contracts. The higher a transaction fee is, the more likely the transaction will be included in the next block. For the smart-contract, any contract on chain needs to be associated with a cost budget, called "Gas", which bounds the execution time of the contract and is a mechanism for the defense of DoS (denial of service) attacks.

3. RESEARCH FORMULATION

3.1 Target Applications

The target application scenarios of ContractChecker are characterized by the following properties: S1) The clients are of limited capability in computing, storage, and availability. This motivates them to outsource data storage to the more powerful cloud. S2) Violating storage consistency leads to security consequences. S3) The data load is low (e.g., typically lower than tens of operations per second). In particular, the low throughput properties of these typical application scenarios make it amenable for the use of Blockchain, which is known to have limited throughput in ingesting transactions.

accept just tens of transactions per second

In real world, there are many application scenarios that fit the above paradigm. As an example, consider DockerHub [3] style container registry distributes software patches for mobile apps. In this scenario, the clients are low-power smart phones (S1). Distributing a stale software image with unfixed security bugs leads to vulnerability on users' phone (S2). In terms of workloads, an IBM report [17] shows that among seven geo-distributed registry deployments, the busiest one serves only 100 requests per minute for more than 80% of time(S3). We believe there are many real-world applications that meet our target scenarios, ranging from Google's certificate-transparency logs [2], iCloud style personal-device synchronization [8], Github-hosted secure software development [38, 7], etc.

3.2 System and Security Model

Our system model consists of three parties: a storage server in the cloud, multiple concurrent clients and the Blockchain. Clients submit storage operations (i.e., data reads and writes) to the storage server. The operations are witnessed by the Blockchain via our ContractChecker protocol.

3.2.1 Clients

Clients are data owners who outsource the data storage to the cloud. They submit operations to the cloud storage to read or update their data. Different clients' operations may occur concurrently, that is, the cloud service may process multiple operations in parallel, with their time intervals overlapped. A client can be active when she has submitted operations recently (we will explain the definition of active clients later) and inactive when she goes offline without any operations on the cloud.

Clients in our model are stateless in the sense that a client does not persist state across active sessions. That is, a client does not need to remember her past operations when she is engaged with a cloud. In addition, among clients, there are no direct communication channels. That is, other than the cloud or Blockchain, clients do not communicate out-of-band. We believe this model reflects low-power clients in many real-world situations. For example, in ContainerRegistry and CT log, clients can be web browsers running on smart phones. Web clients are stateless and do not directly communicate among themselves.

We assume each client is identified by her pseudonymous ID, say her public key. A client knows all other clients' public keys. We assume a protected communication channel or a trusted PKI offline to securely distribute public keys. The client identity management is described in more details in in Appendix § **??**.

We assume clients share synchronized clocks by running NTP protocols. Synchronized clocks will be useful in our protocol when clients are required to log operations. The accuracy of NTP protocols may affect the precision of our consistency verification results. Existing protocols can achieve highly synchronized clocks and limit clock skews to the level of milliseconds [21], which we believe are sufficient in our system.

3.2.2 Cloud storage service

The cloud service hosts a data store shared among multiple clients. The service accepts clients' requests and processes them concurrently. That is, different operations from clients may be processed in parallel and in an out-of-order fashion.

Under this execution model, we consider strong consistency or linearizability [25]. Linearizability considers independent access to different data records and is sufficiently strong to capture necessary conditions in many security scenarios [26, 34]. Weaker consistency that is widely adopted in modern cloud services [35] is commonly less relevant to security-sensitive and critical applications. In this paper, we do not consider database serializability or isolation under multiple keys [18].

We assume the cloud service makes a consistency-centric service-level agreement (SLA) [36] with the clients which states where the cloud service promises to guarantee strong consistency (more specifically, linearizability as will be defined next) over the operations it will serve. An honest server will enforce the promised consistency conditions during operational hours.

A malicious storage server will violate the consistency conditions (as will be elaborated on), by returning a stale result for a read. In addition, a malicious server will not follow our ContractChecker protocol and tries to conceal the operation inconsistency from clients. Consistency violation, if left undetected, can lead to severe security consequences such as impersonation attacks (recall § A.1).

Moreover, we assume the cloud service is rational. While a malicious cloud server may forge operations to make an inconsistent operation history look like consistent (i.e., concealing the inconsistency), the server will not attempt to make a consistent operation history look like inconsistent. We believe this assumption reflects practical situations with SLA where the cloud server will be charged in case of verified inconsistency, and thus does not have the incentive to forge an inconsistent history.

Consistency definition: We consider the storage service exposes a standard key-value API, where each data record is a key-value pair and each storage operation accesses the target record by the data key. A storage operation, be it a read or write, is specified by two timestamps, which respectively represent 1) the begin time when the request of the operation is sent by the client, and 2) the end time when the response of the operation is received by the client. Formally, given a key-value pair $\langle K, V\rangle$, a read operation is $V = r_{[t_b,t_e]}(K)$ and $\texttt{ACK} = w_{[t_b,t_e]}(K, V)$. Here r/w denotes read or write operation. $t_b < t_e$ and they are the begin and end timestamps. If two operations are concurrent, their time intervals $[t_b, t_e]$ may overlap.

An operation history is linearizable if the following two conditions are met: 1) All operations can be mapped to a total-order sequence that is compatible with the real-time intervals of the operations. 2) There is no stale read on the total-order sequence. That is, any read operation should return the record that is fresh on the total-order. This property is also called read freshness.

Any linearizable operation history can be represented by a totally-ordered operation sequence. We denote an operation sequence by operation indices. For instance, in $w_1 w_2 r_3[w_2]$, the subscription is the operation index in the total-order (described below). Linearizability specifies operations of the same data key; for simplicity, we omit data key in this notation. The square bracket of a read indicates the result record. $r_3[w_2]$ denotes a read operation ordered as the third operation in the total-order sequence and which returns the record written by write w_2.

Network: In this work, we assume a reliable network among clients, the server and Blockchain nodes. When one party sends a message to the network, the network will deliver the message to the receiver with negligible delay (comparing with the period our protocol runs). We don't consider network faults or partition in this work.

3.2.3 Blockchain

The Blockchain in our protocol is a public, permissionless Blockchain running over a large P2P network and supporting smart contract execution. Real-world examples include Ethereum and the latest version of Bitcoin. In this setting, we assume the honest majority among Blockchain miners. We believe this is a reasonable assumption as in practice there are no successful 51% attacks on Bitcoin or Ethereum.

The Blockchain in our protocol is parameterized by the following arguments: block time B, which is the average time to find a block in a Blockchain; and transaction-validation delay P, which is the time between a pending transaction enters the memory pool and when it lefts for transaction validation; and finality delay F, which is the number of blocks needed to be appended after a finalized transaction. That is, a transaction is considered to be finalized in Blockchain when there are at least F blocks included in the Blockchain after the transaction.

The Blockchain supports the execution of smart contracts. The contract is executed across all Blockchain miners. It guarantees the execution integrity and non-stoppability. That is, the Blockchain truthfully executes a smart contract based on the provided arguments, despite of the attacks to subvert the Blockchain (as described below). Once the execution of a contract starts, it is hard to stop the execution or to abort.

Blockchain write availability: The Blockchain may drop transactions based on its current load and transaction fee. A transaction with a higher fee will have a lower chance of being dropped [37] and have shorter latency to be included. This assumption will be tested in our evaluation.

Blockchain attacks: The Blockchain is subject to a series of threats. In this work, we focus on practical Blockchain threats and exclude theoretic threats (e.g., 51% attacks, selfish mining, etc.) and off-chain threats (e.g., stealing wallet coins and secret keys). 1) The Blockchain may be forked permanently due to software update as in the case of Bitcoin cash (Blockchain forks). 2) Smart contacts may contain security bugs that can be exploited by a vector of attacks [6], such as reentrancy attacks, buffer overflow attacks, etc. In practice, the DAO (decentralized autonomous organization) incidents are caused by this attack vector.

3.3 Goals

3.3.1 Security Goals

In our system, there are two main threats: 1) In the case that inconsistency occurs, the malicious server wants to hide the inconsistency from victim clients. 2) In the case that all operations are consistent, a malicious client may want to accuse the benign server of the storage inconsistency that does not occur. We exclude other cases as they are not rational. For instance, we don't consider that a rational server will forge operations such that a consistent operation history will appear inconsistent. We also don't consider that a victim client will want to hide an inconsistent operation history. Due to this reason, we assume clients and the server will not collude to either hide inconsistency or accuse of false consistency.

Our security goals are listed below:

Timely detection of inconsistency against malicious server: A malicious server cannot hide the occurrence of inconsistent operations from victim clients. To be concrete, when inconsistency occurs, the protocol will assert there are inconsistent operations in a timely manner, even when a malicious server can forge a seemingly consistent log. In addition to that, the protocol will present a verifiable proof of the inconsistency, so that it can penalize the cloud service violating the consistency.

No false accusation of inconsistency against malicious clients: A malicious client cannot falsely accuse a benign server of inconsistency that does not occur. Given a consistent operation history, the protocol will assert it is consistent, even when there are malicious clients who want to forge inconsistent operations in the log.

3.3.2 Cost Goals

Client cost: Clients in our protocol can remain stateless. That is, running the ContractChecker protocol, a client does not need to store her operations indefinitely and can truncate operations after use (i.e., stateless). In addition, the protocol data stored on a client is limited to the client's local operation; a client does not need to store other clients' operations (i.e., local operations). These two requirements make clients lightweight and render the protocol applicable to scenarios with low-power clients.

3.3.3 Non-goals

Data authenticity: Data authenticity states a malicious cloud service cannot forge a client's data without being detected. We assume an external infrastructure in place that ensures data authenticity, such as MAC or digital signatures. To be more specific, the client who writes a record signs the record, and the signature can be verified to guarantee the data authenticity of the record. Here, clients' public keys are securely distributed by a PKI or secure communication channel.

Collusion: In ContractChecker, we aim at security guarantees against a malicious cloud server or malicious clients. However, we do not consider the case that a server colludes a client, because this would make it impossible for any third-party to detect the attack. To be concrete, a colluding client can lie about her log (e.g., omitting an operation), and the server can do the same. With the client and server both lying about their operations, any third-party cannot detect the existence of the lie. If an operation can be forged or omitted in a log, the consistency assertion over the log cannot be trusted. For instance, omitting w_2 in $w_1 w_2 r_3 [w_1]$ leads to incorrect consistency assertion. Due to this reason, we exclude from our threat model the collusion among clients and server. In practice, we believe server-client collusion is a rare situation.

4. THE SECURITY PROTOCOLS

4.1 Design Motivation

The key design in ContractChecker is to use Blockchain to audit the log in a client-server system (recall Table 1). This use of Blockchain is motivated by our observations below:

1) Low cost on stateless clients: By delegating log audit to the Blockchain, one can significantly relieve the burden from the clients. Existing consistency-verification protocols are based on client-side log auditing [37, 41], which requires a client to persist not only her own operations but also operations of all other clients, rendering it impractical for stateless clients. Delegating log auditing to the Blockchain, clients are relieved from storing any state about operations. Ideally, a client's job is reduced to receiving log-audit results from the Blockchain.

2) Security without client availability: To guarantee the protocol correctness under attacks, existing client-based schemes (e.g., Caelus) assume highly available clients, that is, they have to be online every epoch to participate in the periodic log auditing (see § 5.1 for detailed analysis). Briefly, without highly available clients, it cannot distinguish the case of an inactive client and the case of an active client reporting an attack but whose report gets suppressed by the untrusted server.

Our work is motivated by that Blockchain can be used as a trusted "communication channel" to report attack detection without assuming client availability. As we will see (in § 5.1), the proposed protocol allows an inactive client (who did not submit any operation in a given epoch) to be offline and not to participate in the protocol execution. Intuitively, this is realizable because the trusted communication enabled by Blockchain will not omit messages among clients.

4.2 Protocol Overview

ContractChecker is a protocol that runs among clients, a storage server and the Blockchain. The purpose of the protocol is to present a trustworthy assertion about the consistency of the operations among clients and the server. At a high level, the protocol works by asking clients and the server to attest to their views of operations to the Blockchain where different views are crosschecked and consistency conditions are verified. An attack from a malicious server (or client) by forging her view of the log can be detected and mitigated by the crosscheck on Blockchain.

In our protocol, the interaction between the Blockchain and the clients/server off chain can be described by the following three functions. Note that the protocol runs in epochs and here we consider the operations in one epoch.

- S.`attestServerLog`(Ops_S, sk_S): Server S declares a total-order over the global history of operations in the current epoch (Ops_S). She attests to the declared total-order sequence by signing it with her secret key sk_S.
- C.`attestClientLog`(Ops_C, sk_C): An active client C attests to her local operations in the current epoch (Ops_C) by signing it with her secret key sk_C. Operations of a single client can also be concurrent and a client does not need to declare a total-order over her own operations. If a client is inactive, meaning she did not submit any operation in the epoch, she does not need to call this function.
- C.`consistencyResult()` $= \{Y, N\}$: A client C can check consistency of the current operation history. With Blockchain, the checking result is only valid when the attestations are finalized on the chain. This function is blocking until the finality is confirmed. If the attestations are not finalized, it will retry by calling `attestClientLog`.

4.3 Design Alternatives

We present alternative protocol designs that we encountered, in order to examine the design space and to justify our design choices.

No server attestation: In our protocol, both clients and the server attest to their views of operation histories. An alternative design is to collect the operation history only from clients. In this scheme, all clients send their local operations to the Blockchain without collecting the server log. The union of client logs can reconstruct all operations and their relations in real time (i.e., serial or concurrent with each other). However, it does not have the total-order information necessary to determine the operation consistency (Recall § 3.2.2). Relying on the smart contracts to "solve" the consistency total-order out of concurrent operation history, it would be an expensive approach. Therefore, we rely on the (untrusted) server to declare the consistency total-order in her attestation to the Blockchain. In addition, without server attestation, the protocol will be vulnerable under malicious clients. Also note that Catena has studied the design choice without client attestations.

Use Blockchain as data storage: In our protocol, all operations are uploaded to the Blockchain. An alternative design is to use the Blockchain as the actual data storage. However, Blockchain is ill-suited to be data storage serving online operations. First, Blockchain incurs multiple rounds of confirmations for persisting data, leading to perceivable write latency. Second, when operations access large data records (e.g., multimedia data), storing them directly on Blockchain incurs a high cost. Therefore, in our work, we use Blockchain as a log auditor off the critical path. It records operations at a certain frequency (e.g., every 10 minutes) and only publishes the record hashes[3] to the Blockchain.

Synchronize contract functions: Conceptually, our high-level goal is to delegate the work of log auditing to the Blockchain. The work consists of log attestation, log crosscheck and consistency checking. For correctness, these three steps have to be run serially. Our original design is to implement the three steps in three "public" contract functions which can be called from off-chain. However, the order in which the functions are called off-chain may differ from the order in which they get executed on individual miners. For instance, if a Blockchain client serially calls attestation, crosscheck and checking, the three functions may be delivered to an individual miner in a different order, say checking, crosscheck, attestation. One can rely on clients to synchronize different contract calls, say by requiring a client only calls crosscheck until it confirms the finality of the call of log attestation. However, this client-side synchronization is extremely slow and unnecessarily time-consuming.

Therefore, in our work, we implement the step of log attestation by a public contract function and the other two steps by a private contract function. A private contract function is called by another contract function and cannot be called from off-chain clients. By this means, the serial order is ensured directly by individual miners without expensive client-side finality confirmation.

4.4 The Protocol: Execution and Construction

In this section, we present the protocol in the top-down fashion. We first describe the overall protocol execution flow involving both on-chain and off-chain parties. We then describe the construction of the protocol with the details of the various primitives used in the construction..

4.4.1 Overall Protocol Execution

We present the overall protocol execution. The ContractChecker protocol runs in epochs. Each epoch is of a pre-defined time interval E. During an epoch, clients send read/write requests to the storage server in the cloud. For simplicity, we consider a client runs a single thread.[4] The operations are logged on both sides of clients and the server as in Figure 2). A client logs her own operations while the server logs the operations submitted by all clients. Given an operation, a client logs the start time when the request for the operation is sent and the end time when the response of the operation is received. We assume the NTP protocol in place enables all clients to log operation time with low inaccuracy. Both clients and the server store the logged operations locally.

At the end of the epoch, both clients and the server attest to their operation log. The server calls `attestServerLog(ops_server,`sk_S`)` where she declares a total-order sequence over the operations and signs the log with server key sk_S before sending it the Blockchain. A client calls `attestClientLog(ops_client,` sk_C`)` where she signs the logged operations with secret key sk_C and sends it to the Blockchain. The total-order in the server attestation is necessary for consistency checking (recall the definition in § 3.2.2).

The smart contract running on the Blockchain receives the calls of `attestServerLog(ops_server,`sk_S`)` from the server and `attestClientLog(ops_client,` sk_C`)` from all clients. After that, it runs log verification, log crosschecking and log auditing as described previously. The result, namely the assertion of log consistency, is stored on chain for future inquiry.

Clients can call `consistencyResult()` to check the consistency result. This function checks two conditions: 1) Whether all clients' transactions `attestClientLog(ops_client,` sk_C`)` are finalized on Blockchain. 2) whether the consistency assertion (i.e., the result of running `auditLog()`) is finalized on Blockchain.

The execution schedule of ContractChecker is parameterized by an epoch duration E. In practice, E is set to be multiply of the block time on Blockchain, that is, $E = n \times B$. n is determined based on the client availability and latency requirements in target scenarios.

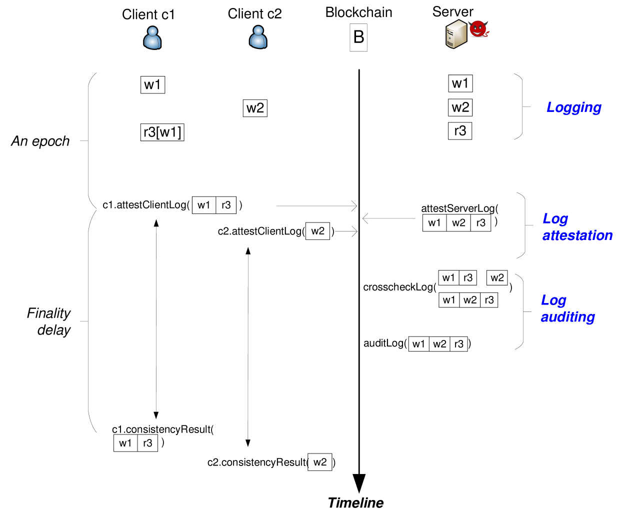


Figure 1: Running ContractChecker: An example scenario with two clients. Client C_1 sends a write w_1 before a read $r_3[w_1]$ that returns the record written by w_1. Client C_2 sends a write w_2 to the server.

An example is presented in Figure 1 where the ContractChecker protocol runs one epoch between two clients C_1 and C_2. An epoch can be set to multiple of Blockchain's block time. On Ethereum, it is multiply of 15 seconds. During the epoch, client C_1 submits two operations to the server, namely w_1 and $r_3[w_1]$. $r_3[w_1]$ represents a read operation that returns write w_1. Client C_2 submits an operation w_2 to the server. All operations are on the same data key, and they are processed by the server in the serial order of $w_1 w_2 r_3[w_1]$. By the end of epoch, it first runs log attestation: C_1 calls `attestClientLog`(w_1, r_3) and C_2 calls `attestClientLog`(w_2). The server declares a total-order w_1, w_2, r_3 by calling `attestServerLog`(w_1, w_2, r_3). The smart contract then stores the two client logs and one server log on the Blockchain. After that, it runs two functions: `crosscheckLog`$(\{w_1, r_3\}, \{w_2\}, \{w_1, w_2, r_3\})$, and `auditLog`$(\{w_1, w_2, r_3[w_1]\})$. As a result, the smart contract asserts the log is inconsistent and stores it on chain.

Meanwhile, the clients may call C`.consistencyResult()`. The function call will block until the transaction of C`.attestClientLog()` is finalized on Blockchain. On Ethereum, it takes 25 epochs to finalize a transaction. After 25 epochs, if the transaction is finalized, the consistency assertion stored on chain can be treated as an immutable statement and be further used by applications. In addition, the client can truncate her local operations in that epoch. If the transaction is not included in the Blockchain, the client will retry C`.attestClientLog()`.

[3]More precisely, given a key-value record, we can publish to the Blockchain the key hash and the value hash.

[4]In practice, the case of a multi-threaded client can be treated as multiple single-threaded virtual clients.

Now we modify the setting of this example that w_1 and w_2 are processed concurrently by the server (i.e., with overlapped time intervals). In this case, a rational server (recall § A.2.1) will search and attest to the total-order that is consistent, namely $w_2 w_1 r_3[w_1]$. Note that the other total-order $w_1 w_2 r_3[w_1]$ also matches the real-time relation, as w_1 and w_2 are concurrent. But it contains inconsistent read $r_3[w_1]$.

Note that in this example, the ContractChecker clients are stateless (in that client C_1 can discard operations $w_1 r_3$ and truncate the log after client attestation) and store only local operations (in that client C_1 does not need to store the operations of client C_2). This saves the client cost, comparing all existing approaches including Catena and Caelus which require clients to maintain global state without log truncation.

4.4.2 *Construction of On-chain Contracts*

```
contract ContractChecker{
  address payable ownerContract;
  attestClientLog(Op[] ops_client, signature_client){
    if(true == verifyClientLog(ops_client,
               signature_client, pubkey_client)){
      lock.acquire();//to prevent race conditions
      if(++attested_clients == N
         && attested_server = 1){
         if(!crosscheckLog(ops_clients, ops_server))
           throw;
         if(!auditLog(ops_server)) throw;
      }
      lock.release();
    }
  }

  attestServerLog(Op[] ops_server, signature_server){
    if(true == verifyServerLog(ops_server,
     signature_server, pubkey_server)){
      lock.acquire();//to prevent race conditions
      if(attested_clients == N
         && ++ attested_server = 1){
         if(!crosscheckLog(ops_clients, ops_server))
           throw;
         if(!auditLog(ops_server)) throw;
      }
      lock.release();
    }
  }

  modifier crosscheckLog(Op[] ops_clients, Op[]
     ops_server)) returns (false) {...}
  modifier auditLog(Op[] ops_server)) returns (false)
     {...}
  mapping ops_server;
  mapping ops_clients;
}
```

Figure 2: Smart-contract program of ContractChecker

The two attestations described in § 4.2 invoke a smart-contract on chain. The smart contract taking as input of log attestations from the server and clients will conduct four operations: verifying client attestations (`verifyClientLog()`), verifying the server attestation (`verifyServerLog()`), crosschecking the logs of clients and server (`crosscheckLog()`), and auditing the checked log to assert consistency (`auditLog()`).

In `verifyClientLog()`, the contract would verify the client attestation using the client's public key. In `verifyServerLog()`, the contract verifies the server attestation using the server's public key. It is optional that an individual operation can be signed by both the client and server. In this case, both verification functions will verify individual operations as well using both clients' and server's public keys. Once logs are verified, the client logs are union-ed and are crosschecked with the server log to find any inequality (`crosscheckLog()`). Once the logs are successfully cross-checked, it runs log auditing (i.e., `auditLog()`) where strong consistency conditions (e.g., operation ordering and read freshness) are checked over the crosschecked server log.

4.4.3 *Storage of Historical Operations*

For consistency verification, historical operations need to be persisted. Historical operations are those operations in the previous epochs. They are needed because an operation in the current epoch returns the record written by a historical operation.

A baseline is to maintain all historical operations on chain (e.g., in a smart-contract container). A slightly better approach is to store the "latest" snapshot of the records in smart contract. These baselines, however, cause high Blockchain cost as they rely on expensive on-chain storage for the actual data storage.

We build a Merkle tree to enable authenticated query processing between the untrusted server and the Blockchain. Briefly, the dataset that historical operations lead to is stored on the untrusted server. The historical dataset is digested by an Merkle tree. The root hash of the Merkle tree is kept in the smart contract on chain. In an epoch, when `auditLog()` is called, the ContractChecker contract will need to query the historical dataset off-chain (see the example below). When a query of a data key is sent, an "authentication path" is constructed by the hash of neighbor nodes along the path from the record to the root in the Merkle tree. This authentication path can be used to prove the membership (or non-membership) of the record in the historical dataset. When `auditLog()` is completed successfully, all operations in the current epoch are reflected to the off-chain Merkle tree and the on-chain hash root is updated as well.

For instance, consider the operation sequence $w_1(K), w_2(K'), r_3[w_1](K)$ where $w_1 w_2(K')$ occurs in Epoch 1 and $r_3[w_1](K)$ occur in Epoch 2. For the Blockchain to assert the consistency of $r_3[w_1]$, it will need to know as of Epoch 2 whether the latest write of data key K is w_1. The Blockchain will send out this query to the server and the server who maintains a Merkle tree of the current state (of latest records of all data keys) returns the authentication path for w_1, so that its freshness can be verified by the Blockchain.

4.4.4 *Finality of Client Attestations*

The finality checking is realized by all active clients staying online and confirming the finality of their own attestations. For a client to check the finality of her attestation, she simply checks if there are F blocks on the Blockchain that are ordered after the block where her attestation transaction is stored. Blockchain's immutability guarantees the hardness of altering or omitting the client attestation if its finality is confirmed on the chain. If the transaction is not finalized, the client is responsible for retrying `attestClientLog()` until the finalized transaction inclusion.

A consistency assertion is only valid when all active clients have confirmed the finality of their log attestations. Had one client's log not confirmed on the Blockchain, a malicious server can launch the selective-omission attack (detailed in § 5.3.1) that leads to an incorrect assertion. Clients wait until all clients' attestations are finalized on Blockchain (see § 5.3.1 for a heuristic client-synchronization scheme).

4.4.5 *Identity Management*

In ContractChecker, both clients and the server are registered in the sense that their public keys are known by the contract code. The public keys are used to verify the log attestations from clients and the server. The log verification is necessary to prevent anyone from injecting arbitrary operations in the log which can easily obstruct the consistency assertion.

To set up the public keys, we assume a secure key-distribution channel exists. Clients can dynamically join and register; we assume an external trusted identity provider in place who assists the user authentication and sends the new public keys to the smart contract.

The clients and server can reuse the public key pair in their Blockchain wallets for ContractChecker operations. In this work, we assume the secret keys are securely managed by clients and server offline. With securely managed keys, a variety of attacks, e.g., clients and server impersonation, are prevented.

5. PROTOCOL ANALYSIS

In this subsection, we analyze the protocol correctness under both benign and malicious settings. We start with the correctness with benign clients and the server. We consider the case of lowly-available clients. We then focus on the malicious cases, by analyzing protocol security with a malicious server, malicious clients, and unreliable Blockchain.

5.1 Correctness

In ContractChecker, the protocol correctness states that if the protocol is truthfully executed by benign clients and server, a consistent operation history will be asserted as a consistent history, and an inconsistent operation history will be asserted as an inconsistent history. Informally, given any consistent operation history Ops_S and any $\{Ops_C\}$ with $\cup Ops_C = Ops_S$, it holds that S.`attestServerLog`(Ops_S), C.`attestClientLog`(Ops_C), C.`consistencyResult`$() = Y$. For any inconsistency operation history Ops_S and any $\{Ops_C\}$, S.`attestServerLog`(Ops_S), C.`attestClientLog`(Ops_C), C.`consistencyResult`$() = N$.

Analyzing the protocol correctness is straightforward. If the clients and server are benign, the protocol guarantees the authentic copies of client logs and server log are send to the Blockchain as the input of `crosscheckLog()` and `auditLog()`. The computation logic guarantees that a server log with consistency total-order will be asserted as consistency. For an inconsistency log, the server cannot find a total-order without inconsistency and the case of inconsistency will be detected by `auditLog()`.

Correctness with low client availability: In ContractChecker, we consider active clients and inactive clients: Given an epoch, an active client is one that has submitted at least one operation to the cloud. Otherwise, it is an inactive client. ContractChecker requires only active clients in an epoch to be available to participate in the protocol by the end of the epoch. Be more precise, given an inactive client, it does not require its availability for protocol participation.

The protocol correctness without the availability of inactive clients is straightforward. Whether inactive clients send their empty log to the Blockchain, it does not affect either the union of all client operations or the log crosscheck (`crosscheckLog()`). Therefore, an attack is detected by the server-client log inequality, which is irrelevant to the inactive clients' empty logs.

This is in contrast with existing client-based protocols which require the availability of both active and inactive clients (as analyzed in Appendix ??). Briefly, the reason that ContractChecker does not require availability of inactive clients while other protocols do is that it cannot distinguish the benign case of an inactive client who is legitimately unavailable from the malicious case of an active client who detects an attack but whose attack report is suppressed by the untrusted server who relays the report among clients. In ContractChecker, it does not rely on the untrusted server to report the case of an attack, but instead by the trusted Blockchain.

5.2 Security under Server/Client Attacks

In this subsection, we consider the attacks launched by individual malicious clients or server. As reasoned before, our threat model excludes the collusion between a client and server. We leave the Blockchain exploits to the next subsection.

In our threat model, either a client or the server can forge her attestation to the Blockchain. Specifically, she can forge a non-existing operation (A1), omit a valid operation (A2), replays a valid operation multiple times (A3), reorders the serially-executed operations (A4). Note that we assume a rational server who will not declare a total-order with inconsistency among concurrent and consistent operations.

A malicious server can exploit the operation forging (A1-A4) to conceal an inconsistent log and to avoid paying the penalty to victim clients. For instance, successfully omitting w_2 in $w_1 w_2 r_3[w_1]$ may fool the ContractChecker to falsely assert the operation history to be consistent. A malicious client may exploit the operation forging (A1-A4) to forge an inconsistent log and to falsely accuse a benign cloud. For instance, a client can forge an operation (A1) $w_{2.5}$ in a consistent log $w_1 w_2 r_3[w_2]$ to make it look like inconsistent.

ContractChecker can detect any operation forging (A1, A2) by a mismatch between server attestation and client attestation (in `crosscheckLog()`). Without server-client collusion, if one party, say the server, forge an operation in the server attestation, the forged/omitted operation will be found in the server/client attestation, but not in the client/server attestation. The replay attack (A3) can be detected by multiple identical operations in the server/client log. Reordered operations (A4) can be detected by the condition that the operation order does not match the real-time order (i.e., an operation that occurs later is ordered before an earlier operation).

In addition, ContractChecker can be security-hardened to not only detect attacks but also mitigate these attacks. Briefly, assuming an online operation is both signed by the server and the client, the ContractChecker can distinguish different attack vectors (i.e., A1, A2, A3 or A4), which enables it to recover the forged attestation logs and to assert the consistency correctly despite the attack. The details of the attack mitigation are presented in Appendix A.2.3 and A.2.

5.3 Security under Blockchain Exploits

5.3.1 *Exploiting Blockchain Write Unavailability*

Recall that the practical Blockchain systems exhibit low write availability, and may drop valid transactions. Given a faulty Blockchain like this, a malicious server can selectively omit operations in her attestation such that dropped transactions of client attestations correspond to the omitted operations in the server attestation. By this mean, the server can omit operations without being detected by ContractChecker, in a way to conceal inconsistency. For instance, in Figure 1, Client C_2's call of `attestClientLog`(w_2) can be dropped by the Blockchain. The malicious server, observing w_2 is not included in the Blockchain, can selectively omit the operation in her attestation. This will allow the forged log (with omitted operations) to pass the log crosschecking (more specifically, `crosscheckLog`$(\{w_1, r_3[w_1]\}, \{w_1, r_3[w_1]\})$, which further tricks the ContractChecker to assert incorrectly that the operation history (which is actually $w_1, w_2, r_3[w_1]$) is consistent. Because in this attack, the server selective the operations to omit based on dropped transactions in Blockchain, we call this attack by selective-omission attack.

Security hardening: The selective-omission attack can be prevented if the finality of any client's log attestation in Blockchain can be assured of. To prevent the miss of a transaction on Blockchain, a common paradigm is to resubmit the transactions. A naive resubmission policy is to resubmit every F block until the transaction is finalized. With this naive policy, it may lead to an unwanted situation where the resubmitted transaction keeps being declined (e.g., due to low transaction fee).

Enforcing a time bound on transaction finality is crucial to the security of ContractChecker. That is, ContractChecker's security relies on whether a transaction (after resubmission) can be guarantee to succeed before a pre-scribed timeout (e.g., $2F$ blocks). Because if both clients and Blockchain are allowed to be unavailable in an arbitrarily long time, it is impossible to distinguish between the benign case of an inactive client where the client does not send a transaction and the malicious case of an unavailable Blockchain under selective-omission attacks.

To break the indistinguishability and to prevent the attack, we propose a transaction-resubmission policy that provide high confidence of transaction finality within a time bound. In our policy, any active client will monitor the state of her `attestCliengLog()` call. If the finality is not confirmed after F blocks, the client will increase the chance of inclusion by maximizing the transaction fee. We determine the maximal transaction fee based on heuristics in practice. High chances are that the second submission with maximal fee will make the transaction successfully included in the Blockchain.

With the resubmission policy, it can be expected all clients' transactions in an epoch are successfully included in the Blockchain. We require that all clients wait after $2F$ blocks before using the log-consistency assertion produced by the ContractChecker program. Through this mechanism, it synchronizes across all active clients on the finality of their log attestations on Blockchain.

Analysis of attack prevention: If all active ContractChecker clients execute the transaction resubmission policy, the selective-omission attack can be prevented. What follows is the security analysis: Assume the resubmission policy can guarantee the high chance (as will be evaluated in § **??**) that all clients' transactions (for `attestClientLog()`) are included in the Blockchain before the timeout of $2F$ blocks. The selective-omission attack cannot succeed because any valid operation can be found in the client log attestations and the omission of the operation in the attack can be detected by the mismatch in log crosschecking. In the previous example, by the time w_2 is included in the Blockchain (before $2F$ blocks), the ContractChecker can detect the w_2 is absent in the server log as it appears in Client c_2's log. The ContractChecker can identify the attack, recover the server log for making a correct assertion about the operation inconsistency.

5.3.2 Forking Attacks exploiting Contract Races

Forking attacks: Consider multiple clients share a state hosted on a server. In this setup, a forking attacker is a malicious server who forks the shared state and serves different clients with different (forked) states. In client-based consistency protocols (e.g., Catena and Caelus), the malicious server may fork her log attestations for different auditing clients, such that the forked global logs appear to be consistent from different clients' local views.

In ContractChecker, the malicious server can mount a forking attack by equivocating with two conflicting attestations to the Blockchain. The hope is to make it crosscheck different clients' attestations with different (forked) server attestations. However, it is more challenging to make a successful forking attack in ContractChecker than client-based auditing protocols. Because the log auditing (for consistency checking) in ContractChecker does not occur on individual clients, but instead in Blockchain nodes where all clients' logs are fused. In this new setting, the success of a forking attack depends on whether one can separate different client logs and make them compared with the forked server logs.

Exploiting contract races: To launch a forking attack in ContractChecker, the malicious server may run contract functions concurrently to trigger race conditions for exploit. To be more concrete, consider the function-call sequence S.`serverAttestLog`$(w_1, w_2, r_3[w_1])$, C_1.`clientAttestLog`(w_2), C_2.`clientAttestLog`$(w_1, r_3[w_1])$, S.`serverAttestLog`$(w_2, w_1, r_3[w_1])$. Suppose before S.`serverAttestLog`$(w_2, w_1, r_3[w_1])$ runs, the contract already enters `crosscheckLog()` where the server log $w_1, w_2, r_3[w_1]$ is being crosschecked with client logs w_2 and $w_1, r_3[w_1]$. When S.`serverAttestLog`$(w_2, w_1, r_3[w_1])$ runs, it might happen that the call of S.`serverAttestLog`$(w_2, w_1, r_3[w_1])$ replaces the variable of `ops_server` during `crosscheckLog()`. This may lead to the unwanted situation that client log w_2 is crosschecked with one version of server log $w_1, w_2, r_3[w_1]$, and client log $w_1, r_3[w_1]$ is crosschecked with another version of server log $w_2, w_1, r_3[w_1]$. In this situation, server log $w_2, w_1, r_3[w_1]$ will survive as the successfully crosschecked version, which will lead to an incorrect consistency. We call this attack by forking-by-races attack.

The ContractChecker prevents the forking-by-races attack by synchronizing the critical functions and avoiding concurrency. Concretely, in ContractChecker, we define a critical section around the functions `crosscheckLog()` and `auditLog()`, such that the execution of these two victim functions needs to be serialized with other functions. To be more specific, in the forking-by-races attack, the attacker (i.e., calling function S.`serverAttestLog`$(w_2, w_1, r_3[w_1])$) and the victim (i.e., running function `crosscheckLog()`) are forced to execute in a serial order. Without concurrent execution of attacking and victim functions, the attack cannot succeed.

5.3.3 Forking Attacks exploiting Blockchain Forks

In this version of forking attack, the malicious server can exploit the Blockchain forks. The server sends forked attestations to different Blockchain forks. In the case of transient forks, the server may be able to encode attestation forks in double-spending transactions as is done in Catena [37]. In the case of permanent forks, the server can simply encode attestation forks in two transactions and submit them separately to the two Blockchain forks. The hope is that if there is some client attestation which is omitted in one of the Blockchain forks, chances are the attack can succeed. For instance, to Blockchain fork 1, the server can send the attestation $w_2, w_1, r_3[w_1]$, client C_1 sends her attestation w_1, r_3, and client C_2 has her attestation be omitted.

The forking attack by exploiting Blockchain forks cannot succeed in ContractChecker, if it guarantees any client will retry her attestation to all Blockchain forks. In the previous example, client C_2's attestation cannot be omitted if she retries the attestation to Blockchain fork 1.

6. EVALUATION OF CLIENT COST

In this section, we evaluate the client cost in ContractChecker. We first present an analysis of client cost and then present our experimental results.

6.1 Cost Analysis

Cost model: For a consistency verification protocol, either client-based schemes or ContractChecker, its execution can be modeled by the following: Clients periodically send their log attestations and check the consistency results. In this process, a client's cost is characterized by 1) how many operations the client needs to store, and 2) how long the client needs to maintain an operation. In our cost model, we accredit to one cost unit storing one operation by a client in one epoch. Given a process of T epochs, a client's total cost is the sum of cost units of all operations stored in the client in all epochs. If an operation is stored continuously in a client for T epochs, it is counted as T units.

Based on the above model, we present a cost analysis of client-based auditing schemes and ContractChecker.

Client cost in client-based auditing: In client-based auditing schemes, a client needs to access operations submitted by all (active) clients in the current epoch, including herself and other clients; these operations may only need to be stored by the clients in one epoch, assuming a trusted third-party attester. For simplicity, we omit the cost of accessing historical operations. Instead, we focus on a low-bound estimate of the client cost in client-based auditing schemes:

$$CCost_{ClientAudit} \geq T \cdot N \cdot M \tag{1}$$

The above equation considers a client's cost in a process of running client-based auditing scheme in T epochs, with totally N clients where on average each client in one epoch submit M operations.

Client cost in ContractChecker: In the above setting, a ContractChecker client only needs to store her own operations, that is, M operation per epoch. But the ContractChecker client needs to keep an operation for, instead of the current epoch, but an extended period of time denoted by F_t/E epochs. F_t is the total delay for successfully submitting a client attestation to the Blockchain in ContractChecker. Briefly, $F_t = r \cdot (B \cdot F + P)$, where r is the average times to resubmit a client attestation, and $F/B/P$ is the finality delay/block time/average wait time for transaction validation. Thus, ContractChecker's client cost is as follows. Note that in ContractChecker, log auditing is fully delegated and clients are relieved from accessing historical operations.

$$CCost_{ContractChecker} = T \cdot M \cdot F_t/E \tag{2}$$

Therefore, the cost saving of ContractChecker comparing existing client-based auditing schemes is:

$$f = \frac{CCost_{ClientAudit}}{CCost_{ContractChecker}} \geq \frac{E \cdot N}{F_t} \tag{3}$$

This formula shows that the cost saving of ContractChecker comparing existing work depends on three factors: the total attestation delay F_t, the maximal number of clients N, and epoch time E.

6.2 Experiments

Based on the cost model above, we design experiments for client cost measurement. Our experiments focus on transaction-validation delay (P) and maximal number of operations ($N \cdot M$), w.r.t. the budgeted amount of Ethers. Based on the results, we compare ContractChecker's client cost with that of client-based auditing schemes.

6.2.1 Experiment Setup

We set up a private Ethereum network in a LAN environment. The Ethereum runs on machines of the following specs: Intel 8-core i7-6820HK CPU of 2.70GHz, 32KB L1 and 8MB LL cache, 32 GB RAM and 1 TB Disk. When setting up the network, we use the default configuration (e.g., difficulty levels) for Ethereum. Note that we only measure the cost (e.g., transaction fee and Gas), which is independent of network scale. We deploy our smart-contract program written in Solidity to the Blockchain, by leveraging an online python tool[5].

6.2.2 Transaction Finality Time

In the experiment, we measure the relationship between transaction fee and the time to finalize a storage operation on Blockchain. We use the Geth client that interacts the Ropsten testnet of Ethereum [14] using asynchronous RPC interface, promise [9]. In the implementation, we bound the maximum depth of the asynchronous operation chain in promise to avoid errors.

We measure the finality delay by recording the timing difference between sending a transaction and finalizing the transaction. In experiments, we generate transactions of the same fee 18 times and report the average and standard deviation of the measured delay. Transactions of different fee are dispatched at a different time and end up at different blocks.

The results are presented in Figure 3c show how finality delay grows along with the transaction fee. As the transaction fee grows, the transaction finality delay (including the validation delay) is decreased from 35 blocks to about 15 blocks.

6.2.3 Clients Cost Comparison

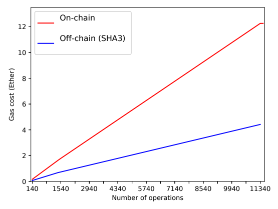
We plot the client costs based on the equations 1 and 2, and the transaction finality time as measured before. The results are plotted in Figure 3d. When the number of clients is small, ContractChecker incurs higher costs due to its need of storing operations across multiple epochs. When the number of clients grows, ContractChecker scales much better than client-based auditing schemes. Because a ContractChecker client only needs to store its own operations. With increasing amount of fees, ContractChecker client cost decreases thanks to the decreased transaction delay.

6.2.4 Storage Capacity

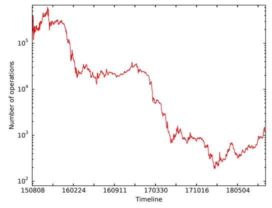
This experiment measures the storage capacity with transactions fee. Given a budget of Ether, ContractChecker's storage capacity is bounded which affects the number of operations the Blockchain can consume and the maximal number of clients it can support.

In this experiment, we use the YCSB benchmark suit. To adapt YCSB to ContractChecker, we first play and record the YCSB workload trace. We then replay the trace during the experiments. In our experiments, we set an epoch to be one block time (B) and limit 140 operations per epoch (so that Blockchain is not saturated). We set finality delay to be $F \cdot B = 6B$. We drive YCSB workload D (of 95% reads) into ContractChecker. The workload consists of 11340 operations in total which are evenly distributed among 81 epochs (with each containing 140 operations). We measure the Gas cost (from the transaction receipt) with the varying number of operations. The result is in Figure 3. The Gas cost grows linearly

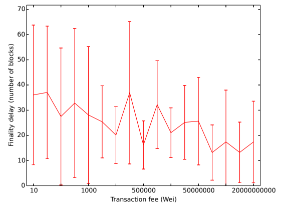
[5] `https://github.com/ConsenSys/ethjsonrpc`



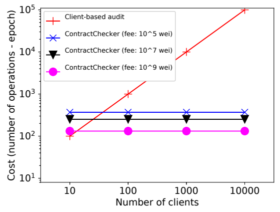
(a) Storage capacity in Ether



(b) Storage capacity with $100



(c) Transaction finality delay



(d) Client costs

Figure 3: ContractChecker's client costs

with operation count. For the off-chain placement of historical operations (of both server and client logs), the total Gas is about 4.5 Ether for 11340 operations.

The off-chain placement of historical operations saves 80% of the cost comparing the on-chain placement. This result shows that transaction fee is cheaper than on-chain storage in Ethereum and the off-chain placement that trades the latter for the former is cost effective.

We also convert the Ether to US dollar and show the dollar cost with ContractChecker storage capacity. Because the price of Ether in US dollar changes over time, we present the ContractChecker storage capacity allowed by 100 dollars at a different time. The result is illustrated in Figure 3b. It shows that with 100 dollars, the maximal capability it can buy occurs on Oct. 2015 when the Ether is cheapest. However, as the cryptocurrency price continuously drops in 2018, the ContractChecker will be more cost-effective in US dollar.

6.3 Related Work

Blockchain applications: Blockchain has been applied in both financial and non-financial applications. In the application scenarios, Blockchain and its security properties are repurposed to address application-specific security needs. CSC [19] runs criminal smart contract on Blockchain for facilitating commissioned crimes. Paralysis [39] supports practical wallet management under key loss/compromise by using Blockchain as trusted clock to detect client unavailability. Blockstack [16] runs trusted directory on Blockchain by treating it as immutable data storage. IKP [30] proposes an incentive scheme based on Blockchain to address the certificate mis-issuance problems. Blockchain is also used as the source of randomness [33], non-equivocation log as in Catena [37], etc. in many novel applications.

Blockchain throughput improvement: There are known performance problems in Blockchain notably the limited throughput. There are protocol-analysis works on Blockchain that try to understand the throughput limitation (e.g., by block sizing [22]). Lightning network [10, 32] is an off-chain scheme that batches multiple micro-payment transactions to reduce the load on chain. Lightweight mining protocols are proposed including proof-of-stake [13, 12], proof-of-elapsed-time [11, 40], etc. Ekiden [19] combines Blockchain with trusted hardware to form a virtual high-throughput Blockchain system. Blockchain sharding [27, 29] partitions the state of transaction graph among multiple miners, in a way to improve its scalability.

7. CONCLUSION

This work presents ContractChecker, a lightweight consistency verification protocol based on public Blockchain. It presents a new approach by auditing the storage log on the Blockchain, and has advantages in saving the clients' cost and lower availability requriements. ContractChecker is implemented in a middleware system federating cloud services, clients and the Blockchain.

8. REFERENCES

[1] Amazon s3: https://aws.amazon.com/s3.
[2] Certificate transparency, https://tools.ietf.org/html/rfc6962.
[3] Docker hub: https://hub.docker.com/.
[4] Dropbox: https://www.dropbox.com/.
[5] Ethereum project: https://www.ethereum.org/.
[6] Ethereum smart contract best practices: Known attacks: https://consensys.github.io/smart-contract-best-practices/known_attacks/.
[7] Github: https://github.com/.
[8] How to fix icloud sync in seconds, https://www.computerworld.com/article/2916476/apple-ios/how-to-fix-icloud-sync-in-seconds.html.
[9] Javascript promises: an introduction: https://developers.google.com/web/fundamentals/primers/promises.
[10] Ligntning network, scalable, instant bitcoin/blockchain transactions: https://lightning.network.
[11] Poet 1.0 specification, https://goo.gl/vd1mco.
[12] Proof of stake (bitcoin), https://en.bitcoin.it/wiki/proof_of_stake.
[13] Proof of stake faqs (ethereum), https://github.com/ethereum/wiki/wiki/proof-of-stake-faqs.
[14] Testnet ropsten (eth) blockchain explorer: https://ropsten.etherscan.io/.
[15] 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017. IEEE Computer Society, 2017.
[16] M. Ali, J. C. Nelson, R. Shea, and M. J. Freedman. Blockstack: A global naming and storage system secured by blockchains. In A. Gulati and H. Weatherspoon, editors, 2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016., pages 181–194. USENIX Association, 2016.
[17] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig, D. Hildebrand, and A. R. Butt. Improving docker registry design based on production workload analysis. In 16th USENIX Conference on File and Storage Technologies (FAST 18), pages 265–278, Oakland, CA, 2018. USENIX Association.
[18] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.
[19] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. M. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart

contract execution. CoRR, abs/1804.05141, 2018.

[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In SoCC, pages 143–154, 2010.

[21] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. ACM Trans. Comput. Syst., 31(3):8, 2013.

[22] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. E. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, D. Song, and R. Wattenhofer. On scaling decentralized blockchains - (A position paper). In J. Clark, S. Meiklejohn, P. Y. A. Ryan, D. S. Wallach, M. Brenner, and K. Rohloff, editors, Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers, volume 9604 of Lecture Notes in Computer Science, pages 106–125. Springer, 2016.

[23] I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers, pages 436–454, 2014.

[24] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan. Bolt: Data management for connected homes. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), pages 243–256, Seattle, WA, 2014. USENIX Association.

[25] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst., 12(3):463–492, 1990.

[26] B. H. Kim and D. Lie. Caelus: Verifying the consistency of cloud services with battery-powered devices. In 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015, pages 880–896. IEEE Computer Society, 2015.

[27] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In 2018 2018 IEEE Symposium on Security and Privacy (SP), volume 00, pages 19–34.

[28] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In OSDI, pages 121–136, 2004.

[29] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. A secure sharding protocol for open blockchains. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016, pages 17–30. ACM, 2016.

[30] S. Matsumoto and R. M. Reischuk. IKP: turning a PKI around with decentralized automated incentives. In 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017 [15], pages 410–426.

[31] D. Mazières and D. Shasha. Building secure file systems out of byantine storage. In Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002, pages 108–117, 2002.

[32] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry. Sprites: Payment channels that go faster than lightning. CoRR, abs/1702.05812, 2017.

[33] A. Narayanan, J. Bonneau, E. W. Felten, A. Miller, and S. Goldfeder. Bitcoin and Cryptocurrency Technologies - A Comprehensive Introduction. Princeton University Press, 2016.

[34] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage slas with cloudproof. In J. Nieh and C. A. Waldspurger, editors, 2011 USENIX Annual Technical Conference, Portland, OR, USA, June 15-17, 2011. USENIX Association, 2011.

[35] D. Terry. Replicated data consistency explained through baseball. Commun. ACM, 56(12):82–89, 2013.

[36] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In M. Kaminsky and M. Dahlin, editors, ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013, pages 309–324. ACM, 2013.

[37] A. Tomescu and S. Devadas. Catena: Efficient non-equivocation via bitcoin. In 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017 [15], pages 393–409.

[38] S. Torres-Arias, A. K. Ammula, R. Curtmola, and J. Cappos. On omitting commits and committing omissions: Preventing git metadata tampering that (re)introduces software vulnerabilities. In T. Holz and S. Savage, editors, 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016., pages 379–395. USENIX Association, 2016.

[39] F. Zhang, P. Daian, I. Bentov, and A. Juels. Paralysis proofs: Safe access-structure updates for cryptocurrencies and more. IACR Cryptology ePrint Archive, 2018:96, 2018.

[40] F. Zhang, I. Eyal, R. Escriva, A. Juels, and R. van Renesse. REM: resource-efficient mining for blockchains. In E. Kirda and T. Ristenpart, editors, 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017., pages 1427–1444. USENIX Association, 2017.

[41] G. Zyskind, O. Nathan, and A. Pentland. Decentralizing privacy: Using blockchain to protect personal data. In 2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015, pages 180–184. IEEE Computer Society, 2015.

APPENDIX

A. APPENDICES: SUPPLEMENTARY DETAILS

A.1 Motivating Applications

While ContractChecker can definitely benefit small-scale cloud-hosted applications such as smart homes [24], we argue that ContractChecker can be utilized in a long spectrum of application scenarios, ranging from low-throughput small organizations [24], to enterprise-scale cloud object stores [17]. In the following we list two representative examples of real-world use cases.

DockerHub style container-distribution infrastructures (hereafter called ContainerRegistry): The "ContainerRegistry" distributes the latest software from developers to users in the form of Docker container images. The ContainerRegistry is usually hosted in third-party public cloud, such as in hub.docker.com, which is independent of both developers and users. The storage consistency implies security as returning a stale image (violating the strong consistency) from the hub means the users may run buggy programs, vulnerable to the latest attacks. The ContainerRegistry is usually accessed infrequently. For instance, workload analysis on IBM Docker Registry reports that, among seven geo-distributed registry deployments, the busiest one serves only 100 requests per minute for more than 80% of time [17][6].

Certificate-transparency(CT) log: The CT log stores the certificates about key-identity bindings, and is made public to invite public scrutiny for timely detection of certificate mis-issuance. Violating storage consistency in CT log leads to the security breaches such as concealing of mis-issued certificate or use of revoked keys. The load of writes in a CT log (regenerating/revoking a public key) is usually low and the read load is not high on a small number of domains (websites).

Additional relevant scenarios such as device synchronization through cloud and Github-hosted software development are found in Appendix § **??**. These application scenarios feature the following properties that motivate the use of Blockchain: S1) The clients are of limited capability in computing, storage, and availability. Thus they outsource data storage to a third-party cloud. In both scenarios above, the client can be a mobile phone (e.g., hosting a web browser against a CT log or installing security patches against a ContainerRegistry). S2) On the cloud-hosted data storage, violating storage consistency leads to a security consequence to the application. Intuitively, using a Blockchain as a "trusted" third-party witness can harden the application security. S3) The storage-access load in our application scenarios is typically lower than tens of operations per second. In particular, the low throughput properties of these typical application scenarios make it amenable for the use of Blockchain, which is known to have limited throughput in ingesting transactions.

A.2 Security under Client/Server Attacks

A.2.1 Security against the Malicious Server

In ContractChecker, the server's job is to declare a total-order over concurrent operations and to attest to it. A benign server is allowed to find a total-order without inconsistent operations to the degree that she does not forge operations.

Before describing the server threats, we stress that our server, be it benign or malicious, is rational. Given a consistent operation history, a rational server does not falsely attest to a total order with inconsistency, as it does not have the incentive (recall § **??**). For instance, consider an operation history $w_1|w_2r_3[w_1]$ where w_1 and w_2 occur concurrently. A rational server will find the total-order for consistency (i.e., $w_2w_1r_3[w_1]$) and attest to it. The rational server is not incentivized to attest to the total-order with inconsistent operations (i.e., $w_1w_2r_3[w_1]$). Note that both total-orders match the real-time relation in the concurrent operation history.

Server threats: The goal of a malicious (and rational) server is this: Given an inconsistent operation history, a malicious server aims at concealing the inconsistency by forging operations and declaring a false total order over it. In order to conceal inconsistency, the malicious server can launch a variety of threats: A malicious server forges a non-existing operation (AS1), omits the valid operation (AS2), replays a valid operation multiple times (AS3), reorders the serially-executed operations (AS4).

Attack detection: ContractChecker can detect the server attacks by the difference between the client logs and server log. Concretely, the server forging an operation (AS1) can be detected by an operation in the server log that cannot be found in any client log. The server omitting an operation (AS2) can be detected by an operation in a client log that cannot be found in the server log. The replay attack (AS3) can be detected by identical operations in the server log. Reordered operations (AS4) can be detected by the condition that the operation order does not match real-time (i.e., an operation that occurs later is ordered before an earlier operation).

A.2.2 Security against Malicious Clients

Client threats: A malicious client is incentivized to falsely accuse a benign server of inconsistent operations that the server did not process. Towards this goal, a malicious client can mount the following attacks to forge her log attestation. Specifically, she omits her local operation (AC1), forges an operation the server did not process (AC2), replays a valid operation multiple times (AC3), reorder the serially-executed operations (AC4). For instance, given a consistent history, the client can forge a write operation w_2 between w_1 and $r_3[w_1]$ to make r_3 look like a stale read. She can omit w_1 to make $r_3[w_1]$ look like an invalid read. She can reorder her local operations from $w_2w_1r_3[w_1]$ to $w_1w_2r_3[w_1]$, such that $r_3[w_1]$ is falsely inconsistent.

Attack detection: The attacks by forging a client log can be detected by the difference between the client logs and server log. Due to no server-client collusion, we assume that the server attests to a truthful log of operations. Thus, any operation forged by the client will result in a mismatch to the server log (which can be similarly analyzed as malicious server attacks).

A.2.3 Security Hardening for Attack Mitigation

In the ContractChecker, attacks from a malicious server or clients are detected by the mismatch between the server attestation and client attestations (in `crosscheckLog()`). In the case of attacks, the ContractChecker does not only detect them but also mitigate them in the sense of making a trustworthy assertion about the log consistency in the presence of attacks. We describe how attacks are detected and mitigated.

In order to distinguish attacks from clients and the server, we augment the basic protocol (in § 4.4) by requiring both client and server signatures on both server attestation and client attestation. More concretely, every online operation (i.e., read/write) is augmented with a server signature and a client signature. This can be naturally integrated to the regular request/response workflow without adding extra communication round. When a client submits a request, she embeds in the request her signature of the requested operation, such that the server receives a client-signed operation. When the server processes the operation, she produces a server sig-

[6]The workloads exhibit diurnal patterns and the peak throughput is remarkably low as well.

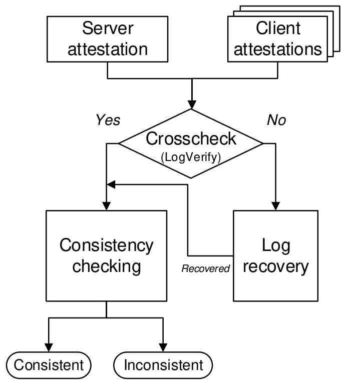


Figure 4: The logic in `LogAudit()` in ContractChecker for attack detection and mitigation

nature over the operation response, such that the client receives a server-signed operation response. When the client starts to do attestation, she will sign her local log of server-signed operations with the client signature. When the server starts to do attestation, she will declare the total order based on the client-signed operations. When the Blockchain receives a client attestation, it will validate the input by verifying both the client signatures and server signatures attached to the attestation. So is the case of validating a server attestation. We call this extension by double-signed attestation.

Distinguishing causes of attacks: With the double-signed attestation, the ContractChecker can distinguish different attacks (in our threat model). When the server attestation contains an operation whose client signature cannot be verified, this is attributed to a server attack forging a non-existing operation (AS1). If the client signature can be verified but the operation cannot be found in a client attestation, this is attributed to a client attack who omits her local operation (AC1). When the client attestation contains an operation whose server signature cannot be verified, it is attributed to a client attack forging an operation the server did not process (AC2). If the server signature can be verified but the operation cannot be found in the server attestation, it is attributed to a server attack omitting the operation (AS2). The replay attack (AC3 and AS3) where a client (or server) may duplicate a signed operation multiple times can be detected by finding duplicated operations on the server (or client) attestation. Normally, each operation is uniquely identified by a client-generated nonce (using an external source of trusted randomness). In addition, a malicious server (or client) may want to reorder the serially-executed operations in a dfferent order (AC4 and AS4). This attack can be detected by the mismatch between the timestamps in the operations and the total in the attestation.

Note that we assume a rational server who will find a consistent total-order, if it exists, from the operation history and attests to it. An irrational server who does not find such a total order for consistent operation history may falsely report a case of inconsistency in ContractChecker.

Repair server attestation: In different attack scenarios, the ContractChecker can repair the server attestation to recover the truthful log and to further assert the log consistency. In AS1 (the server forging attack), the ContractChecker removes the forged operation from the server attestation. In AS2 (the server omission attack), the ContractChecker copy the omitted operation from the client attestation to the server attestation. In AC1 and AC2 (the client attacks by operation omission and forging), no action is needed to recover the server attestation. In replay attacks (AC3 and AS3), if it finds replayed operations on the server attestation, the ContractChecker removes the duplicated copy from the server attestation. After the repair, the ContractChecker can move forward and conduct consistency checking over the repaired server attestation. In reorder attacks (AC4 and AS4), the serial operations are reordered back to match the execution order in their timestamps.

The overall workflow of our ContractChecker for attack detection and mitigation is illustrated in Figure 4.