Troxy: Transparent Access to Byzantine Fault-Tolerant Systems

Bijun Li[1], Nico Weichbrodt[1], Johannes Behl[1], Pierre-Louis Aublin[2], Tobias Distler[3], and Rüdiger Kapitza[1]
[1]TU Braunschweig [2]Imperial College London [3]Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

***Abstract*—Various protocols and architectures have been proposed to make Byzantine fault tolerance (BFT) increasingly practical. However, the deployment of such systems requires dedicated client-side functionality. This is necessary as clients have to connect to multiple replicas and perform majority voting over the received replies to outvote faulty responses. Deploying custom client-side code is cumbersome, and often not an option, especially in open heterogeneous systems and for well-established protocols (e.g., HTTP and IMAP) where diverse client-side implementations co-exist.**

We propose Troxy, a system which relocates the BFT-specific client-side functionality to the server side, thereby making BFT *transparent* to legacy clients. To achieve this, Troxy relies on a trusted subsystem built upon hardware protection enabled by Intel SGX. Additionally, Troxy reduces the replication cost of BFT for read-heavy workloads by offering an actively maintained cache that supports trustworthy read operations while preserving the consistency guarantees offered by the underlying BFT protocol. A prototype of Troxy has been built and evaluated, and results indicate that using Troxy (1) leads to at most 43% performance loss with small ordered messages in a local network environment, while (2) improves throughput by 130% with read-heavy workloads in a simulated wide-area network.

I. Introduction

If high availability and resilience to arbitrary faults for networked services is required, Byzantine fault-tolerant state machine replication offers a solution. While initially Byzantine fault tolerance (BFT) was considered impractical, the seminal work of Castro and Liskov [1] enabled a stream of research that improved the performance, lowered the complexity, and reduced the resource usage of BFT [2], [3], [4], [5], [6]. Today, BFT can be considered as ready for custom deployments and, for example, is currently evaluated in the scope of permissioned blockchain infrastructures [7]. However, when it comes to user-facing offerings in open and heterogeneous environments – such as the Internet – BFT faces a major, so far largely overlooked hurdle: the client side. Here, standardized protocols such as HTTP and IMAP are dominant and users typically utilize diverse implementations. Thus, offering for example a BFT-enabled web server is infeasible as Byzantine fault tolerance is based on the assumption that a client contacts multiple replicas and performs a majority voting over the received replies to prevent the processing of faulty replies. Of course, by means of extending the HTTP protocol and adding custom software to browsers [8] one could consider the use of BFT, but this would address only one of many standardized protocols. Instead of this more or less unrealistic endeavor, we propose to deploy BFT in a *client-transparent* fashion for different kinds of protocols.

In this paper we present Troxy, a system which achieves client-transparent BFT by relocating traditional client-side BFT functionality such as connection handling, request distribution, and majority voting to the server side, co-located to the replicas. This is enabled by relying on a trusted subsystem that can only fail by crashing and implements basic message handling, majority voting, and transport encryption: the Troxy. At implementation level, Troxy utilizes trusted execution support as offered by Intel's Software Guard Extensions (SGX) [9], [10]. At its core, SGX provides a set of new instructions that allows user-level code to allocate private and secure regions of memory called *enclaves*. By executing the application's code within enclaves, SGX provides CPU-enhanced application security and protects the enclaves from being manipulated by malicious privileged code or even hardware attacks such as memory probes. Hence, the functionality of Troxy is guaranteed to be trustworthy even in the presence of Byzantine faults in the surrounding replicas.

Based on trusted execution, Troxy offers a trusted proxy to clients that can be accessed via the original legacy protocol. Once a Troxy instance receives a client request, it forwards the request to the BFT framework, which in turn orders the request, executes it, and forwards the computed replies to the requesting Troxy. As soon as the responsible Troxy instance has received enough replies, it performs a voting over the replies and returns the correct result to the client.

As a malicious replica may intercept the communication of its Troxy, we ensure that the replica cannot alter messages without being detected: Communication between clients and Troxy instances is protected via secure, encrypted connections, which are the norm for more and more Internet-based services [11]. In addition, messages exchanged between Troxies and replicas are authenticated using common message certificates, as they are prevalent for BFT. Although immune to arbitrary or malicious behaviors, it is still possible that a Troxy instance crashes or is disconnected from its clients, and as a consequence becomes unavailable. This case is equivalent to a failing service replica in commodity infrastructures and can be handled by DNS round-robin or load-balancing appliances that enable a fail-over to another Troxy instance.

Acknowledgments: The authors thank the anonymous reviewers for their valuable feedback. This research was supported by the German Research Council (DFG) under grant no. KA 3171/1-2 and DI 2097/1-2 ("REFIT"). Funding was received from the EU's Horizon 2020 research and innovation programme under grant agreements 645011 (SERECA) and 690111 (SecureCloud).

This is the authors' version of an article published in the *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN'18)*, Luxembourg City, 25–28 June 2018.

Troxy is specifically designed for user-facing Internet-based services and therefore offers tailored support for read-heavy workloads and distant clients. In particular, this is achieved by enabling caching for Troxy instances. To not reduce the consistency guarantees of state machine replication, Troxy ensures linearizability [12] by offering a managed cache. In the context of ordering write requests, a quorum of Troxy caches is consulted and the affected data is invalidated. This way, cached read requests can be directly answered by consulting a quorum of $f + 1$ Troxy instances. Otherwise, requests are ordered via the regular BFT protocol.

We implemented Troxy on top of Hybster [13], a hybrid BFT system that already features a trusted subsystem to reduce the number of replicas to $2f + 1$. However, Troxy builds an independent extension that can be applied to other hybrid systems featuring a trusted subsystem [14], [15] as well as traditional BFT agreement protocols.

This paper makes the following contributions:

- It introduces the concept of making BFT systems transparent to clients by utilizing trusted hardware to implement a substitute of the client-side BFT library on the server side (Section II).
- It presents Troxy, which uses Intel SGX to provide transparent access to a BFT system while ensuring its integrity and security (Section III).
- It introduces a fast managed cache for read-heavy workloads that transparently switches to traditional request ordering in case of write contention (Section IV).
- It implements a prototype of Troxy that is fully transparent to clients, secure, and provides the read-cache optimization without sacrificing linearizability (Section V).

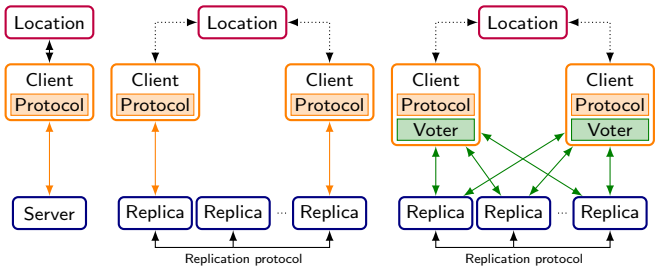
In addition, Section VI presents detailed evaluation results for Troxy gained from experiments with both microbenchmarks as well as a web server. Finally, Section VII summarizes related work and Section VIII concludes the paper.

II. Background and Problem Statement

In this section, we provide background on how the roles of clients differ between non-fault-tolerant and crash-tolerant systems on the one side, which currently constitute the vast majority of systems used in production, and BFT systems on the other side, which in recent years have been widely studied and are now ready to be applied in practice. Based on this comparison, we then discuss the implications of moving from existing system architectures to BFT replication from a client-implementation perspective, thereby explaining the inherent difficulties that so far prevented the migration to BFT for many real-world use-case scenarios. Finally, we outline our approach to address these problems with Troxy by introducing a trusted proxy component at the server side that allows legacy client implementations to remain unchanged.

A. Clients in Different System Architectures

The means necessary for a client to access a network-based service in general depend on how the service is implemented at the server side. As illustrated in Figure 1a, in the most simple



(a) Unreplic. (b) Crash-tolerant replication (c) BFT replication

Fig. 1. Differences in client perspectives: While the client of an unreplicated or crash-tolerant system usually only interacts with a single server/replica, a BFT client communicates with all replicas in the system.

case of a non-fault-tolerant system containing only a single server, a client first queries a location service (e.g., DNS) to obtain the server's address and then directly establishes a connection to the server. Using this connection, both sides then subsequently interact with each other based on a specific protocol, for example, HTTP for a web service. Many services rely on secure channels to protect the client-server communication. These channels, such as TLS, handle authentication and encryption/decryption of the exchanged data.

In systems where the server side is replicated to provide resilience against crashes, each client usually also only maintains a connection to a single server at a time (see Figure 1b). To prevent bottlenecks, such systems typically ensure that client connections are distributed across the different servers available. One way to achieve this in a transparent manner for the client is, for example, to introduce a load balancer [16], possibly integrated with the location service. Such a mechanism also ensures that in the event of a replica crash the affected clients are automatically reassigned to other replicas once they try to reconnect to the service.

In contrast to clients in unreplicated or crash-tolerant systems, clients in BFT systems not only need to implement the service's protocol but also require a voting component for safely accessing the server side [1], [2], [3], [4], [6], [13]. This is due to the fact that a BFT client cannot trust a single replica, because the replica might be faulty and therefore possibly ignores requests or provides erroneous replies. To address this issue, as illustrated in Figure 1c, BFT clients do not only contact a single replica but instead establish connections to all replicas in the system. As a consequence, they are able to verify the correctness of a result by comparing the replies of different replicas. This means that, although the specific communication patterns of clients and replicas vary between BFT systems, in general a BFT client requires knowledge about the identity of replicas in order to be able to distinguish their replies. Usually, such information is provided to the client at configuration time. Many BFT systems exploit this knowledge to establish a dedicated shared secret between each client and each replica, which is then used to authenticate the exchanged messages and therefore, amongst other things, allows a client to verify that a received reply indeed originates from the presumed replica.

B. Problem Statement

Most of the systems and services in production today are either unable to tolerate faults or are only resilient against crashes, resulting in outages or unwanted behavior in situations where Byzantine faults actually occur [17], [18], [19]. One reason for this, despite the recent advances in BFT research, is the fact that there is a plethora of legacy client implementations for which migrating to BFT would produce significant costs. On the one hand, this includes the efforts required for modifying existing client libraries in order to allow them to tolerate Byzantine faults; even worse, in many cases the necessary changes are not limited to the client itself because, as discussed in Section II-A, BFT clients have to be aware of both the identity as well as the number of replicas, and providing this information to clients is usually not straightforward if the overall system has not been designed to publicly reveal such knowledge. On the other hand, migrating to BFT also comes with an increased network and processor usage at runtime due to the client's need to receive, authenticate, and compare multiple replies for each operation. Such an increase in resource usage especially poses a problem to clients with low-bandwidth connections or limited processing power. This, for example, includes clients running on mobile devices. To summarize, making existing client implementations ready for BFT does not only lead to costs for the actual migration, but also results in runtime overhead, which explains why this step so far has not been taken for many real-world systems.

C. The Troxy Approach

To circumvent the previously described problems associated with adding and operating BFT mechanisms at the client side, our approach is to introduce a trusted proxy, or *Troxy* for short, into the system that acts as a representative of the client at the server side and allows legacy client implementations to benefit from Byzantine fault tolerance without requiring modifications. Furthermore, due to the fact that the Troxy is transparent to the client and handles all BFT-related tasks such as reply authentication and voting, this solution does not incur additional network or processor usage at the client.

As shown in Figure 2, the unmodified client in a Troxy-backed system only establishes a connection to a single Troxy instance, which then handles the communication with the replicas in the system for all of its clients. If at one point a Troxy instance fails, the affected clients reestablish their connections to the service as they would do in a traditional system, for example using a location service (see Section II-A), thereby switching to different Troxies. In contrast to all other replica components, which are untrusted and may fail in arbitrary ways, Troxies are trusted and assumed to only fail by crashing. To justify this trust, we run each Troxy inside the trusted subsystem that is provided by modern processors based on technologies such as Intel SGX [9], which guarantees the integrity of the executed program code. In addition, to protect the communication of a client with the service, a Troxy supports the establishment of secure channels using TLS.

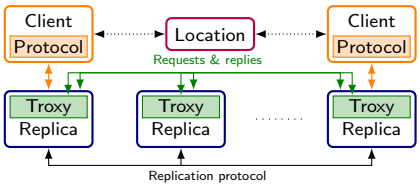


Fig. 2. Architecture of a Troxy-backed BFT system.

In summary, by offering clients transparent access to BFT systems, our approach greatly facilitates the migration of existing services to Byzantine fault tolerance, because legacy client implementations can be reused without modifications or additional resource overhead. At the same time Troxy requires only moderate integration effort into the underlying BFT system at specific extension points.

III. Troxy System Design

In this section, we present details on the design of a Troxy-backed BFT system in general and on the trusted proxy in particular. For clarity, we postpone the discussion of the fast-read optimization to Section IV.

A. Overview

Figure 3 shows an overview of the different components of a Troxy and illustrates how they conceptually interact with each other and with other system components outside the Troxy. When a client issues a request to the service through a secure channel, the Troxy first decrypts the message (①). For a read request, the Troxy then executes the fast path for reads (②) and in case of success immediately returns the cached reply (see Section IV). For a write request or in case of a read-cache miss, the Troxy forwards the client request to its local replication logic to invoke the BFT agreement protocol (③), thereby itself assuming the role of a BFT client. Having received the request, the BFT protocol distributes the request to the other replicas in the system and ensures that all correct replicas execute all client requests in the same order. After processing the request, each replica returns the corresponding reply to the replica the client is connected to, where the Troxy's voting component then determines the correct result by comparing the replies of different replicas (④). To tolerate f faults, the voter waits until having obtained $f + 1$ matching replies from different replicas before returning the result to the client (⑤) as this guarantees that at least one of the replies stems from a non-faulty replica and is therefore correct. In summary, by acting as a BFT client for the replication protocol a Troxy already assumes all the additional responsibilities necessary to access a BFT service, freeing the client from the need to perform these tasks itself.

B. System Model

The Troxy approach relies on a hybrid fault model [13], [14], [15], [20], [21], [22], [23] in which a system is a collection of components with different resilience characteristics. All Troxies in the system are assumed to either operate correctly

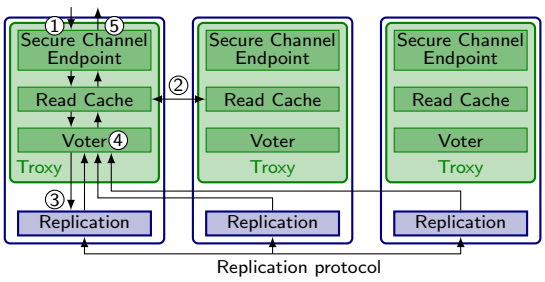


Fig. 3. Overview of Troxy components and their interactions.

or to fail by crashing; in particular, this means that once a client receives a result from a Troxy over a secure channel, the client can trust the result to be correct. In later sections, we discuss how we ensure this trustworthiness for Troxies based on minimizing a Troxy's trusted computing base (see Section III-C) and utilizing Intel SGX (see Section V).

Apart from Troxies, all other replicas and network components in the system may fail in arbitrary ways. The number of servers required in a Troxy-backed system to tolerate such Byzantine faults depends on the BFT replication protocol executed among replicas: using a traditional BFT protocol [1], [4], [24], [25], a minimum of $3f + 1$ replicas are necessary to tolerate up to f faults. If a replication protocol itself makes use of trusted components [13], [14], [15], [20], [21], [22], [23], this number can be reduced to $2f + 1$ replicas. Replica components located outside the Troxy do not trust each other. Components of different replicas communicate by exchanging authenticated messages over the network. If a correct component receives a message it cannot verify, the component discards the message.

C. Minimizing the Trusted Computing Base

Relying on a hybrid fault model, it is crucial to keep the trusted components as small as possible [21], because the more complex a component, the more likely it is to fail in an arbitrary way, for example, as the result of a program error. To justify the trust put in the Troxy, we therefore minimize its complexity by only performing those tasks inside the Troxy that are critical and actually require to be trusted; in contrast, all noncritical tasks are executed outside the Troxy in the untrusted part of the replica. In essence, this leads to a design where the Troxy is basically a library whose functionality is used by the untrusted replica part via method calls.

With regard to client communication, this separation of critical and noncritical tasks means that most of the network-connection handling can be performed outside the Troxy. In particular, this includes the management of connected sockets, the handling of worker threads operating on these sockets, as well as the execution of the actual send and receive operations. Overall, there are only three major critical tasks the trusted Troxy needs to perform: (1) when the client connects to a replica, the replica's Troxy controls the establishment of the secure channel and afterwards stores the associated session key in order to prevent the untrusted part of the replica from being able to impersonate the Troxy. (2) When the client sends a request to the server over the secure channel, the untrusted part of the replica receives the request message. However, the Troxy is the only one to be able to decrypt the request using the session key. Having decrypted the message, the Troxy then checks its integrity and creates a BFT-protocol request in which it includes the client request as payload. Finally, the Troxy authenticates the BFT request using the method expected by the underlying BFT replication protocol (e.g., a keyed-hash message authentication code (HMAC) in our implementation) before handing over the request to the untrusted part of the replica. This way, by atomically decrypting the client request and creating a corresponding authenticated BFT request, the Troxy ensures that the request cannot be altered by the untrusted replica part without being detected. (3) After the request has been executed, the Troxy collects the replies provided by different replicas, verifies the authenticity of these replies, and then compares them to determine the correct result. Based on this result, in a final step, the Troxy creates a reply to the client and encrypts this message using the session key of the client's secure channel. The actual transmission of the reply is performed outside the Troxy in the untrusted part of the replica. However, due to the untrusted replica part not having access to the session key, it is unable to manipulate the reply without the client detecting such a modification.

D. Fault Handling

When a Troxy returns a reply to the client, the client can trust the reply to be correct. However, in case of faults there can be situations in which a client at first does not receive a reply to its request, for example, due to the server hosting the Troxy having crashed. To handle such scenarios where a Troxy ceases to operate, we exploit the fact that clients of user-facing services typically are already equipped with a mechanism to automatically reconnect to the service once their existing connections time out, for example, relying on an external location service to assist in the failover to another replica (see Section II-A). As soon as the client reaches a non-faulty replica, after retransmitting the request, the client will eventually receive a corresponding reply from the service.

Using the same failover mechanism, clients are also able to tolerate scenarios in which the untrusted part of a replica, which performs the actual send and receive operations on network connections (see Section III-C), fails to deliver the correct reply provided by the Troxy. Depending on the nature of the fault, in such case the client either detects a corrupted channel (if the untrusted part sends data that is not encrypted with the Troxy's session key) or experiences a timeout (if the untrusted part sends no data at all). Either way, the client can solve the problem by reconnecting to the service.

In contrast to the Troxy, the untrusted part of a replica may fail in arbitrary ways. Apart from the scenarios discussed above, handling these kinds of faults mainly lies in the responsibility of the underlying BFT replication protocol, as it is the case in traditional BFT systems. The fact that a Troxy, while acting as a BFT client, is co-located with a BFT replica has no effect on

the internal fault-handling procedures of the protocol. Replay attacks are prevented by the secure channel that connects the client with the Troxy. By design, each endpoint will never accept the same chunk of encrypted data twice.

E. Introducing Byzantine Fault Tolerance Using Troxies

In the following, we illustrate the steps necessary to migrate an existing user-facing service that is implemented by a crash-tolerant system to a Troxy-backed BFT system. As an example, we consider a RESTful web service that originally relies on Paxos [26] for fault tolerance and is accessed by a wide spectrum of heterogeneous clients via HTTPS.

The first step to make such a service Byzantine fault tolerant using our approach is to select a BFT replication protocol and to integrate its server-side implementation with the Troxy. This task is greatly facilitated by the Troxy essentially being a library that needs to be invoked at a small number of well-defined locations in the replica logic in order to be able to establish secure channels, to safely translate incoming client requests into BFT requests, and to determine and encrypt the final replies (see Section III-C). On the other hand, the most complex parts of a BFT protocol implementation, such as the ordering and view-change protocols, are left unmodified.

In a second step, the server-side application logic of the web service must be ported from the original crash-tolerant protocol to the BFT protocol. For this task, it is usually possible to benefit from the fact that BFT protocols and crash-tolerant protocols such as Paxos or Raft [27] in general provide comparable interfaces and pose similar requirements on applications, for example, with regard to execution determinism or the ability to create/apply checkpoints of their state.

To enable the Troxy to communicate with clients, in a final step, the Troxy must be made aware of the message format used by the service for requests. In this context, there is no need for the Troxy to fully parse and understand incoming requests. Instead, it is sufficient for the Troxy to identify request boundaries in order to be able to properly store the incoming client request in the newly created BFT request; for replies, the Troxy usually can simply extract the payload contained in the verified BFT result and return it to the client. For many communication protocols, including HTTP, identifying message boundaries is straightforward due to messages carrying information about their own length.

The steps discussed above have shown that the migration overhead is small if a service is already resilient against crashes. However, with Troxy providing transparent access to BFT systems, even for unreplicated services that so far offer no fault tolerance at all, the changes necessary to integrate Byzantine fault tolerance are limited to the location service (i.e., to make it replication aware) as well as the server-side implementation. In contrast, there is no need to modify the potentially large number of diverse client implementations.

IV. FAST-READ CACHE

Troxy features a *managed fast-read cache* that not only validates cache entries when processing regular read requests, but also removes entries from the cache if a write request is about to outdate cached data. As a key benefit, by invalidating cache entries while processing write requests and before their effects are emitted to clients, Troxy is able to maintain consistency guarantees offered by the underlying BFT protocol. In the following, we present details on Troxy's fast path for reads using the example of a BFT system that is based on a hybrid fault model and therefore can tolerate f faults with $2f + 1$ replicas, as it is the case for our prototype implementation (see Section V-B).

A. Protocol

In line with previous research [3], [4], [5], our fast-read optimization assumes that read and write requests can be distinguished before executing them and that it can be determined which part of the state a request is about to access or modify. The described functionality is executed inside a Troxy instance and therefore trusted with the exception of functions that are provided by the surrounding replica.

Our fast-read cache utilizes the processing of a write request to remove an outdated entry from the cache before the effects of the write are visible to any client, that is, before the reply to the write is returned to its client. To ensure this, we make two important changes to introduce the cache: (1) We modify the voter to only take the reply of another replica into account if the reply is authenticated by the other replica's Troxy. As a consequence, this requirement forces a replica to hand over a reply to its local Troxy in order for the reply to have an impact on the final result, thereby giving the Troxy the opportunity to learn about a write and to subsequently invalidate an outdated cache entry. To authenticate a local reply, a Troxy computes an HMAC that is based on a shared secret, which is known amongst all Troxies, and an identifier specific to each Troxy instance. (2) We extend the replies provided by local replicas to not only contain the application's result but also (a hash of) the original request in order to allow a Troxy to identify the cache entry to invalidate. As before, a Troxy only returns a result to the client after having received $f + 1$ *matching replies* (which now include the request) from different replicas. With regard to the fast-read cache, this means that when a write reply reaches this point, it is ensured that a majority of replicas in the system have invalidated the associated cache entry.

As shown in Figure 4, if a Troxy receives a read request from a connected client it first determines if the fast-read cache can be utilized by calling *check_cache* that takes the client-provided request as an input. Next, it checks if the cache contains data that answers the request. If not, the request is ordered and executed as any other request. Otherwise, a set of f remote Troxies is randomly chosen and queried using *get_remote_cache_entry(r,req)*. This function generates an authenticated message for replica *r* to query its Troxy about the currently processed request, which is handed over to the untrusted replica code for transmission. On the remote side, the receiving Troxy instances validate the message and then check if the requested data is cached (see L. 21, Figure 4). The request and associated reply, both authenticated, are returned

```
1 // Cache lookup in case of voting Troxy instance
2 upon call check_cache(req) such that req is READ do
3     reply := cache.get(id(req))
4     if reply is not NULL // request is cached
5         replicas := choose_f_replicas() // select f remote caches
6         rc := ∅ // set of remote cached replies
7         // collect cache entries of f remote replicas
8         ∀r ∈ replicas, rc.add(get_remote_cache_entry(r,req))
9         // remote caches match local cache
10        if ∀(r_req, u_rep) ∈ rc, (id(r_req), u_rep) = (id(req), reply)
11            return reply // fast read succeed
12        else return null // mismatch amongst caches
13    else return null // cache miss

15 // Cache lookup in case of remote Troxy instance
16 upon call get_local_cache_entry(req) do
17    reply := cache.get(id(req))
18    return (req,reply)
```

Fig. 4. Cache lookup when processing read requests.

to the initial requesting Troxy. Next, it is validated if all f request and reply pairs match the local data. If this is the case, the reply is returned to the client and a successful cache lookup has been performed. In case of a mismatch, which for example can be the result of concurrent write requests or actions performed by malicious replicas (e.g., the replay of a stale reply), the read request is ordered in the common way.

Note that a more aggressive use of hashes can reduce the amount of exchanged data. In addition, timeouts might be used to detect unresponsive replicas.

B. Ensuring Consistency and Resilience to Performance Attacks

In scope of the implemented prototype we considered a system that relies on an hybrid fault model that requires only $2f+1$ replicas and offers strong consistency. The aim of Troxy and its fast-read cache is to preserve the guarantees offered by the underlying protocol. This is achieved by immutable entangling the maintenance of the fast-read cache with the protocol execution, so an attacker cannot diverge replicas and Troxies to make conflicting statements. With a total amount of $2f+1$ replicas in the hybrid fault model, completing a write operation takes a quorum of $f+1$ replicas for providing authenticated replies. Since reply authentication is done by Troxy inside the trusted subsystem, these $f+1$ replicas must have deleted the related entry in their fast-read cache before the reply becomes visible to any client. Meanwhile a successful fast-read operation also needs $f+1$ identical entries, meaning that at least $f+1$ replicas must still contain a matching entry in their caches. This is not possible as both quorums intersect by one replica and its trusted Troxy is responsible for providing the necessary response to either side. By doing so, a successful fast-read is ensured to reflect the state of the latest write. One option for an attacker would be to roll-back the trusted subsystem by a reboot, however in this case the cache would simply lose its entire state and queries are returned unanswered, which will result in the execution of the underlying protocol. In general the forwarding of a reply due to a write request always result in a cache *invalidation* but not in a cache update. This is necessary as the local Troxy can confirm the origin of the reply but not its correctness, thus a faulty replica should not be able to pollute the cache.

This leads to the question if a faulty replica can negatively impact the performance beyond its capabilities in a traditional system. This is not the case as for classical BFT systems like PBFT [1] that feature a read optimization where $2f+1$ replicas are queried, a client can only utilize the result if all replies match. Thus, faulty replicas can return wrong results and frequently prevent a successful read optimization. In case of Troxy we are in a similar situation, as we query f randomly chosen Troxies for their cache entries. However, additionally we measure the cache miss rate inside the Troxy. If the miss rate reaches a configurable system constant, the fast read optimization is avoided in favor of a traditional protocol run. As shown in the evaluation this also addresses the case of write contention, where a lot of cache misses occur due to conflicts.

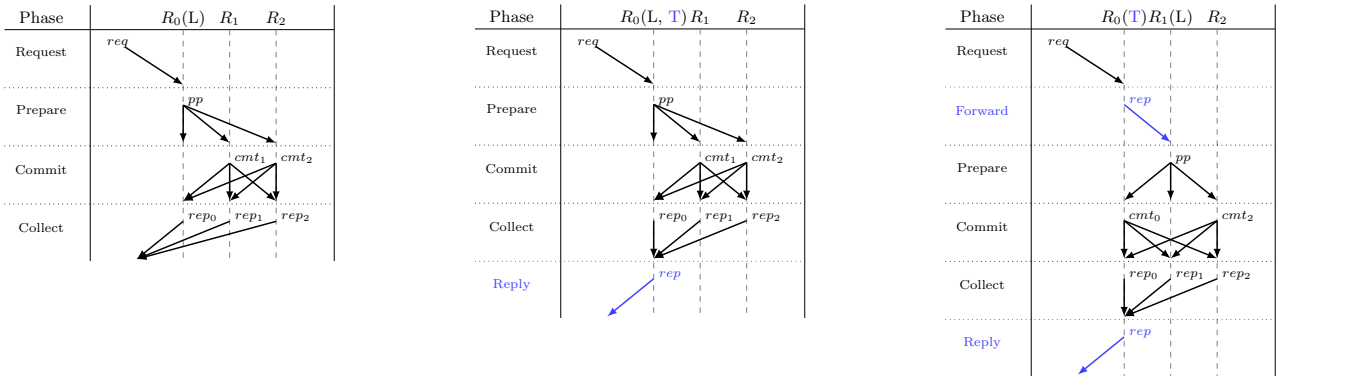
V. Implementation

Below, we present our prototype of a Troxy-backed system, providing details on the SGX-based Troxy implementation as well as its integration with the BFT protocol Hybster [13].

A. Troxy Implementation

Our Troxy implementation is written in C/C++ and relies on Intel's Software Guard Extensions (SGX) [9] and its SDK [28] to achieve isolation between the trusted and the untrusted parts of a replica. The Troxy runs inside a trusted execution environment provided by SGX, a so-called *enclave*, that is protected by the CPU via transparent memory encryption and integrity checking. To enter and exit an enclave, the only possible way is to go through an enclave interface which defines the entry points and the maximum number of concurrent threads allowed at any point in time inside the enclave. An enclave call (*ecall*) is needed for calling the enclave functions, while an outside call (*ocall*) is explicitly used for calling from an enclave to the untrusted environment. An *ecall* leads to executing a TLB flush, switching to a trusted stack located inside the enclave, copying the parameters from untrusted memory and calling the trusted function. Similarly, an *ocall* causes a TLB flush, switching back the untrusted stack, moving parameters out of the trusted memory, and exit of the enclave. Due to their high overhead, it is best practice to minimize enclave transitions.

Troxy implements *ecalls* for data transfer between enclaves and the untrusted environment as well as for data processing inside enclaves. In order to keep the interface small, Troxy defines only 16 *ecalls* and no *ocalls* under a security-aware programming model. More precisely, these *ecalls* have been manually verified and are hardened to prevent possible attacks such as Iago attacks [29] or time-of-check-to-time-of-use attacks [30]. For example, the data transfer between the untrusted environment and enclaves requires additional copies of the message buffers. A read buffer is always directly copied into the enclave to avoid time-of-check-to-time-of-use attacks; in contrast, the copy of a write buffer can be done outside the enclave to achieve better performance.



(a) Message flow in Hybster.

(b) Message flow in Troxy-backed Hybster when the client is connected to the leader replica.

(c) Message flow in Troxy-backed Hybster when the client is connected to a follower replica.

Fig. 5. Comparison of the message flows in Hybster and Troxy-backed Hybster.

To enforce the validity of enclaves, Intel provides a remote attestation service [9]. In a nutshell, a hash of the memory pages of the enclave is securely computed and sent to the remote attestation service so that the user can obtain a proof that the enclave has been initialized correctly. Once the enclave has been correctly attested it is possible to provision it. Any cryptographic key and secret, such as the private key used by Troxy to initialize a secure connection with the clients, can be securely sent to the enclave during the provisioning phase.

The enclave code and data is stored in the Enclave Page Cache (EPC), a specific region of memory protected from untrusted accesses. In the current implementation of Intel SGX, this memory area has a maximum size of 128MB. Accessing memory beyond the size of the EPC results in costly paging, as the pages need to be encrypted and integrity-protected before being evicted to main memory. As this operation incurs a high performance overhead [31], we limit memory allocations to keep the memory footprint as small as possible. Furthermore, to avoid additional *ocalls* and paging [32], the Troxy can store data in an encrypted manner outside the enclave. When it needs to be accessed, it is directly read from the untrusted memory and validated by comparing it against a hash securely stored inside the Troxy.

Finally, Troxy provides bidirectional TLS authentication to all messages exchanged between clients and replicas. For this purpose, Troxy uses the TaLoS [33] library, which exposes a TLS interface to existing application while securely executing the TLS logic inside an Intel SGX enclave. Note that we run it in a completely encapsulated manner: there are no *ecalls* nor *ocalls* between TaLoS and the untrusted environment.

B. Troxy-backed Hybster

To provide fault tolerance, our prototype implementation relies on Hybster [13], a BFT replication protocol that is based on a hybrid fault model and therefore only requires $2f + 1$ replicas to tolerate f Byzantine faults. Hybster is implemented in Java and uses Intel SGX to realize a trusted susbsystem for message authentication. It achieves high performance via parallelization, where the performance scales well along with the number of NICs and CPU cores. The trusted subsystem of Hybster is also used by Troxy for trusted authentication upon internally exchanged messages during the ordering phase. In our implementation the interaction between the protocol running in the untrusted part of the replica and the SGX enclave is handled via the Java Native Interface (JNI).

Hybster is a leader-based BFT protocol: a special node is in charge of proposing an ordering on the requests received by the clients. Figure 5 shows the message flow in the resulting Troxy-backed system. Compared with the original Hybster (see Figure 5a), introducing the Troxy adds one message delay for a client that is connected to Hybster's leader replica (see Figure 5b). In this extra phase, the corresponding Troxy collects and compares the replies to the client's request in order to determine the correct result. For clients connected to servers hosting Hybster followers, an additional phase is necessary to transmit the request to the leader, as only the leader is able to initiate the agreement process for requests (see Figure 5c). Note that for a setting in which the replicas of a system are hosted in different fault domains inside the same data center (e.g., different racks with independent power and network supply [34]), the additional messages only have a minor impact on the overall latency experienced by the client.

Apart from highlighting individual message flows, Figure 5 also illustrates another important difference between traditional BFT systems and a Troxy-backed BFT system: with the Troxy performing reply voting at the server side, the client receives only a single reply per request. In practice, this approach has several key advantages: First, in a typical setting where clients are connected to the service over a wide-area network, less data has to be sent over long-distance links, which is especially beneficial for low-bandwidth clients. Second, during periods of unstable (wide-area) network conditions it improves the response time of the service due to the latency experienced by the client no longer depending on the arrival of the $f + 1$ slowest (normal request) or $2f + 1$ slowest (read optimization) matching reply. Third, and most important, it makes the BFT replication system transparent to clients.

VI. Evaluation

In this section we evaluate the performance of Troxy compared to Hybster using both microbenchmarks and an HTTP service. The results show that: (1) For ordered small-payload messages in a local network, Troxy has an overhead of at most 43% due to its extra communication steps (see Figure 5) and trusted environment transitions. (2) For larger messages with network delay, Troxy improves the performance compared to Hybster by at most 70%. (3) For read-heavy workloads with network delay, the fast-read cache optimization improves the throughput by 130% even in the presence of conflicting write requests. (4) When considering an HTTP service with network delay, Troxy can almost hide the replication cost, allowing clients to observe similar latency as for a non-replicated service.

A. Experimental Setup

The measurements are conducted on a cluster of five identical machines connected via four 1 Gbps Ethernet NICs. Each machine is equipped with an SGX-capable Intel Core i7-6700 quad-core processor running at 3.4 GHz with Hyper-Threading activated as well as 24 GB of memory. Three machines are dedicated to the replicas (hence we consider $f = 1$ faults) while the two remaining ones are running as clients. All the machines are running 64-bit Ubuntu 16.04 with a Linux kernel 4.4.0, OpenJDK 1.8 and the Intel SGX SDK v1.9. We compare the performance of our Troxy-backed Hybster variant with the original Hybster protocol, noted as BL (for *baseline*).

B. Security Analysis

In this section we analyse the security of Troxy.

Performance attacks: A malicious replica could try to return old cache entries in the case of the fast-read cache optimization. As a result the fast read would fail, slowing down the protocol. As discussed in Section IV-B, Troxy selects f random replicas to reply to a fast-read query and monitors the cache miss ratio to address such attacks.

Side-channel attacks: We consider side-channel attacks out of the scope of this paper. However, Troxy can implement existing technics to limit side-channel attacks inside an SGX enclave [35], [36], [37].

Bypassing Troxy: A malicious replica could bypass Troxy in order to break the safety of the system, by directly communicating with the clients. To prevent this attack the clients and Troxy initiate secure connections using the TLS protocol. The session keys are securely stored inside the Troxy, thus the malicious replica cannot forge correct messages.

Interface attacks: A malicious replica could attack the enclave interface in order to get access to the secrets stored inside the Troxy. As discussed in Section V, the enclave interface has been hardened to prevent such attacks.

Denial-of-Service and flooding: A malicious replica could decide to perform a Denial-of-Service attack, not executing the Troxy or following its protocol, or at the opposite flood the correct replicas or clients with invalid messages. In all these cases the goal of the malicious replica is to render the system not usable. Troxy can leverage existing techniques [38] to prevent such attacks.

C. Microbenchmark

We created a microbenchmark to evaluate the full capacity of Troxy and to investigate the overhead of (1) relocating the traditional client-side library to the server side and (2) using the trusted subsystem for protection of the Troxy. A configured number of clients are created to constantly issue asynchronous requests and measure the average throughput and latency for 60 seconds. The final results are the average values of three runs. Batching is not used as it is an orthogonal approach that has independent influence to the results.

Secure socket connections are applied to the client-to-replica communication for both the baseline and Troxy, while the replica-to-replica communication keeps using plain sockets and HMACs for message authentication. Clients only connect to the leader in the baseline system, while Troxy allows connections to any replica. We created a simple service that accepts requests and generates a reply message of configurable size. Read and write requests can be distinguished by their operation types. We ran experiments in three different scenarios, where (1) write requests are totally ordered; (2) read optimizations are applied to handle read-only requests, and (3) concurrent write requests cause conflicting reads, which leads to the traditional ordering of conflicting read requests.

In addition to the local network configuration, we also simulate a wide-area network by adding 100 ± 20 ms (in a normal distribution model) delay to the NICs of the client machines. We consider this as the typical usage scenario of Troxy, that is, data-center-hosted services that are accessed by remote legacy clients.

1) Totally Ordered Requests: In this scenario, we consider write requests of different sizes: 256 B, 1 KB, 4 KB and 8 KB. The size of the reply is always 10 B. Two implementations of Troxy in C/C++ are compared against the baseline: *ctroxy*, running in the untrusted environment without SGX, indicates the impact of using JNI; while *etroxy*, running inside an enclave, adds the overhead of utilizing the trusted subsystem.

Figure 6 shows the measurement result for handling write requests in the local network. With a small request payload size (256 B), etroxy shows about 43% of performance loss due to the transitions between the trusted and untrusted environments as well as the extra steps in processing ordered requests (see Figure 5). More precisely, by considering the performance of ctroxy (without SGX), half of the performance loss in etroxy is caused by using the trusted subsystem. When the payload size increases, ctroxy and etroxy start to provide similar performance and etroxy reaches the baseline at 8 KB. This is due to the fact that authenticating messages with large payload is faster in C/C++ than it is in Java.

We also measure the performance with a network delay in between the clients and replicas. As illustrated in Figure 7, the server-side reply voter brings a huge advantage to Troxy. In this case, for each request, the clients wait for only one reply that is affected by the delay instead of $f + 1$ replies. This advantage applies to different request payload sizes, and leads to up to 60% performance gain.

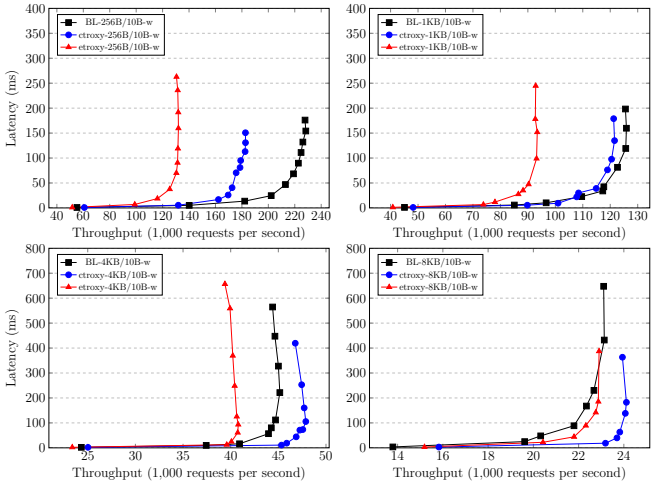


Fig. 6. Handling totally ordered write requests in the local network.

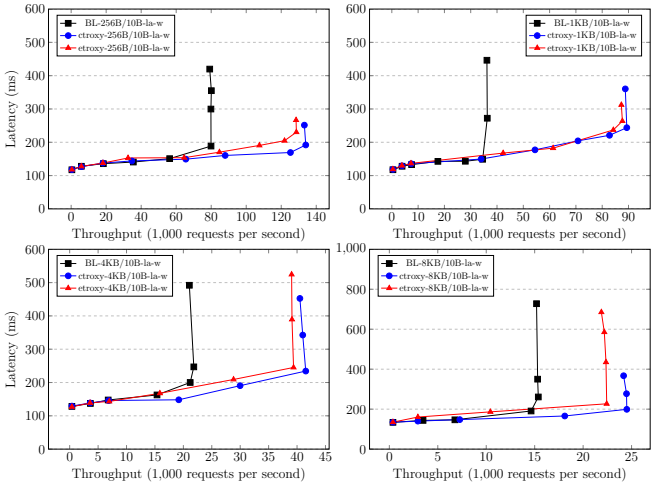


Fig. 7. Handling totally ordered write requests with network delay.

2) Read Optimizations: We measure the performance of the fast-read cache using read-only requests with different payload sizes: 10 B / 256 B, 10 B / 1 KB, 10 B / 4 KB and 10 B / 8 KB for request / reply messages, respectively. The baseline system implements a PBFT-like read optimization approach [1], where read requests are directly forwarded to the followers for execution without being ordered. For read-only workloads, this approach can be very effective as there are no concurrent state transitions to create conflicts in the read results.

Figure 8 shows the results of handling read-only requests in the local network. On the one hand, with small requests (10 B), the fast message authentication cannot compensate the overhead of the server-side reply voter. The overhead with 256 B reply is as high as 115%. On the other hand, along with the increasing reply size, the effect of fast authentication becomes more visible. With 4 KB replies etroxy can already overtake the baseline, and at 8 KB we can observe about 30% throughput improvement.

The result of the measurement with a network delay is shown in Figure 9. Although the server-side reply voter adds overhead to Troxy, the extra network delay has less impact on Troxy's performance. Compared to the baseline, with 256 B replies etroxy only incurs a 33% performance degradation with network delay, compared with 115% without network delay. In addition, as the fast-read cache only needs to transfer the hash of the reply between replicas for a fast-read operation instead of a full reply, this further reduces the authentication and transmission cost. When the reply size is above 1 KB, etroxy outperforms the baseline by at least 15%.

3) Concurrency Handling: In this scenario, 1% of write requests are generated among the reads, to introduce concurrent state transitions during fast-read operations. Due to different read optimization approaches, the 1% write workload results in different read conflict rates for the baseline and Troxy (only etroxy is evaluated in this scenario). For the baseline,

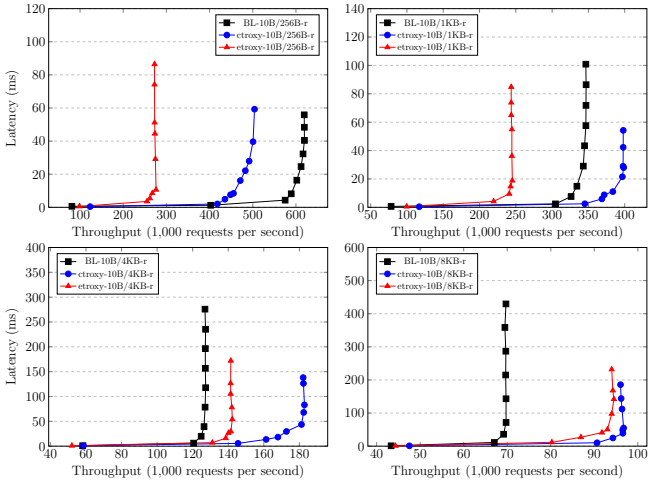


Fig. 8. Handling read-only requests in the local network.

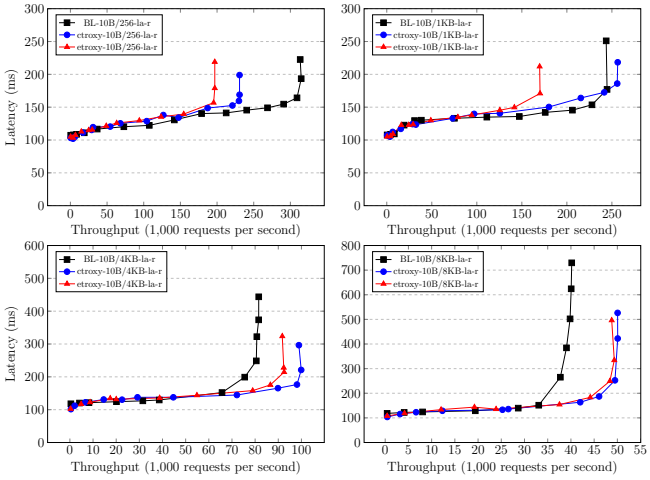


Fig. 9. Handling read-only requests with network delay.

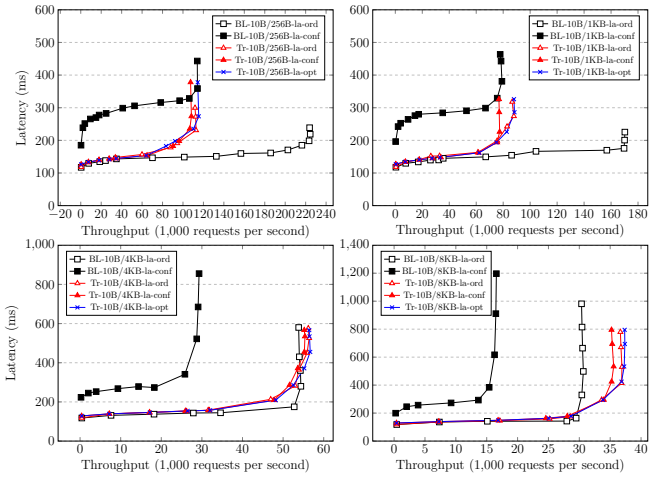


Fig. 10. Handling read conflicts with network delay.

nearly 50% of reads return conflicting results and have to be ordered for a second time of processing, adding substantial extra overhead to the system. As for Troxy, the fast-read cache acts in a conservative way: When it uses write requests to invalidate existing cache entries, the later read requests will be ordered to prevent conflicts. This way, the observed conflict rate goes down to 14%.

We also conducted a measurement where no optimization is applied so that all reads are ordered, to get a reference throughput of each system for comparison. Figure 10 illustrates that the overhead of having 50% read conflicts contributes to the significant performance loss of the baseline, resulting in the read optimization to only achieve half of the reference throughput. For Troxy, the 14% read conflicts also decreases performance to a point that is slightly lower than its reference throughput. Therefore, we further optimized the approach to monitor the conflict rate inside Troxy in order to ensure that once the conflict rate goes beyond a certain threshold, Troxy will automatically switch to the total-order mode where all requests will be ordered (see Section IV-B). This threshold can be learned by sampling the system to determine at which conflict rate the benefits gained by fast reads will disappear. This way, the optimized fast-read cache can guarantee the lower-bound performance in case of frequent conflicts.

D. HTTP Service

In addition to the microbenchmark, we created a simple, replicated HTTP service that handles HTTP GET and POST requests and returns the queried or modified pages as responses. Its performance is measured with the HTTP benchmarking tool Apache JMeter [39]. As we are interested in evaluating the overhead of using a BFT system and the trusted subsystem in a latency-sensitive application, we ensure that JMeter is configured not to saturate the replicas, launching 100 clients to issue a total of 500 requests per second.

We measure the performance of the HTTP service in three implementations: (1) with the baseline protocol; (2) with

TABLE I
SUMMARY OF READ OPTIMIZATION APPROACHES.

	Replica	Quorum	Consistency
BL	$2f+1$	$f+1$ replicas	Strong
Prophecy	$3f+1$	1 replica + middlebox	Weak
Troxy	$2f+1$	$f+1$ replicas	Strong

Prophecy [5], a middlebox-based approach that mimics clients towards the BFT replicas and is tailored to improve the performance of read-heavy workloads; and (3) with Troxy. Table I summarizes the three implementations regarding their read optimization approaches and consistency level.

The baseline protocol implements a PBFT-like read optimization, which optimistically executes non-ordered read requests and accepts a result as soon as $f+1$ identical replies are received. In case of a failed quorum due to concurrent write operations, the client has to resend the request and ask for a regular ordering to enforce linearizability. Prophecy deploys a cache in a middlebox placed between the client and the replicas. This cache stores the results of the ordered reads to reduce the execution cost of read requests with large payloads for read-heavy applications. It requires only one reply from a randomly chosen replica to be compared with the cached result. However it trades consistency for a higher throughput: the reply of a read operation reflects the state of the latest *read*, so in the worst case it would return a stale but correct result to the client. In contrast, Troxy actively manages the fast-read cache to reflect the state changes of the latest *write*, thus guaranteeing strong consistency.

For the baseline, we run JMeter on the same machine as the client-side library, and use a local socket connection for message forwarding. As for Prophecy, JMeter is running on a separate machine, and establishes a secure socket connection to the client machine where the middlebox is located. Since Troxy provides transparent access to clients, JMeter can directly connect to the replicas without any modifications. Besides that, we also run a stand-alone version of the HTTP service using Jetty (v9.4) [40] to see its original performance.

The measurements are conducted in two scenarios: in the local network and with 100 ± 20 ms network delay. The GET and POST requests are issued with a payload size of 200 B, while the response message size ranges between 4 KB and 18 KB. The average latency to execute requests

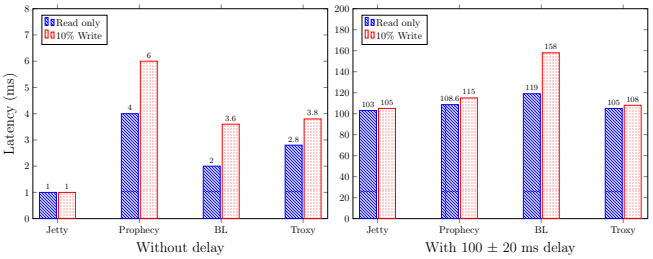


Fig. 11. HTTP service in local network and with network delay.

is reported in Figure 11. In both scenarios, the stand-alone implementation (Jetty) indicates the original performance of the service. In case of a local network, both the baseline and Troxy keep a low latency, with an overhead of at most 1.8 ms, while the two socket connections in Prophecy contribute to a latency almost twice as high. When the network delay is applied, the latency of the baseline implementation raises dramatically, as its reply voter is located on the client machine. The network delay between the client and the replicas significantly impacts the latency observed by the client. For Prophecy and Troxy, as their voters are close to the replicas (on the middlebox machine and in the fast-read cache on a replica, respectively), this extra round-trip impact is negligible. The results of this measurement show that in a wide-area network, using Troxy-backed BFT systems is beneficial for user-facing legacy applications.

VII. Related Work

Traditional BFT state machine protocols consist of libraries attached to both client and server [1], [3], [4], [6], [41]. The client-side library is mainly responsible for service invocation, message transfer, and reply voting. In contrast, Troxy provides a transparent and secure connection between the client and the replicated service by leveraging trusted computing technology. The complexity of the replicated fault-tolerant system, in terms of protocol, exchanged messages, and interface is therefore hidden from the clients and legacy clients can interact with BFT services without any changes.

Troxy is not the first protocol to explore the usage of trusted subsystems in BFT systems. A2M-PBFT [15] is based on a trusted append-only log, enabling it to reduce the number of required replicas compared to traditional protocols from $3f + 1$ to $2f + 1$. TrInc [22] is a subsystem providing trusted counters that can be employed as a less complex replacement for the trusted log of A2M-PBFT. MinBFT and MinZyzzyva [14] are two protocols that directly make use of a counter-based trusted subsystem. The most recent representative of this class of protocols is Hybster [13]. Hybster is also based on trusted counters and $2f + 1$ replicas. However, it overcomes the difficulties of other hybrid protocols such as a time-dependent memory demand and exhibits a significantly improved performance by introducing the consensus-oriented parallelization [2] into the hybrid fault model. Besides using an FPGA-based trusted subsystem, CheapBFT [21] saves resources by exploiting passive replication: f out of $2f + 1$ required replicas remain passive and are activated only in case a faulty behavior is suspected. Similarly, V-PR [23] employs trusted computing technology, named XMHF/TrustVisor [42], to design a fully-passive replicated system for tolerating Byzantine failures. By leveraging a trusted subsystem, all those protocols have a lower complexity, in terms of exchanged messages and number of replicas, compared to traditional BFT protocols. Nevertheless, none of the aforementioned systems are transparent from the client's point of view.

Prophecy [5] executes a special component between the client and the server and thus does not require modifications at the client side. As in Troxy, this component needs to be trusted and acts as a proxy by receiving the client request, collecting the replies from the replicas and sending a single reply back to the client. However, compared to Troxy, Prophecy (i) requires a large trusted computing base comprised of a middlebox, operating system, and network stack; and (ii) is not able to ensure strong consistency.

SPARE [43] is transparent to the clients by locating the reply voter on the server side. SPARE executes replicas inside virtual machines, thus requiring a virtualization layer and hypervisor, and considers a specific fault model where the replicas can exhibit Byzantine behavior; the hypervisor and reply voter fail by crashing only. The practicality of SPARE is limited by its large trusted computing base, composed of an entire hypervisor, a management operating system, and the reply voter.

Thema [44] and BFT-WS [45] extend the classic approach of having a generic client-side library and a server-side library with an additional web-service library. This library collects identical request messages from the different replicas, sends the request to a non-replicated web service, and forwards the reply back to the replicas. Thus, these works address an orthogonal problem and could be combined with Troxy.

Avoine et al. [46] present a deterministic fair exchange algorithm running in untrusted hosts with security modules. The untrusted hosts are unable to forge valid protocol messages due to the security modules comprising the entire consensus-protocol implementation. In contrast, the goal of BFT protocols such as Hybster (the protocol used by Troxy) is to keep the trusted computing base as small as possible by implementing most protocol parts in the untrusted host.

There is a growing number of systems that utilize SGX to secure computing in the context of cloud computing [31], [47], perform application level secure data processing [48], and enable trusted client-side computing and offloading [49], [50], just to name a few. To our knowledge, none of these systems have used trusted execution to enable compatibility with legacy systems as proposed by Troxy.

VIII. Conclusion

We have presented Troxy, a system which leverages trusted execution environments to offer clients transparent access to BFT systems. In contrast to traditional BFT systems, a Troxy-backed system does not require to execute a special library at the client side. Instead, it implements a substitute of the library inside each replica. In addition, it introduces a novel read optimization that features a managed fast-read cache to accelerate read-heavy operations while providing strong consistency guarantees. We implemented a prototype of Troxy in C/C++ with Intel SGX and evaluated its performance with both microbenchmarks and an HTTP service. The results indicate that (1) while Troxy is slower by up to 43% for small payloads, it outperforms a state-of-the-art hybrid BFT protocol by 130% for larger, read-heavy workloads and a realistic network delay; (2) Troxy introduces a negligible latency overhead and is transparent to legacy clients when providing Byzantine fault tolerance to an HTTP service.

REFERENCES

[1] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proc. of the 3rd USENIX Symp. on Operating Systems Design and Implementation (OSDI '99)*, 1999, pp. 173–186.

[2] J. Behl, T. Distler, and R. Kapitza, "Consensus-Oriented Parallelization: How to Earn Your First Million," in *Proc. of the 16th Middleware Conference (Middleware '15)*. ACM, 2015, pp. 173–184.

[3] T. Distler and R. Kapitza, "Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency," in *Proc. of the 6th ACM European Conf. on Computer Systems (EuroSys '11)*, 2011.

[4] R. Kotla and M. Dahlin, "High throughput Byzantine fault tolerance," in *Dependable Systems and Networks, 2004 International Conference on*. IEEE, 2004, pp. 575–584.

[5] S. Sen, W. Lloyd, and M. J. Freedman, "Prophecy: Using History for High-Throughput Fault Tolerance." in *NSDI*, 2010, pp. 345–360.

[6] G. S. Veronese, M. Correia, A. Bessani, and L. C. Lung, "Spin one's wheels? byzantine fault tolerance with a spinning primary," in *Proc. of the 28th IEEE Int'l Symp. on Reliable Distributed Systems (SRDS '09)*. IEEE, 2009, pp. 135–144.

[7] J. Sousa, A. Bessani, and M. Vukolić, "A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform," *ArXiv e-prints*, Sep. 2017.

[8] R. Ferraz, B. Gonçalves, J. Sequeira, M. Correia, N. F. Neves, and P. Veríssimo, "An intrusiontolerant web server based on the distract architecture," in *In Proceedings of the Workshop on Dependable Distributed Data Management*. Citeseer, 2004.

[9] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proc. of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13. ACM, 2013.

[10] "Intel SGX," https://software.intel.com/en-us/sgx, 2017.

[11] "We're Halfway to Encrypting the Entire Web," https://goo.gl/em8elg, 2017.

[12] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[13] J. Behl, T. Distler, and R. Kapitza, "Hybrids on Steroids: SGX-Based High Performance BFT," in *Proceedings of the 12th European Conference on Computer Systems (EuroSys '17)*, 2017, pp. 222–237.

[14] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo, "Efficient Byzantine Fault-Tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2013.

[15] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested Append-only Memory: Making Adversaries Stick to Their Word," in *Proceedings of the twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, New York, NY, USA, 2007. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294280

[16] "HAProxy," http://www.haproxy.org/, 2017.

[17] "Amazon S3 Availability Event," http://status.aws.amazon.com/s3-20080720.html, 2008.

[18] "WhatsApp messenger service suffers major outage," https://goo.gl/uTv3TW, 2017.

[19] "What did OVH learn from 24-hour outage? Water and servers do not mix," https://goo.gl/BabnkY, 2017.

[20] T. Distler, C. Cachin, and R. Kapitza, "Resource-efficient Byzantine Fault Tolerance," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2807–2819, 2016.

[21] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "CheapBFT: resource-efficient byzantine fault tolerance," in *Proc. of the 7th ACM european conference on Computer Systems*, 2012.

[22] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "TrInc: Small trusted hardware for large distributed systems," in *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI '09)*, 2009, pp. 1–14.

[23] B. Vavala, N. Neves, and P. Steenkiste, "Securing Passive Replication Through Verification," in *Reliable Distributed Systems (SRDS), 2015 IEEE 34th Symposium on*. IEEE, 2015, pp. 176–181.

[24] M. Castro and B. Liskov, "Proactive recovery in a byzantine-fault-tolerant system," in *Proc. of the 4th Conf. on Symp. on Operating System Design & Implementation-Volume 4*. USENIX, 2000, pp. 19–19.

[25] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," in *ACM SIGOPS Operating Systems Review*, 2003.

[26] L. Lamport, "The Part-time Parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[27] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*, 2014, pp. 305–320.

[28] "Intel SGX SDK," https://software.intel.com/sgx-sdk.

[29] S. Checkoway and H. Shacham, *Iago attacks: Why the system call api is a bad untrusted rpc interface*. ACM, 2013.

[30] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves," in *European Symposium on Research in Computer Security*. Springer, 2016.

[31] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell *et al.*, "SCONE: Secure Linux Containers with Intel SGX." in *OSDI*, 2016, pp. 689–703.

[32] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless os services for sgx enclaves," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 238–253.

[33] P.-L. Aublin, F. Kelbert, D. OKeeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, and P. Pietzuch, "TaLoS: Secure and Transparent TLS Termination inside SGX Enclaves."

[34] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci *et al.*, "Windows Azure Storage: a highly available cloud storage service with strong consistency," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 143–157.

[35] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating controlled-channel attacks against enclave programs," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[36] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," *arXiv preprint arXiv:1611.06952*, 2016.

[37] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "SGX-Shield: Enabling address space layout randomization for SGX programs," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[38] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults." in *NSDI*, vol. 9, 2009, pp. 153–168.

[39] "Apache JMeter," http://jmeter.apache.org/.

[40] "Embedded Jetty v.9.4," https://goo.gl/cTEMge.

[41] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 45–58.

[42] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB reduction and attestation," in *IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 143–158.

[43] T. Distler, R. Kapitza, I. Popov, H. P. Reiser, and W. Schröder-Preikschat, "SPARE: Replicas on hold," in *Proc. of the 18th Network and Distributed System Security Symposium (NDSS '11)*, 2011.

[44] M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan, "Thema: Byzantine-fault-tolerant middleware for web-service applications," in *Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium on*. IEEE, 2005, pp. 131–140.

[45] W. Zhao, "BFT-WS: A byzantine fault tolerance framework for web services," in *Eleventh International IEEE EDOC Conference Workshop, 2007*, 2007, pp. 89–96.

[46] G. Avoine, F. Gärtner, R. Guerraoui, and M. Vukolić, "Gracefully degrading fair exchange with security modules," in *European Dependable Computing Conference*. Springer, 2005, pp. 55–71.

[47] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data," in *OSDI 16*, 2016.

[48] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, "SecureKeeper: Confidential ZooKeeper using Intel SGX," in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware '16, 2016.

[49] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. Pietzuch, and R. Kapitza, "TrustJS: Trusted Client-side Execution of JavaScript," in *EuroSec'17*, 2017.

[50] K. Mast, L. Chen, and E. G. Sirer, "Scaling Databases Through Trusted Hardware Proxies," in *SysTEX'17*, 2017.