Belisarius: BFT storage with confidentiality

Ricardo Padilha, Fernando Pedone
University of Lugano
Switzerland
Email: {ricardo.padilha, fernando.pedone}@usi.ch

***Abstract*—Traditional approaches to byzantine fault-tolerance have mostly avoided the problem of confidentiality. Current confidentiality-aware solutions rely on a heavy infrastructure investment or depend on complex key management schemes. The framework presented in this paper relies on a novel approach that combines *byzantine fault-tolerance*, *secure storage* and *verifiable secret sharing* to significantly reduce the additional infrastructure and complexity required by confidentiality protection. The proposed framework was compared to other solutions using a micro-benchmark, and an implementation of TPC-B and NFS.**

***Keywords*-byzantine fault-tolerance, confidentiality, secret sharing, storage, cloud computing, security**

I. Introduction

Byzantine fault tolerance has received a resurge of interest in the research community. While many recent works have improved performance (e.g., [1]) and extended failure models (e.g., Byzantine clients [2]), few works have addressed confidentiality, that is, how to prevent information leakage in Byzantine fault-tolerant (BFT) systems. In fact, the original definition of Byzantine faults [3] does not encompass information leakage, where a Byzantine server co-operates (maliciously or not) with unauthorized clients in order to leak information.

Confidentiality is a major issue whenever users rely on third-party services (e.g., storage in cloud computing settings) because of the increased risk of information leakage. Although a majority quorum of honest servers is enough to detect and discard invalid replies, a single malicious server can employ several covert channels to leak information [4], [5]. This makes information leakage an intrinsically harder problem than discarding invalid replies. Given the exposed nature of modern applications [6], confidentiality becomes an essential requirement to deal with servers that may fail due to coding mistakes and hardware problems, and that are also under constant scrutiny by attackers.

There are two main classes of solutions to confidentiality in BFT systems: *obfuscation* and *firewalling*. Obfuscation (e.g., [7], [8], [9]) is a common approach to handling confidentiality of data. Obfuscating stored data involves some cryptographic or secret sharing mechanism. Obfuscation schemes based on cryptographic mechanisms present high overhead and complexity (e.g., the data re-encryption protocols required to revoke a client's rights). Secret sharing schemes have been sparsely employed [10]. Firewalling is an architectural solution, since it prevents unauthorized communication between Byzantine clients and servers. The only firewalling approach to confidentiality in BFT systems we are aware of is the Privacy Firewall proposed in [11], which allows arbitrary operations but requires a significant hardware investment and specific network layout.

This work was supported in part by the Hasler Foundation, Switzerland, under grant number 2316.

In this paper we revisit secret sharing schemes in order to guarantee data confidentiality in a replicated Byzantine fault-tolerant storage service. Belisarius, the system that we designed and implemented, relies on the mathematical properties of Shamir's secret sharing algorithm. Belisarius compares favorably to previous solutions in four aspects: First, differently from a Privacy Firewall, its architecture is economical and simple. Second, although in Belisarius data is obfuscated, the performance impact of obfuscating the data is minimal. Third, Belisarius provides confidentiality without requiring complex and cumbersome key management systems. Fourth, it has very good overall performance (i.e., high throughput and low latency).

Belisarius's performance is credited not only to avoiding complex and computationally inefficient cryptography, but also to its design choice of moving a significant part of the protocol from the servers to the clients. Clients apply a novel approach to cheater detection in secret sharing schemes in the context of Byzantine fault-tolerance. In other words, we opportunistically shift some of the protocol overhead to the clients, resulting in a performance boost. To ensure confidentiality during transport, our framework allows for, but does not mandate, point-to-point encryption of secret data.

In summary, the contributions of this paper are:

- A simple, economical and efficient design for a BFT storage service with confidentiality.
- The application of Shamir's secret sharing algorithm to all data being stored on servers while preserving its additive homomorphism.
- The validation of the system through a detailed performance evaluation using several application scenarios.

The remainder of the paper is structured as follows. Section II describes the system model, including our assumptions about client and server behavior, and the handling of secret data. Section III presents the design of Belisarius. Section IV details its implementation. Section V assesses Belisarius's performance under different setups. Section VI discusses related work, and Section VII concludes the paper.

II. System model and assumptions

A. Clients and servers

We assume an asynchronous distributed system where nodes are connected by a network. There are no known bounds on processing times and message delays. Links may fail to deliver, delay, or duplicate messages, or deliver them out of order. However, links between nodes are fair: if a message is sent infinitely often to a receiver, then it is received infinitely often.

Nodes can be correct or faulty. A correct node follows its specification whilst a faulty, or Byzantine, node presents arbitrary behavior. We allow for a strong adversary that can coordinate faulty nodes, inject spurious messages into the network, or delay correct nodes in order to cause the most damage to the replicated service. However, adversaries cannot delay correct nodes indefinitely.

There is an arbitrary number of client nodes and a fixed number n of server nodes, where clients and servers are disjoint. Clients can be authorized or unauthorized. Authorized clients are trusted (i.e., they are not Byzantine). Authorization is used to limit the access scope of a given client to the server's functionality and data. Correct servers are responsible for enforcing access control based on the access credentials of clients.

We use cryptographic techniques in the communication layer for authentication and digest calculation. We assume that adversaries (and Byzantine nodes under their control) are computationally bound so that they are unable, with very high probability, to subvert the cryptographic techniques used.

Our system implements a BFT key-value store service using state machine replication [12]. There are four operations to manipulate entries in the store: $read(key)$, $write(key, value)$, $add(key, value)$, and $cmp(key, value)$. $Read$ returns the share that a server possesses for a given key. $Write$ sets the share for the given key. Add tells the server to add the given value to the value it possesses for the given key. Cmp performs value comparison. Both key and value are of arbitrary type and length. Our key-value store service requires total ordering of messages, and the communication protocol is an extension of the one proposed in [13].

We assume that at most f servers can be Byzantine, and therefore require $n = 3f + 1$ servers of which at least $2f + 2$ are required for BFT execution and storage (see Section III-B).

B. Secret sharing

In our model a Byzantine server cannot compromise the confidentiality of the stored data for the simple reason that no server possesses a complete in-the-clear version of the data. This is achieved by employing a secret sharing scheme for all the data stored in the servers. There are several secret sharing algorithms in the literature [14]. In this paper, we consider Shamir's secret sharing algorithm (SSS). In a nutshell, the algorithm is based on the idea that polynomial coefficients and points on a polynomial curve are interchangeable. Therefore, to share a secret S in a number n of shares, where at least t shares are required to reassemble the secret, we sample n points (x, y) from a polynomial P of degree $t - 1$ whose constant coefficient is the secret and the other coefficients are taken randomly. To reassemble the secret we need to retrieve at least t shares to interpolate the polynomial P at the origin. To ensure that at least one share comes from a correct server, we must have $t > f$.

This algorithm has been proved *information-theoretically secure*, since its security relies only on the mathematical properties of polynomial interpolation, and therefore is secure against adversaries with unbounded computing power. Moreover, it has been proved to be additively homomorphic [15]: any manipulation performed on the shares by means of sums or subtractions will be propagated to the secret once it has been reassembled.

C. Verifying shares

Although secret sharing schemes have been proven safe from attackers with access to unbounded computing power, they are still vulnerable to a different class of attacks, in which participants can disrupt the reconstruction of the secret by providing corrupt shares (i.e., "cheating"). It becomes then necessary to detect shares from cheaters and remove them from the reassembly procedure. A secret sharing scheme that can detect cheaters is called a *verifiable secret sharing scheme*.

The majority of methods for cheater detection either relies on sharing algorithms that are not homomorphic (e.g., [16], [17], [18], [19], [20]), or on the introduction of additional validation information (such as signatures or checksums) in the secret, which removes the homomorphic property of the sharing algorithm.

The verification of shares can also be done by using quorums [21]. In this paper, we use this approach since it maintains the additive homomorphism and has a very small overhead. The details of quorum usage in Belisarius are presented in Section III-B.

III. System design

A. Architecture overview

The three main components of Belisarius are the client-side confidentiality handler, the BFT communication protocol, and the server-side transparent manipulation of obfuscated data (see Figure 1). The client-side confidentiality handler is responsible for submitting operations to the servers and applying the secret sharing scheme to the confidential data. The communication protocol offers Byzantine fault tolerant total ordering of requests to application clients and servers. The server-side data manipulation component is responsible for performing operations on the obfuscated data that can make use of the additive homomorphism of the secret sharing scheme.

B. The client side

To execute an operation against the servers, a client decomposes the operation parameters (i.e., the confidential data) into shares using SSS. The operations $write(key, value)$,

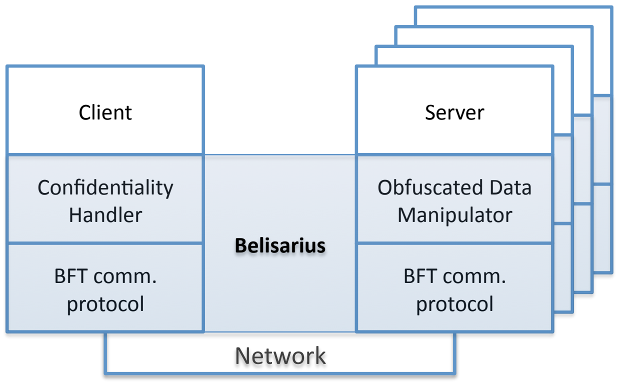


Fig. 1. Overview of Belisarius.

$add(key, value)$ and $cmp(key, value)$ are converted respectively into individual $write(key, share)$, $add(key, share)$ and $cmp(key, share)$ operations for each server. Once the shares have been calculated, they are broadcast using the total order protocol described in the next section.

In Belisarius, clients encrypt the shares using server-specific session keys and assemble a single message to be broadcast. This approach has the advantage of batching all the server shares into one single message that is ip-multicast by the total order broadcast layer. Session keys can be negotiated using any session key negotiation protocol (e.g., the SSL/TLS handshake protocol). If the underlying communication links provide end-to-end confidentiality (i.e., no eavesdropping), clients can send server shares by means of point-to-point communication, bypassing the total order broadcast protocol. Clients must still broadcast the operation identifier (or a digest of the operation) for total order. The advantage of such an approach is that it offloads the broadcast primitive.

The client then waits for the result of the operation from the servers. In traditional BFT systems, the minimal quorum required for completion is $2f+1$ replicas [11]. In other words, if each server possesses a full in-the-clear copy of the data, clients need only to perform a simple comparison between replies and return the one that has at least a $f+1$ majority.

For *quorum-based secret share verification*, however, $2f+1$ replies are not enough. This can be easily illustrated for $f = 1$ with a quorum of $2f + 1 = 3$ servers (see Figure 2). Clients need at least two replies to be able to reconstruct the original data. There are, therefore, three possible combinations of shares and each server's share participates in two combinations (i.e., share$_1$ with share$_2$, share$_2$ with share$_3$ and share$_1$ with share$_3$ – see Figure 2(a)). The share coming from the Byzantine server can corrupt two out of three combinations, making it impossible to isolate the correct combination using a majority test (see Figure 2(b)).

Since in Belisarius we strive not to modify the data by mixing or appending validation information (see Section III-D), each client must perform *secret share validation* using a large enough quorum by combining shares until a majority of identical combinations is found [21].

By introducing another correct server in the $f = 1$ example, thus increasing the total number of servers to $2f + 2$, clients will receive three correct shares and one corrupt. The three possible combinations of two shares out of the three correct shares will have the same value, while all the combinations involving the corrupt share will be different (see Figure 2(c)).

We capture secret share validation in Belisarius by two observations:

Observation 1. *For a system with at most f Byzantine servers, clients need at least $2f+2$ servers to perform secret share validation without augmenting the shares with validation information.*[1]

Observation 2. *In a system that has at most f Byzantine servers out of $2f + 2$, the correct combination will have a cardinality of $f + 2$ and corrupt combinations will have a cardinality of at most f.*

Observation 1 allows clients to distinguish correct shares, as explained above. Observation 2 has a very pragmatic application: clients do not need to wait until they gather $f+2$ identical combinations, since a cardinality of $f + 1$ can only be achieved by the correct combination.

In the general case, a client waits for a set R with $f + 2$ server replies and starts computing the secret for enough subsets of R of size $f+1$ until it assembles a set C of $f+1$ combinations with the same value. If a Byzantine share is present, then the client may have to wait for up to $2f + 2$ replies.

C. Communication protocol

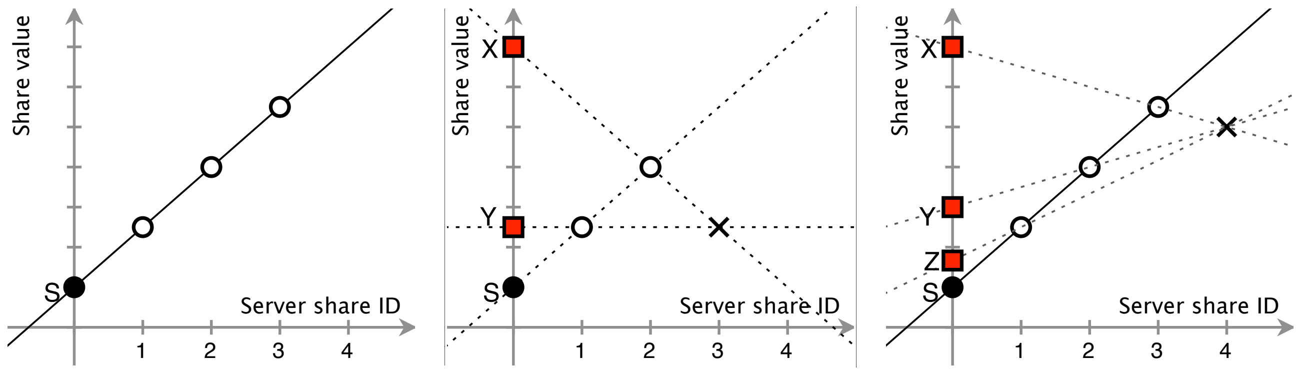
Our total order broadcast protocol builds on PBFT [13]. Like PBFT, for each client *REQUEST* message there is one *PRE-PREPARE* message from the primary replica, responsible for dictating the global order of requests, one *PREPARE* message from each backup replica, and a *COMMIT* message from each replica. The client receives at least $2f + 1$ *REPLY* messages from the servers. Unlike PBFT, clients always multicast their requests to the server group entirely, not only to the primary.

For services that do not employ secret sharing, the client has to collect at least $f + 1$ unique *REPLY* messages with equal content (e.g., at the byte level). As soon as the client has received $f + 1$ identical results from different senders, it can deliver the result to the client application. In Belisarius, additional steps are needed, as detailed in Section III-B.

D. The server side

Confidentiality of the data stored on servers is guaranteed by the fact that no single server contains any usable data

[1]Although [21] defines 3 types of attacks on share verification, in Belisarius only types 1 (single server failure) and 2 (Byzantine servers collusion) apply, since servers distrust each other and therefore do not have access to each others' shares.



(a) with $2f + 1$ correct shares and $f = 1$ (b) with one corrupt share (c) with $2f + 2$ shares, including one corrupted

Fig. 2. Share verification in Belisarius (a) with $2f + 1$ correct shares and $f = 1$; (b) with one corrupt share; (c) with $2f + 2$ shares, including one corrupted. S, the black-filled dot, is the correct combination (secret), X, Y and Z are combinations including corrupt shares. White-filled dots indicate the share values for different servers. Squares indicate combinations of shares including one corrupt share.

by itself. In fact, due to the properties of the secret sharing algorithm, we setup the system so that there will always be a correct replica participating in any execution quorum. This will prevent information leakage by making it impossible for Byzantine clients to reconstruct leaked data by requesting it only to the Byzantine servers.

Obfuscated data on the servers cannot be as easily manipulated as plain data. For example, in the case of systems that obfuscate their data using traditional encryption schemes (i.e., non-homomorphic encryption schemes) arbitrary changes to the obfuscated data cause the loss of the original data. This has the unfortunate side-effect of preventing any kind of server-side data manipulation. We enable additive operations to be performed on the server-side. As a proof of concept, we exploited additive homomorphism in the TPC-B benchmark (see Section V). Multiplicative server-side operations could also be implemented [22].

One difficulty with data obfuscation through secret sharing is that it makes state transfer between servers more complicated. As in PBFT, servers in Belisarius can checkpoint their state. In case of recovery, however, a Belisarius server cannot simply ask the state of another server. A simple solution is for authorized clients to intermediate the recovery process. More complex techniques could also be used [23]. A full description of Belisarius's recovery protocol lies outside the scope of this paper.

IV. Implementation

We implemented a prototype of Belisarius in Java using the client/server socket framework Netty [24]. Our prototype implements all the features described in Section III. The prototype does not yet implement the *checkpoint* and *view change* protocols of PBFT [13].

We have organized the nodes in two multicast groups. Both contain all replicas, but one is used by clients to send requests, and the other is used only for inter-replica communication of protocol-related control messages. Replies are sent directly back to the client using UDP.

Our BFT total order broadcast layer behaves as an application-agnostic communication library. We also implemented Shamir's secret sharing scheme ourselves, since the available libraries either did not support arbitrary-length payloads or were not implemented in Java. Our implementation uses the Horner scheme to evaluate polynomials, and an optimized version of Lagrange interpolation where the basis polynomials $\ell_j(x)$ are precomputed for $x = 0$.

For the purposes of performance comparison, we also implemented the "Privacy Firewall" proposed in [11]. We build it on top of the total order broadcast communication library provided by Belisarius and using ThreshSig [25] to provide the RSA threshold signature algorithm [26].

V. Performance Evaluation

We evaluated the performance of Belisarius with four benchmarks:

- Belisarius *no-op* throughput and latency: we measured the throughput and latency of "null operations" for different sizes of payload and number of clients to stress test our BFT total order broadcast communication protocol.
- Privacy Firewall comparison: we compared the throughput and latency of Belisarius's confidentiality stack to the Privacy Firewall.
- TPC-B: we implemented two versions of a TPC-B-like benchmark, comparing client-side and server-side execution.
- NFS: we compared the performance of NFS version 2 when implemented on top of Belisarius to a native Linux implementation.

We ran all the tests on a cluster of Dell SC1435 servers equipped with two dual-core AMD Opteron processors running at 2.0 GHz and 4 GB of main memory. The servers are interconnected through an HP ProCurve 2900-48G gigabit network switch. The servers ran Ubuntu Linux 10.04 LTS 64-bit with kernel 2.6.32-21. We used the OpenJDK Runtime version 1.6.0_18 with the 64-Bit Server VM (build 16.0-b13).

A. No-op *throughput and latency*

We implemented a *no-op server* (i.e., the server simply returns the data sent by clients, without secret sharing) on top of Belisarius to measure throughput and latency under different combinations of payload sizes and numbers of clients. The number of clients started with one, and doubled until 512. The payload size started at 4 bytes, and doubled until 8 kB.

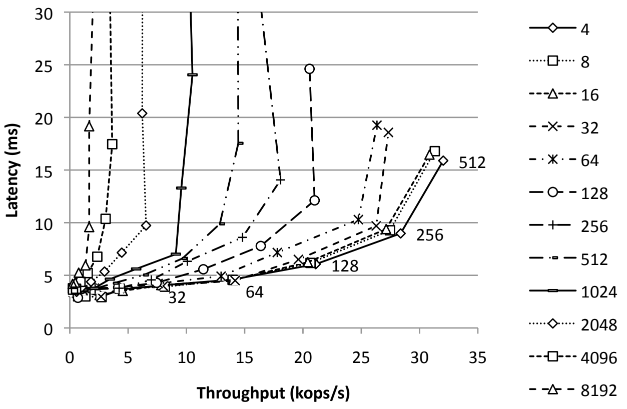


Fig. 3. Belisarius's total order broadcast library for payload sizes from 4 to 8192 bytes.

A summary of throughput vs. latency for different payload sizes is presented in Figure 3. Each curve represents a payload size and each point in a curve represents a different number of clients. The maximum throughput is around 32 kops/s for a latency of 16 ms. For payloads between 4 and 64 bytes, the break-point is around 256 clients for a latency between 9 and 10 ms. For payloads up to 64 bytes, the throughput scales almost linearly with the number of clients until it reaches 256 clients. After this number of clients the throughput saturates at around 32 kops/s for payloads between 4 and 16 bytes. The latency remains under 5 ms for up to 64 clients with 64-bytes payloads, and under 10 ms for up to 256 clients with 64-bytes payloads. Even for 8 kB payloads, the latency remains under 10 ms for up to 16 clients.

For payloads larger than 64 bytes, we have a much sharper increase in latency. This is explained by network traffic overload: in these configurations, the number of clients multiplied by the size of the payload generates enough traffic to overload incoming network links, causing packet loss at the receivers, a fact that was confirmed by operating system counters.

We also compared the throughput of Belisarius's approach to confidentiality (i.e., secret sharing) to the traditional approach (i.e., encryption). We extended clients and servers to support both confidentiality approaches, using AES with 128-bit keys, for $f = 1$, and with the same number of replicas. The performance difference is minimal for most cases, ranging from about the same performance for up to two clients and messages of up to 128 bytes and stabilizes at about half the throughput for about twice the latency for larger messages and number of clients. The throughput difference can be explained by the fact that servers in our implementation receive n times as much information as they need, due to secret sharing via multicast. The response time difference is explained by the fact that clients need to wait for $f + 2$ replies in the secret sharing case instead of $f + 1$ replies in the encryption case.

B. *Privacy Firewall comparison*

We compared Belisarius to the Privacy Firewall system (PFW) proposed in [11], the only Byzantine fault-tolerant system that provides confidentiality and allows full server-side functionality. A more detailed qualitative analysis of PFW is presented in Section VI.

We implemented a prototype of PFW as closely as possible to the specification presented in the original paper. Since in Belisarius we use AES with 128-bit keys for transport authentication and encryption, we first tried to use an equivalently secure RSA key size for PFW. Since the key size is not reported in [11], we followed NIST's recommendation of a 3072-bit RSA key [27]. Using that key length, however, resulted in latencies of at least 1500 ms, for a single client. These latencies are much larger than the ones published in [11], and thus we experimented with a shorter, 128 bit RSA key.

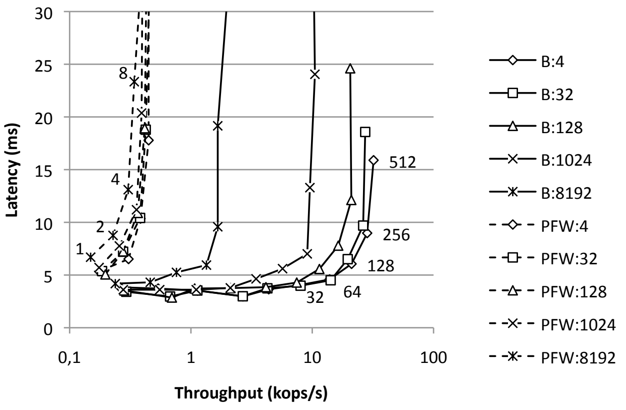


Fig. 4. Belisarius vs. Privacy Firewall. Each curve represents a different payload (in bytes)

We have found that even using 128-bit keys, PFW reaches its saturation point with relatively few clients (less than 5, see "PFW" curves on Figure 4). Further investigation indicated that the threshold signing procedure is very CPU-intensive, and quickly becomes the predominant activity inside the JVM. The maximum throughput is 480 ops/s for a latency of 65 ms. The breakpoint seems to be around 4 clients for a latency between 10 and 12 ms. In comparison, although the payload size does have an impact on the throughput of Belisarius, even at its worst saturation point, it is still one order of magnitude faster than PFW.

We conclude that the advantage presented by PFW over Belisarius (e.g., each server contains a full, in-the-clear copy of the data, allowing server-side data manipulation while preserving confidentiality) presents a serious performance compromise. In our tests, a Belisarius-based system could execute at least 10 times as many operations as an equivalent PFW-based system in the same time. In other words, the performance impact caused by PFW is acceptable if the services provided involve a large number of operations or if they involve a large amount of data, such as stored procedures on database servers. Even in such a case, though, the execution of such stored procedures would have some important restrictions (see Section VI).

C. TPC-B

We compared Belisarius's homomorphic capabilities against a system without such functionality. In practice, we executed a test using the same operations as the TPC-B benchmark, but in one case performing additive operations on the server side, and on the other case first retrieving the data to the client, performing the additive operations and then sending the results back to the servers. In the TPC-B-like benchmark, this means that each transaction (16 bytes) is broken into two parts: the first one reads data (16 bytes request and 12 bytes reply), and the second one writes the results (28 bytes request).

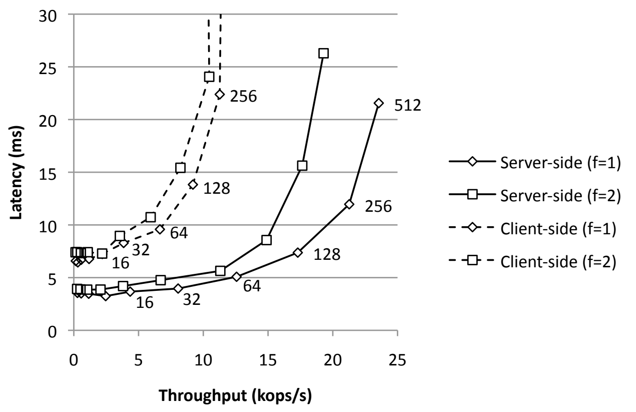


Fig. 5. Client- vs. server-side computation for TPC-B-like benchmark.

The usage of an homomorphic secret sharing scheme shows a clear performance advantage (see Figure 5). Without server-side processing, our TPC-B-like benchmark peaked at 11.5 kops/s, with a saturation point around 128 clients and a latency of 13 ms. By employing server-side processing (through additive homomorphism), the peak throughput is beyond 24 kops/s, and the saturation point lies around 256 clients for a latency around 12 ms.

We also tested throughput and latency for $f = 2$ (i.e., a total of 7 replicas). Although there is a performance impact, the prevalent use of multicast mitigates most of the cost of the additional replicas. For client-side computations, the throughput loss is on average under 10%, and a latency increase is on average around 10% (see "$f = 2$" curves on Figure 5). For server-side computations, the worst case scenario is an 18% throughput loss (average 13%) for a 22% latency increase (average 16%).

For equivalent numbers of clients, we observe that: (1) as expected, the latency for client-side transactions is about twice the latency of server-side transactions, and (2) executing transactions on the server-side doubles the throughput. Notice that performing client-side operations with Belisarius is more than an order of magnitude more efficient than PFW.

D. NFS

Finally, we used Belisarius to implement an NFS version 2 server with and without confidentiality, and compare it to the NFS version 2 implementation that is provided with Linux (referred to as "native") on our cluster nodes. We benchmarked the systems with *IOzone* [28]. Since our implementation of the NFS server did not offer any application-level caching, buffering or client authentication, we configured native servers in the same way. To minimize the influence of hardware bottlenecks, we also disabled synchronous writes for all tests.

To establish a baseline, we compared both read and write performances of the native NFS server against our Java implementation. As expected, the native implementation performed significantly better than our Java implementation. Our implementation reached up to 55% of the native speed for writes and up to 40% of the native speed for reads. We believe this is due to the fact that our Java implementation did not have the same level of optimization as the native version (e.g., zero-copy file reads, JVM overhead).

It is interesting to observe that for some cases discussed next, with or without confidentiality, we obtained significant performance gains over the single native server. We attribute these performance gains to two factors: (1) IOzone is a single-thread client, and therefore lower latencies result in higher throughput, and (2) in average, a single replica is slower than the fastest three out of four.

The read performance of Belisarius NFS with confidentiality peaks at 1.5 times the single native server for small block sizes, but for larger block sizes (larger than 8 kB) the native server overtakes our replicated service. This reversal is a consequence of the performance gap between our Java implementation and the native server (see Figure 6(a)).

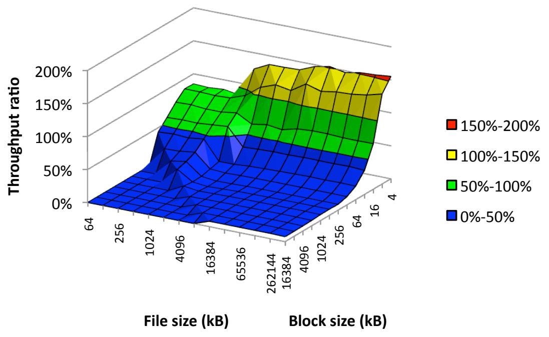
The write performance peaks at 3.4 times the single native server for small files, and even for large files our replicated server keeps quite close to the native server, trailing at 90% of the throughput (see Figure 6(b)).

We have also measured latency but for lack of space do not report it in the paper. Belisarius NFS has lower latency than the native implementation for write operations in all scenarios, but higher latency for read operations.

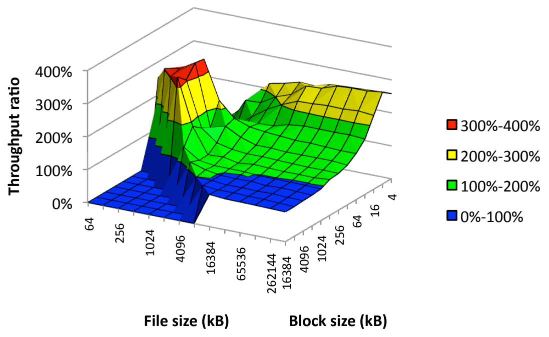
VI. Related Work

Belisarius stands in the intersection of Byzantine fault tolerance, secure storage and verifiable secret sharing. Its goal is to offer Byzantine fault tolerance with confidentiality protection. BFT systems with confidentiality protection can be based on *obfuscation* or *firewalling*.

Within the *obfuscation* class of BFT systems, servers can tolerate data leakage, since all the data they host is unusable due to the obfuscation mechanism. The obfuscation mechanism can be a cryptographic scheme, a secret sharing scheme,



(a) Read



(b) Write

Fig. 6. NFS with confidentiality over native server for read and write operations.

or a combination of both [7], [8], [9], [29], [30], [31], [33]. Obfuscating systems are only as strong as their obfuscation scheme. Systems that rely simply on cryptography, for example, have to worry about key length, key revocation and algorithm strength. Systems that rely on secret sharing have to deal with share renewal and share validation. Data obfuscation through encryption has been throughly investigated in [32].

The majority of traditional obfuscation schemes does not have homomorphic properties. For instance, traditional symmetric and asymmetric encryption, as well as secret sharing schemes based on XOR decomposition [30] do not offer any kind of homomorphism. There is an active research community around the development of homomorphic encryption algorithms, but so far the state-of-the-art in that area has severe performance issues [34].

The usage of secret sharing for obfuscation of data has an extensive literature. However, due to the traditional assumption of poor performance for secret sharing algorithms, most implementations limited themselves to sharing the keys for the encryption subsystem [10]. In [33], the authors employ a hybrid obfuscation model, which trades homomorphic properties for reduced storage space by applying symmetric encryption of the data, secret sharing of the encryption key, and erasure codes to distribute the encrypted data.

The verification of the obfuscated data is an area of intensive research [20], [35], [36], [37], and is usually done by introducing some token with the data (e.g., a checksum, a digest, a previously agreed piece of information). By introducing such a token, any homomorphic property is taken away, since token generators are usually not homomorphic.

Belisarius's verifiable secret sharing scheme builds on the algorithm proposed in [21], which does not rely on an additional token and thus preserves the homomorphic properties of the obfuscation scheme. Belisarius shows how the algorithm in [21] can be turned into a high performance system that tolerates Byzantine failures and how applications can make efficiently use of it.

Within the *firewalling* class of BFT systems, data leakage is prevented by a careful isolation of the servers from the clients by a separation layer. This layer must guarantee that correct requests and results are properly forwarded through, but Byzantine replies are eliminated before they reach the clients. The greatest advantage of firewalling systems is the support for arbitrary commands. However, this has also been the reason why firewalling systems were considered unfeasible for general-purpose systems [13]. Recent developments have changed this view. Yin et al. [11] have presented the only general-purpose system based on firewalling that we are aware of. That system is based on a *Privacy Firewall* which separates the ordering of requests from their execution. These two layers are separated by several layers of filters, which are responsible for eliminating any Byzantine replies. Byzantine replies are detected using threshold signatures [26], which are fast to reconstruct and verify, but are computationally expensive to create. Besides the performance cost of threshold signatures and increased latency caused by the filter layers, there are two significant downsides to Privacy Firewall, as we explain next.

First, it can be applied to any system as long as the offered services behave deterministically, that is, if each request yields exactly the same bit-by-bit reply from each replica. If replies from different replicas differ even for one bit, then the threshold signature verification algorithm will fail at the first filter layer and all the replies will be discarded. For example, any service dealing with server-generated timestamps will require either perfect clock synchronization between replicas or compromise for less accurate timestamps. Since the first layer of filters prevents any reply from passing through unless it matches the shared signature (i.e., matches the majority of replies), clients are not allowed to see non-perfectly matching answers, even if it would be possible for clients to implement their own logic to handle such cases (e.g., averaging the timestamps).

Second, Privacy Firewall introduces two new sources of Byzantine nodes. Whereas in traditional BFT systems each node was responsible for both order and execution and thus only one type of Byzantine node existed, in the Privacy Firewall system there may be f Byzantine agreement nodes, g Byzantine filter nodes, and h Byzantine execution nodes.

This causes a heavy investment in infrastructure, even if some of these roles can be combined in the same physical node. For example, to tolerate a single fault of each type, the system must have at least 11 processes running in 9 nodes. Agreement nodes require a $3f+1$ quorum, filter nodes require a $(f+1)^2$ quorum, and execution nodes require a $2f+1$ quorum. Finally, the execution nodes must be on a separate network partition from the agreement nodes, and they must interface only through the filter nodes, which must themselves also be partitioned in $f+1$ layers.

VII. Conclusion

In this paper we have presented Belisarius, a lightweight BFT storage system with confidentiality. Although a few systems in the literature offer Byzantine fault-tolerance with confidentiality through encryption or non-homomorphic secret sharing schemes, Belisarius distinguishes itself by applying a secret sharing scheme and retaining its additive homomorphism, which enables the implementation of multi-party computation applications on top of a BFT total order broadcast. Our performance measurements indicate that Belisarius compares favorably to other BFT systems with confidentiality. We also observed that it compares favorably to simple BFT storage engines with confidentiality.

References

[1] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Transactions on Computer Systems*, vol. 27, no. 4, December 2009.

[2] B. Liskov and R. Rodrigues, "Byzantine clients rendered harmless," in *DISC '05*, ser. LNCS, vol. 3724, September 2005, pp. 311–325.

[3] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, July 1982.

[4] A. Brodsky, C. Farkas, and S. Jajodia, "Secure databases: Constraints, inference channels, and monitoring disclosures," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 6, pp. 900–919, November 2000.

[5] R. C. Newman, "Covert computer and network communications," in *InfoSecCD '07*, September 2007, pp. 1–8.

[6] C. Cachin, I. Keidar, and A. Shraer, "Trusting the cloud," *SIGACT News*, vol. 40, no. 2, pp. 81–86, June 2009.

[7] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," in *OSDI '02*, December 2002, pp. 1–14.

[8] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: an architecture for global-scale persistent storage," in *ASPLOS 2000*, November 2000, pp. 190–201.

[9] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti, "Potshards: Secure long-term storage without encryption," in *USENIX '07*, June 2007, pp. 143–156.

[10] M. K. Reiter, M. Franklin, J. Lacy, and R. Wright, "The Ω key management service," in *CCS '96*, March 1996, pp. 38–47.

[11] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," in *SOSP '03*, October 2003, pp. 253–267.

[12] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, December 1990.

[13] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI '99*, February 1999, pp. 173–186.

[14] D. R. Stinson, "An explication of secret sharing schemes," *Designs, Codes and Cryptography*, vol. 2, no. 4, pp. 357–390, December 1992.

[15] J. C. Benaloh, "Secret sharing homomorphisms: Keeping shares of a secret secret (extended abstract)," in *CRYPTO '86*, ser. LNCS, vol. 263, August 1986, pp. 251–260.

[16] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch, "Verifiable secret sharing and achieving simultaneity in the presence of faults," in *FOCS '85*, October 1985, pp. 383–395.

[17] E. Dawson and D. Donovan, "The breadth of shamir's secret-sharing scheme," *Computers & Security*, vol. 13, no. 1, pp. 69–78, February 1994.

[18] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," in *FOCS '87*, October 1987, pp. 427–438.

[19] T. Rabin and M. Ben-Or, "Verifiable secret sharing and multiparty protocols with honest majority," in *STOC '89*, May 1989, pp. 73–85.

[20] M. Tompa and H. Woll, "How to share a secret with cheaters," *Journal of Cryptology*, vol. 1, no. 3, pp. 133–138, October 1989.

[21] L. Harn and C. Lin, "Detection and identification of cheaters in (t, n) secret sharing scheme," *Designs, Codes and Cryptography*, vol. 52, no. 1, pp. 15–24, January 2009.

[22] R. Cramer, I. Damgård, and U. Maurer, "General secure multi-party computation from any linear secret-sharing scheme," in *EUROCRYPT 2000*, ser. LNCS, vol. 1807, May 2000, pp. 316–334.

[23] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft, "Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation," in *TCC '06*, ser. LNCS, vol. 3876, March 2006, pp. 285–304.

[24] T. H. Lee, "Netty - the java nio client server socket framework," http://jboss.org/netty.

[25] S. Weis and L. Kissner, "Threshsig: Java threshold signatures," http://threshsig.sf.net/.

[26] V. Shoup, "Practical threshold signatures," in *EUROCRYPT 2000*, ser. LNCS, vol. 1807, May 2000, pp. 207–220.

[27] E. Barker and A. Roginsky, "Draft sp 800-131: Recommendation for the transitioning of cryptographic algorithms and key sizes," NIST Special Publication, p. 13, January 2010.

[28] D. Capps, "Iozone version 3.327," http://www.iozone.org/.

[29] S. Lakshmanan, M. Ahamad, and H. Venkateswaran, "Responsive security for stored data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 9, pp. 818–828, September 2003.

[30] A. Subbiah, "A new approach for fault tolerant and secure distributed storage," in *DSN '06*, vol. Supplemental Proceedings, June 2006, pp. 157–159.

[31] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga, "Depspace: a byzantine fault-tolerant coordination service," in *EUROSYS '08*, April 2008, pp. 163–176.

[32] E. Damiani, S. D. C. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, "Balancing confidentiality and efficiency in untrusted relational dbmss," in *CCS '03*, October 2003, pp. 93–102.

[33] A. N. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "Depsky: Dependable and secure storage in a cloud-of-clouds," in *EUROSYS '11*, April 2011, pp. 31–46.

[34] C. Fontaine and F. Galand, "A survey of homomorphic encryption for nonspecialists," *EURASIP Journal on Information Security*, vol. 2007, October 2007.

[35] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, "Proactive secret sharing or: How to cope with perpetual leakage," in *CRYPTO '95*, ser. LNCS, vol. 963, August 1995, pp. 339–352.

[36] W. Ogata, K. Kurosawa, and D. R. Stinson, "Optimum secret sharing scheme secure against cheating," in *EUROCRYPT '96*, ser. LNCS, vol. 1070, May 1996, pp. 200–211.

[37] C.-C. Chang and C.-W. Chan, "Detecting dealer cheating in secret sharing systems," in *COMPSAC 2000*, October 2000, pp. 449–453.