# Tiered Fault Tolerance for Long-Term Integrity

Byung-Gon Chun, Petros Maniatis
*Intel Research Berkeley*

Scott Shenker, John Kubiatowicz
*University of California at Berkeley*

## Abstract

Fault-tolerant services typically make assumptions about the type and maximum number of faults that they can tolerate while providing their correctness guarantees; when such a *fault threshold* is violated, correctness is lost. We revisit the notion of fault thresholds in the context of long-term archival storage. We observe that fault thresholds are inevitably violated in long-term services, making traditional fault tolerance inapplicable to the long-term. In this work, we undertake a "reallocation of the fault-tolerance budget" of a long-term service. We split the service into service pieces, each of which can tolerate a different number of faults without failing (and without causing the whole service to fail): each piece can be either in a critical *trusted fault tier*, which must never fail, or an *untrusted fault tier*, which can fail massively and often, or other fault tiers in between. By carefully engineering the split of a long-term service into pieces that must obey distinct fault thresholds, we can prolong its inevitable demise. We demonstrate this approach with *Bonafide*, a long-term key-value store that, unlike all similar systems proposed in the literature, maintains integrity in the face of Byzantine faults *without requiring self-certified data*. We describe the notion of tiered fault tolerance, the design, implementation, and experimental evaluation of Bonafide, and argue that our approach is a practical yet significant improvement over the state of the art for long-term services.

## 1 Introduction

Current fault-tolerant replicated service designs are often unsuitable for long-term applications, such as archival storage for digital artifacts, which is gaining importance for business [42], regulatory [5, 6], and cultural [36] reasons. This unsuitability results from the typical fault assumptions on which the correctness of such systems is conditioned. For example, in typical Byzantine-fault tolerant (BFT) replicated systems [13], it is assumed that the number of faulty replicas is always less than some fixed threshold such as $1/3$ of the replica population.

In typical, "near-term" applications, such a uniform-threshold-based fault assumption can be reasonable and achievable. For example, one can argue that in a well-maintained population of diverse, high-assurance replica servers, by the time a third of the population is broken into or just grows faulty, the operators of faulty replicas can repair them. Thus, the repair reduces the number of faulty replicas, averts a threshold breach, and thereby keeps the system's fault assumption inviolate.

Unfortunately, this reasoning falls apart for applications and deployments with a long-term horizon, say many decades. Whereas a population of replica servers can be plausibly "well-maintained" enough for a few years, it is difficult to protect perfectly from momentary threshold breaches over the long haul. Even improbable correlated faults become probable given enough time [10]. Once that threshold is breached, for however brief a period, the system's fault assumption is violated, and correctness can no longer be guaranteed at any point in time thereafter (Section 2.1).

In this work, we focus on storage applications with a long-term horizon and design a replicated service model that suits them. We observe that the reliability of a system's components over long spans of time can vary dramatically. First, a complex but formally unverified software artifact might be likely to exhibit vulnerabilities; all that stands between it and a bug is a lapse in the judgment of a human programmer. However, a formally verified software artifact might take much longer to exhibit vulnerabilities: it will not exhibit bugs against which it was verified, but perhaps the assumptions under which its correctness was verified might cease to hold upon a radical technology change (think of the transition from uniprocessors to multiprocessors as such a change). Taking this rationale to its extreme, a trusted third party—a "component" in a distributed service, such as a root DNS server—might take even longer to fail: for example, even if all involved hardware and software components are operating as specified, the trusted component can fail if the organization operating it sells out to criminals. We argue that whereas this differentiation might be esoteric and moot for near-term services, it may be an unavoidable consideration for long-term applications.

This observation leads us to a *tiered fault framework* for such replicated applications (Section 2.2). This framework partitions system components into different classes; for instance, software and hardware used for write operations is in a different class from software and hardware used for read-only operations. The framework treats components of one class across all nodes separately from components from another class, assigning a separate *fault tier* to each class. Like more traditional models, the fault assumption within each tier is threshold-based, but the actual threshold differs from tier to tier. For instance, the fault assumption for the population of write-operation components may be a 1/3

**This paper apepars in USENIX FAST 2009.**

threshold as with typical BFT systems, whereas the fault threshold for the population of read-operation components may be higher. There is no magic in this formulation: each fault tier is itself subject to a fault threshold. However, this multi-tier approach enables us to structure a system so as to operate longer without violating its overall fault assumptions.

One could informally view this tiered fault framework as a "reallocation of the dependability budget" across the different hardware and software components and across time. This stronger fault framework implies different operational practices for each component class: high-trust components must be formally verified before deployment—which might imply that they be limited in functionality or that they run infrequently and briefly, and are mostly off-line to reduce their attack surface—whereas lower-trust components might be larger, buggier, and running continuously.

To make things concrete, we study a particular kind of long-term application: an authentic long-term key-value store. Such a facility can be useful, for instance, as a directory for finding sensitive data given a human-memorable name. One example is a directory for self-certifying names of stored files given a file's human-friendly name. Such a service can close the loop for previously proposed reliable long-term archival services such as Glacier [25], Oceanstore [31], Pergamum [49], and Preservation DataStores [20], which can withstand Byzantine attacks only as long as a client holds a self-certifying name for a data item. This leaves out of scope the task of *finding* those human-unfriendly names by a user in the future, not to mention that today's self-certifying names (e.g., the SHA-1 hash of a document) will not be certifying anything in the future if the technology behind them is defeated (this was recently demonstrated as inevitable for SHA-1 hashes [19]).

*Bonafide* is such a key-value store that provides its correctness guarantees (integrity and liveness) under a tiered fault model (Section 3). Bonafide partitions the operation of a key-value store into three classes of components bound by three tiers of threshold-based fault assumptions. The lowest, most error-prone tier of Bonafide is occupied by the *service process*, a mechanism for responding to the clients' read-only requests (e.g., looking up existing key-value bindings) and for buffering—but not executing—new key-value additions. The middle tier contains the *update process*, which performs in batch all buffered key-value additions, but runs periodically and only briefly. The highest tier contains a minimal trusted facility for a *moded, attested storage* module (MAS), which keeps the error-prone service process safe and protects the integrity of the update process.

Bonafide provides its guarantees as long as no component of the trusted top tier and fewer than a third of middle-tier components fail at the same time; any number of bottom-tier components can fail. In addition, like other systems such as Carbonite [15], Bonafide offers durability (that is, does not lose stored key-value bindings) as long as the system creates copies of data faster than they are lost.

Our prototype implementation provides a simple add/get interface and shows reasonable performance (Section 4). We note that most building blocks for Bonafide are borrowed from prior work, most notably from trusted primitives, authenticated data structures, proactive recovery, and BFT replicated state machines. It is the structuring of Bonafide as a service observing a tiered fault framework for long-term operation that we claim as novel.

We discuss the tradeoffs and extensions of Bonafide in Section 5, describe related work in Section 6, and conclude in Section 7.

## 2 Tiered Fault Tolerance

In this section, we demonstrate how a uniform-fault-threshold system model is not suitable for long-term applications, and we introduce a tiered fault framework examining its feasibility for long-term applications. Although we focus here on Byzantine faults, the approach applies to weaker kinds of faults as well.

### 2.1 Fault Assumption Violations

We give here an example of how a system built on Castro and Liskov's popular Practical Byzantine Fault Tolerance (PBFT) [13] protocol for replicating state machines breaks with even a transient violation of the fault threshold. In PBFT, an upper bound $f$ on the number $N$ of replicas allows the use of replica quorums (typically of size $2f+1$) to protect the safety and liveness of the system, only as long as $N > 3f$. Figure 1 illustrates a population of $N = 4$ replicas, of which $r_1$ and $r_2$ are faulty, in violation of PBFT's fault bound[1] ($f = 1$). Furthermore, non-faulty replicas $r_0$ and $r_3$ cannot temporarily communicate with each other, e.g., due to transient interference such as DoS from the faulty replicas. Client $a$ sends $req_a$ to the system. The two faulty replicas convince $r_0$ to commit and execute $req_a$ first, since the three of them form a quorum of $3 = 2f + 1$. Later client $b$ sends $req_b$ to the system. The two faulty replicas convince $r_3$ to commit and execute $req_b$ first, since $r_3$ never saw $req_a$. Henceforth, all results that the two non-faulty replicas send back to clients will be dependent on divergent views of the system's global history and state.

Because only $2(= f+1)$ matching replies are required to convince a client of a result, even if the fault assumption is again met because one faulty replica is repaired, the remaining one faulty replica will always be able to corroborate $r_0$'s view of the world to some

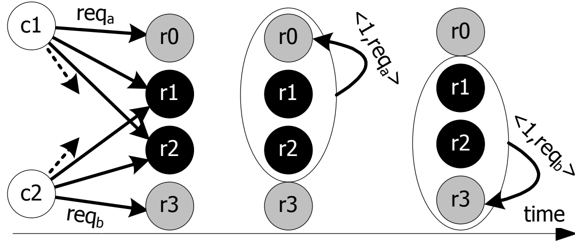


**Figure 1**: An example that shows the potential effects of a fault threshold violation in PBFT. Black circles are faulty replicas (one of them is the primary), gray circles are correct replicas, and white circles are clients. When two clients $c_1$ and $c_2$ submit requests $req_a$ and $req_b$ to the replicas at roughly the same time, but only manage to reach one correct replica each, the two faulty replicas can convince the two correct replicas to assign the same sequence number to different requests.

clients and $r_3$'s view of the world to other clients, keeping up the charade indefinitely. Crash-fault tolerant replicated state machines based on the Paxos [32] protocol do not deal with Byzantine faults explicitly (i.e., assume a Byzantine-fault threshold of 0) and they can have similar problems if a Byzantine fault crops up rarely and briefly.

Though not general for all possible replicated state machine protocols, this illustration serves to demonstrate the common trend: once the fault assumption is violated (the same as a threshold breach in traditional BFT protocols) the system cannot offer its correctness guarantees again, even if the fault assumption is later restored.

The fault bound in the original PBFT protocol applies over the lifetime of the system, assuming that once a replica becomes faulty it does not recover. PBFT's authors devised PBFT-PR, an enhanced protocol with some hardware support that attempts to repair faulty replicas. As a result, PBFT reduces the length of the *vulnerability window* of the system during which the fault bound might be breached; even though more than $f$ faults may occur during the lifetime of the system, as long as faults are repaired frequently enough so that no more than $f$ faults are ever simultaneously present, the system maintains its guarantees.

PBFT-PR achieves this repair using *proactive recovery* [14]: a hardware watchdog on every replica periodically reboots it with a fresh software installation from a read-only medium (e.g., a CD-ROM), flushing any runtime code damage caused since the last reboot. Upon reboot, the protocol cleans up the service state before it goes back into regular operation. Now the window of vulnerability[2] is the period of time between two successive, successful proactive recovery phases across the replica population, which is much shorter than the lifetime of the system. However, if the $f$ fault bound is violated within a vulnerability window, the protocol fails once again.

## 2.2 Tiered Fault Framework

We observe that the traditional fault model mostly presents an either-good-or-bad view of the world. Nodes that are faulty are incorrect and there is nothing in between. In reality, however, different components of nodes in a complex system exhibit different levels of fault tolerance; this fact is also explored in the wormholes model for short-term applications [51, 53] and we compare our approach with the wormholes model in detail in Section 6. In this work we argue that it may be unavoidable for long-term services to use a *tiered fault framework*, which exploits different levels of reliability in different components of the system. In this framework, different classes of components of the service implementation are assumed to keep their numbers of (Byzantine) faults under different thresholds.

We believe that a tiered fault framework is desirable because of the broad differentiation among software and hardware components. One source of differentiation comes from *different assurance practices*. Hardware microprocessor designs undergo extensive formal verification before production and, though extremely complex, tend to exhibit fewer bugs and security vulnerabilities in their implementation than typical software systems. Even in the software world, formally verified components can rigorously prove their correctness guarantees under specific execution models and, as a result, be protected from many runtime bugs and vulnerabilities [40, 45]. This approach can be leveraged with some success and performance cost via the use of strongly typed languages such as Java and C#, which are touted as safer environments for building robust systems: they offer a formal guarantee that, as long as the execution runtime implements the language semantics correctly, no application will be vulnerable to some of the typical system plagues like buffer overflows. Similar guarantees are offered by language-based type-safe operating systems such as Singularity [27].

A second source of differentiation comes from *care in the deployment of a system*: tight physical access controls, proactive hardware and software replacement, responsive system administration, well-designed firewalls and intrusion detection mechanisms contribute to keeping out the threats that can exploit any vulnerabilities present in the physical and logical interfaces of a system component. For example, a software component that is vulnerable to a particular exploit borne over SSH traffic can be shielded from that exploit if the firewalls between the Internet and that component drop all SSH packets before they reach it [54], or if it only communicates with other trusted components over a private network [18, 46].

A third source of differentiation comes from the *rolling procurement* characteristics of the software and

hardware technologies in long-term services. Unlike typical "near-term" systems, it is not the data that "flow" through the hardware and software, but instead the hardware and software that "flow" through the long-term service data[3]: though the service needs to remain the same, hardware becomes obsolete, operating systems evolve, communication standards grow, and cryptographic best-known methods are broken and replaced by their successors. For example, a trusted logical component assumed to never fail would require the expensive proactive replacement of the cryptographic libraries or the trusted hardware platform used to implement it, as new crypt-analysis techniques become possible, faster hardware is introduced, and new processes for protecting processor packages from physical or electrical tampering become available. In contrast, a less trusted component could afford to trail the state of the art and use replication or other techniques to mask faults, only migrating to new software and hardware less frequently and potentially at a lower cost.

Finally, fault differentiation can come from *limited exposure*. Many high-assurance systems such as certification authorities keep their sensitive components (e.g., their signing keys) mostly or wholly off-line, limiting attack opportunities. Services that have limited or potentially batched updates but can be mostly read-only (or indeed off-line with on-line, untrusted proxies [21,30,33]), can be protected quite effectively in this fashion.

Interestingly, there are non-trivial dependencies among all these sources of differentiation. For example, a proven-correct system that is operated by a trustworthy organization is strictly more reliable than the same system operated by an unreliable organization. As goes the usual secure systems' truism, a complex system is as secure as its weakest link. This simple observation allows us to argue that long-term fault models can usefully and realistically be constructed in which different system components belong in different *fault tiers*: in each tier, a different fault threshold can be assumed, though the justification for that fault threshold might imply restrictions on the component capabilities for each tier. For example, if one were to argue that a component tier is afforded a low fault threshold thanks to its being formally verified, that component cannot be too complex: formal verification is still an extremely expensive proposition both in terms of human effort and computational resources [56]. Similarly, a tier whose fault threshold is justified by its remaining mostly off-line had better correspond only to functionality that the service can afford to perform periodically in batches.

# 3 Bonafide

Our target application in this paper is Bonafide, a key-value store designed to provide long-term integrity using replication in a tiered fault model. We are motivated to build a long-term key-value store not only as a case study for the system-building approaches we described earlier in the paper, but also because it is fundamentally needed by archival storage systems such as Glacier [25], OceanStore [31], Pergamum [49], and Preservation DataStores [20]. Whereas such systems provide durability (protection from data loss) and authenticity, they require data to be *self-certified* for their authenticity properties to hold: a client who needs to fetch a document from the archival system must have an authenticator such as a cryptographic hash of the document's contents to ensure that what the service returns has not been modified; a client who does not have such an authenticator cannot obtain any authenticity guarantees from the service. We seek to create an archival service for providing indirection for precisely such authenticators: it can be used as a lookup service from a URL or a human-readable name to the random bits making up the authenticator, which can then be used to fetch the actual document from Glacier, Oceanstore, or systems similar to them.

In the simplest case, a system like Bonafide offers a minimal interface: clients invoke Bonafide's `Add`$(key, value)$ method to store and preserve a particular key-value pair, if no such key is already being preserved, and the `Get`$(key) \rightarrow value$ method to obtain any stored key-value pair by that key, if one exists. The service is append-only. There is no method to remove or replace an existing key-value pair. We use an append-only interface for simplicity; it is not a requirement.

## 3.1 Design Rationale

We apply the intuition behind the tiered fault framework by attempting to refactor the functionality of a service such as Bonafide into a more reliable fault tier for state changes and a less reliable fault tier for answering read-only state questions (i.e., `Get` requests). In keeping with the justification for distinct fault tiers, we make the reliable, state-changing functionality mostly off-line, running periodically to execute state changes in batch (for `Add` requests buffered but not executed during the mostly unreliable operation of the system).

One challenge with such a high-level design, especially when using commodity hardware, is the isolation between the reliable and the unreliable class of components. The Castro and Liskov approach punctuates a system's timeline with periodic software refreshes, which can help bring a system whose faults are climbing towards the fault threshold back from the precipice. Unfortunately, that isolation goes away once the system has crossed the precipice; even if the number of faults is somehow reduced below the fault threshold again, data

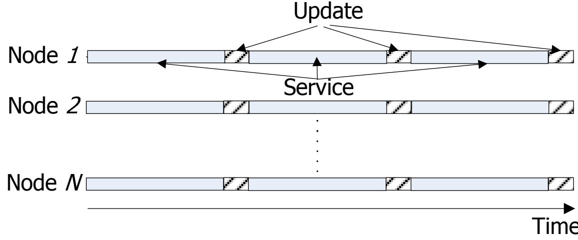


**Figure 2**: Operation of Bonafide. Each Bonafide replica alternates between a service phase (S phase) and an update phase (U phase).

changes before and after the fault violation cannot be isolated, causing the loss of safety and liveness guarantees.

To address this challenge, we require our third, most reliable class of component in its own top fault tier: a trusted mechanism for protecting state during execution of the unreliable class of components. This mechanism is not only extremely critical, but cannot even be mostly off-line; as a result, to ensure its fault threshold is plausible, we must make it extremely simple. For Bonafide's top-tier component class we use a *moded, attested storage* (MAS) facility, akin to the sealed storage mechanism provided by modern trusted platform hardware [4]. This facility allows us to store reliably very small amounts of memory, only allowing the reliable stage-changing mechanism to update this storage.

A second challenge is that our middle-tier, reliable state-changing mechanism must somehow be able to mask faults experienced by its components. We use a Byzantine-fault tolerant replicated state machine approach to implement this middle-tier mechanism. However, since this middle tier is mostly off, we require that all of the system's nodes execute this tier in a synchronized fashion, which implies loose clock synchronization and a fairly long period for the execution of this tier. We describe how to relax the requirement for clock synchronization and synchronized execution of the state machine replication in Section 5.2.

The final challenge is figuring out how to use our very limited attested storage to protect a potentially large service state. The approach we adopt is the use of an authenticated data structure, which allows the integrity of arbitrarily large, structured data to be protected by a small cryptographic digest.

## 3.2 Overview

Bonafide is a replicated service running on replicas $R = \{1, \ldots, N\}$ ($N = 3f + 1$). The replicas operate in alternating synchronous phases of two types: a *service phase* (S phase) and a subsequent *state-update phase* (U phase) (Figure 2). During the $i$-th S phase, `Get` requests can query the service state committed (i.e., fetch bindings

| Fault bound | Component | When | How used |
|---|---|---|---|
| 0 | Watchdog | Periodic | Invoked |
| | MAS | S phase | Read |
| | | U phase | Written/Read |
| $1/3$ Byzantine[4] | Update | U phase | Replicated store<br>Serve ADDs |
| Unbounded | Service | S phase | Serve GETs<br>Buffer ADDs<br>Audit and repair |

**Table 1**: The components in Bonafide and their associated fault tiers.

that were added) up to the $(i-1)$-st U phase. `Add` requests are buffered and executed after the end of the $i$-th S phase, during the $i$-th U phase. In other words, service state changes occur in batch *only* during the U phase.

Bonafide consists of three component classes (trusted storage, state update, and service), each of which belongs to a fault tier with a different fault threshold. Table 1 summarizes the fault tiers in Bonafide. The state update component of a replica contains the state update process, OS, and hardware excluding the trusted top tier, and the service component of a replica contains the service process, OS, and hardware excluding the trusted top tier.

In addition, Bonafide has the following standard, partial synchrony assumption for liveness. In the network, packet drops, reorderings, and duplications can occur but retransmissions of a message eventually deliver it. However, though finite upper bounds exist for message delivery and operation execution times, those bounds are not known to protocol entities. This is a standard network assumption for Byzantine-fault tolerant systems and is not unique to Bonafide.

Under this tiered fault assumption, Bonafide guarantees service safety, that is, *integrity of returned data*. However, to guarantee *durability* as well (i.e., that no data are lost) the system should create copies of data faster than they are lost, as in systems such as Carbonite [15]. Also, to ensure *liveness* (i.e., non-starvation) S phases with at most $2/3$ faulty replicas must occur once in a while (more precisely, within a finite number of phases at all points in time). This is to ensure that an `Add` request must be resubmitted by a client a finite number of times before it is eventually served by a U phase.

A Bonafide node contains a MAS as well as a buffer to hold `Add` requests temporarily and a main data structure that maintains committed bindings (Figure 3). In Bonafide, the service state—the key-value pairs—is maintained as a variation of a hash tree [37], which computes a cryptographic digest of the whole state from the leaves up, storing it at the tree root. The results of individual state queries (i.e., key lookups in the tree) can be validated against that root digest; as long as the digest is kept safe from tampering, individual lookups can be performed by an untrusted service component with-

out risking an integrity violation. This state is replicated at each replica in the system in untrusted storage (bottom tier) but its root digest (of size on the order of 1 Kbit in today's hardware) is stored in each node's MAS. Each replica's MAS module lies in its most trusted fault tier: we assume that while in service no MAS module returns contents other than those that were stored at it. We use a MAS for the root digest of the service state, since it cryptographically protects the integrity of any answers about that state provided by even an untrusted component.

The service state is updated during the U phase, which is invoked by a trusted watchdog in the most trusted fault tier. In the U phase, all buffered writes are agreed upon by non-faulty replicas using a state machine replication protocol and then reflected in the service state, replacing the integrity digest in each replica's MAS. The U phase is in the next most trusted fault tier in Bonafide: we assume that no more than a third of the replicas' update software can fail simultaneously, to ensure that the state machine replication protocol safety and liveness guarantees can be upheld within a single U phase.

The service state is served to clients during the S phase. Responses to `Get`/`Add` requests are accepted by clients when $f+1$ replicas return to the client the same result, and each result is consistent with the corresponding replica's service state digest in its MAS module. The $f+1$ number comes from the fault bound of the update tier, which assumes no more than $f$ update processes can be faulty in any single U phase; as a result, no more than $f$ update processes can put an incorrectly updated digest into their own MAS. If the same response to a client request is provided by at least $f+1$ untrusted service processes, but backed by the $f+1$ trusted state digests in the MAS, the client is guaranteed to be getting what at least one correct replica provides. At worst, the client will receive no valid responses or obviously invalid responses from the replicas and try again. Also, the service state is audited (for latent storage faults or other bit loss) and repaired during the S phase.

At the protocol level, Bonafide provides the following safety property. If an `Add` or `Get` operation collects $f+1$ matching replies from distinct replicas, the reply is correct. In other words, there is a serial schedule of committed bindings, and once a binding is committed, it is seen by clients. This is similar to the integrity guarantees of other long-term storage systems, but unlike them, Bonafide can take any key to "name" the sought data value, not only self-certified names. In this paper, we discuss only `Add`s, but the safety property of Bonafide holds even when there are `Remove`s or `Replace`s in the system API.

In addition to safety, Bonafide provides the following liveness property. If an `Add` operation collects $2f+1$ tentative acknowledgments from distinct replicas, the binding is guaranteed to be committed during the U phase if there are at most $f$ faults in the S phase during which the operation is invoked. If there are more than $f$ faults in the S phase, the `Add` operation does not guarantee the binding to be committed during the U phase since faulty replicas can send fake tentative acknowledgments.

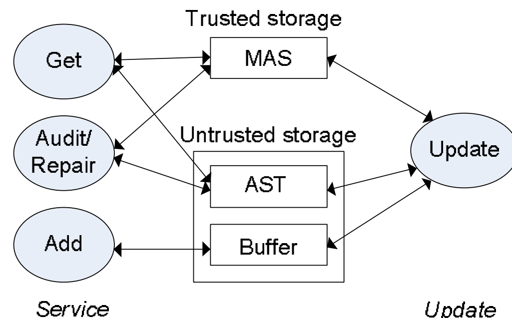


**Figure 3**: A Bonafide node contains the following state shown in the middle of the figure: a MAS, a buffer to hold `Add` requests temporarily, and an AST that maintains committed bindings. The MAS stores the AST root digest, a sequence number, and a checkpoint certificate. The left side shows the get, add, audit/repair processes running during the S phase, and the right side shows the update process running during the U phase. The arrows show what state the processes access.

A fundamental limitation of Bonafide is that it is not resistant to common denial-of-service attacks, such as *name squatting*. As in all existing archival systems we are aware of, faulty clients can insert arbitrary bindings into Bonafide, preventing legitimate clients from using those bindings.

Next we detail the service state and component functionalities of Bonafide.

## 3.3 Bonafide State

In Bonafide, the service state (the collection of key-value bindings) is maintained as an authenticated search tree (AST). An AST [11] is an incremental mechanism for maintaining cryptographic digests over sorted data sets (such as key-value pairs sorted by key), extending the concept of a traditional Merkle tree [37] for search. Every node contains a key-value pair and an authentication label. The label for an AST node is computed by hashing together its content and the labels of all child nodes. The label of the tree root is a cryptographic digest for the entire contents of the tree: it is *collision-resistant*, which means it is intractable to find two different data sets yielding the same AST digest and, as a result, it can serve as a *commitment* on the contents of the AST [39].

As with Merkle trees, a succinct *witness* (sometimes also called a *proof*) can be generated showing that a particular key-value pair appears within an AST with a root label. Unlike Merkle trees, an AST can also provide a succinct witness that a key *does not appear* within it.

Witnesses have logarithmic length in the number of the key-value pairs contained within a tree.

Each Bonafide replica maintains an AST in typical (untrusted) storage containing its collection of key-value pairs sorted by key, a buffer of received but yet uncommitted client requests for adding new key-value pairs also in untrusted storage, and a MAS having two slots within its trusted hardware device (see Section 3.4). The MAS slot with identifier $q_s$ stores values of the form $\langle s, r\rangle$, where $s$ is the latest AST digest and $r$ is an integer sequence number associated with a particular U phase. The slot with identifier $q_c$ stores a checkpoint certificate described in Section 3.6. Finally, replicas know each other's public keys and hardware device public keys.

## 3.4 Top Tier: Trusted

**Cryptography:** We assume standard cryptographic primitives for digital signing and hashing. These primitives belong in the top tier of the fault model (as they do in virtually all systems research and practice). In keeping with our "rolling procurement" argument for this trust over long periods (Section 2.2), we assume that cryptography is replaced with up-to-date technologies, algorithms, and key sizes well before its compromise is even feasible let alone practical (see Section 5.2 for a sketch of how this is accomplished). For a managed system, this is a reasonable and plausible approach.

For conciseness, we denote by $\langle M\rangle_i$ a message $M$ that is digitally signed by principal $i$. Replica $i$'s trusted hardware device (i.e., its MAS principal) is principal $i'$. If $M$ is not signed, there is no subscript in the notation.

**Trusted Hardware:** Bonafide relies on the existence of a trusted hardware device on every replica in the system [8]. Today, such a device could be implemented in a programmable, tamper-resistant secure coprocessor such as IBM's commercially available PCIXCC [8] board, but cheaper trusted alternatives are implementable with the trusted computing platforms coming from Intel's (e.g., Intel TXT) and AMD's (Presidio) pipelines.

To help conduct periodic recovery operations, this device contains a time source (this can be a regular, monotonic, crystal-based clock source with an upper bound on drift, or an external trusted time source received by the device such as GPS). A hardware watchdog, also contained within, uses this time source to trigger proactive recovery periodically, by causing the host to reboot from read-only media. This hardware watchdog sets a *mode* bit of the MAS associated with it. This bit is used to indicate that the system is in its U phase, and cannot be set in any fashion other than by triggering the watchdog. The mode bit can, however, be reset by the operating system. Typically this is done during the U phase, while the software is still under the middle trust tier. During the S phase, the no longer trusted operating system can of course reset this bit, but bit resets are idempotent, so this misbehavior is ineffective. Such mode bits are sometimes called *sticky registers*.

Finally, the device contains a MAS with a simple interface. A MAS contains a mode bit, and a set of storage slots, each of which is identified by an identifier $q$. The write interface to a MAS is $\texttt{Store}(q, v)$ where $v$ is a value; this stores $v$ to the slot with identifier $q$. This interface allows requests only when the mode bit indicates a U phase is ongoing. The read interface of MAS allows access all the time. It allows the attested, fresh retrieval of any slot; a $\texttt{Lookup}(q, z)$, where $q$ is a slot identifier and $z$ is a nonce used for freshness (typically provided by clients), returns $\langle \text{LOOKUP}, q, v, z, t, m\rangle_{i'}$, where $v$ is the value currently occupying the slot with identifier $q$ of the MAS, $t$ is the internal time in the device, $m$ is the current mode bit, and $i'$ is the hardware device principal. If the slot is empty, then $v = \text{EMPTY}$ in the returned attestation. In our own recent work, we have introduced an Attested Append-Only Memory (A2M) [16] and we compare MAS with A2M in detail in Section 6.

**Membership Management:** Bonafide is intended for a well provisioned, low-churn node infrastructure. Since membership churn is low, the membership of nodes can be managed manually. Membership management is conceptually also trusted, in the top tier of the fault model. We discuss how to extend Bonafide to automate membership management in the middle tier by refactoring it with MAS in Section 5.2.

## 3.5 Bottom Tier: Service Process

In the S phase, each Bonafide replica runs an add/get process to serve client requests, and a continuous audit and repair process in the background for durability. Pseudocode for the service process is given in Figure 4.

**Get:** When a client $c$ invokes $\texttt{Get}(k)$ to retrieve a value of key $k$, its Bonafide stub (called a *proxy* below) multicasts $\langle \text{GET}, k, z, c\rangle_c$ messages to $R$ where $z$ is a nonce used for freshness and waits for $f + 1$ $\langle \text{REPLY}, i, v, p_i, \langle \text{LOOKUP}, q_s, \langle s_i, r_i\rangle, z, t, m\rangle_{i'}\rangle$ valid matching messages confirming that $(k, v)$ is within the AST, or that $(k, v)$ does not exist in the AST. Note that the attestation includes the nonce the client sent to ensure it does not accept a stale response.

A replica handles a GET by looking it up by key in its local AST and producing an existence/non-existence witness, accompanied by its latest MAS attestation.

**Add Buffering:** When a client $c$ invokes $\texttt{Add}(k, v)$ to insert a binding between key $k$ and value $v$, the Bonafide client proxy code multicasts $\langle \text{ADD}, k, v, z, c\rangle_c$ to $R$. The client waits for $2f + 1$ *tentative* acknowledgments, each of which is a $\langle \text{TENTREPLY}, k, v, z, c\rangle_i$ message where $i$ is a replica identifier, from distinct replicas. If the client

```
CLIENT.GET(key)
  // quo_RPC sends msg to R, collects matching responses on non-*
  // fields from a quorum of given size, retransmits on timeout
  ⟨REPLY, *, value, witness, *⟩ ← quo_RPC(⟨GET, key⟩, f + 1)
  return value
SERVER.GET(client, key)_i   // this is server i
  ⟨value, witness⟩ ← lookup_AST(key)
  att ← lookup_MAS (q_s)   // attestation
  send client a ⟨REPLY, i, value, witness, att⟩

CLIENT.ADD(key, value)
  ⟨TENTREPLY, *, key, value⟩ ←
    quo_RPC(⟨ADD, key, value⟩, 2f + 1)
  // at this point, the client holds a tentative reply
  collect REPLY messages // in the next S phase
  if (f + 1 valid, matching replies are collected)
    return accepted(key, value)
SERVER.ADD(client, key, value)_i
  if (⟨key, value'⟩ in AST), treat as a GET and return
  add ⟨client, key, value⟩ to Adds
  send client a ⟨TENTREPLY, i, key, value⟩

SERVER.AUDIT(ASTNode, h_ASTNode)_i
  status ← check ASTNode, h_ASTNode
  if (status invalid) repair ASTNode   // fetch from other
  for each child C of ASTNode
      AUDIT(C, h_C) // h_C is contained in the label of ASTNode

SERVER.START_SERVICE(Committed_Adds)_i
  // reply for ADDs committed in the previous U phase
  for each ⟨key, value, client⟩ in Committed_Adds
      send client a ⟨REPLY, i, value, witness, att⟩
```

**Figure 4**: Simplified service process pseudocode.

```
SERVER.UPDATESTART()_i
  PBFT.Invoke(⟨BATCH, i, stable_ckpt_cert, Adds⟩_i)
  SERVER.FINALIZE()_i

SERVER.EXECUTE(batch)_i // PBFT Execute callback
  append batch in batch_log
  on receiving the 2f + 1-st batch:
    choose the latest stable_ckpt from batch_log
    AllAdds ← the union of the Adds set from each batch
    for each ⟨key, value, client⟩ in AllAdds
        repair the AST path to this new key if needed
        insert key, value into AST
        insert ⟨key, value, client⟩ into Committed_Adds
    store_MAS(q_s, ASTRootDigest)
    multicast a UCHECKPOINT and flush batch_log

SERVER.FINALIZE()_i
  on receiving 2f + 1 matching UCHECKPOINTs:
    store_MAS(q_c, stable_ckpt_cert)
    reset the watchdog timer and the mode bit
    begin a new S phase
```

**Figure 5**: Update process pseudocode.

does not receive the responses within a timeout, it presumes that the request has been dropped by the network, so it retransmits the request to the replicas that did not respond. Note that receiving $2f+1$ tentative acknowledgments is a hint that means the binding is likely to be committed. The client does not perform any operation that depends on the fact that the binding is committed and cannot be undone. Our liveness guarantee ensures that the client will receive a final commitment (see below) for each `Add` after receiving a finite number of uncommitted $2f+1$ such hints.

The client also waits asynchronously for *commit* replies in MAS-attested messages of the form $\langle \textsc{Reply}, i, v, p_i, \langle \textsc{Lookup}, q_s, \langle s_i, r_i \rangle, z, t, m \rangle_{i'} \rangle$ (the attestation is the result of a MAS `Lookup`). These messages are sent by replicas in the beginning of the next S phase. A reply is valid if witness $p_i$ verifies the existence of the key-value pair $(k, v)$ within an AST with digest $s_i$, and the attestation is correctly signed by the sender's MAS. As soon as the client proxy obtains $f+1$ valid matching replies from distinct replicas, all confirming the addition of the same key-value pair, it accepts the request as *complete* and notifies the application.

During the S phase, a replica only buffers ADDS and sends a TENTREPLY message back for each ADD. It handles the ADDS during the U phase. The replica also returns the existing mapping for ADDS for already assigned names. During the next S phase, the replica responds to newly inserted ADDS with a REPLY message. The average (committed) response time for ADDS of new bindings is on the order of the S phase length.

**Audit and Repair:** The audit and repair process ensures that all reachable AST nodes from the AST root are correctly stored. This process is recursive, starting with $\textsc{Server.Audit}(ASTRoot, h_{ASTRoot})$ where $h_{ASTRoot}$ is the digest of the AST root and traversing the tree in order, during which a tree node is fetched from storage if still available and verified by computing its hash value and comparing it with the hash contained in the label of its parent node.

For every missing AST node with digest $h$, replica $i$ multicasts a $\langle \textsc{ReqASTNode}, i, h \rangle_i$ request to $R$, waiting for at least one $\langle \textsc{RespASTNode}, h, ASTNode \rangle$ response. The response need not be signed, since the replica can verify its integrity thanks to the recursive hashes of the AST. As long as the root digest remains in the trusted MAS, the rest of the AST nodes are self-certifying.

## 3.6 Middle Tier: Update Process

When the trusted watchdog timer expires, the system begins a reboot securely from a read-only medium of its proactive recovery software. The main responsibility of the U phase is to commit a new set of additions into the main service state. At the end of the U phase, the system ensures that at least $2f+1$ replicas store the latest service state digest in MAS (see Figure 5 for the pseudocode).

We use the PBFT protocol [14] to replicate the state machines of individual U phases, though any BFT state machine replication protocol would work. PBFT offers a synchronous `Invoke`$(request)$ method, that returns a $response$. A PBFT client (which is a replica's U phase

in our use of the protocol) uses this method to submit application requests—buffered `Add` requests—to replicas and eventually receives replies containing the result. PBFT also offers replicas an `Execute`(*request*) callback, which invokes the application code that processes ordered client requests to be executed—the actual insertion of `Add`ed key bindings into the service state.

All messages exchanged between replicas contain a fresh attestation fetched from the MAS after the current U phase began: the mode bit shown must be on, and the timestamp must be recent. Messages unaccompanied by this attestation are invalid and dropped. This is to ensure that update operations, including invocations of PBFT, are performed by nodes that have rebooted into their U phase. Any faults caused by such nodes are due to update process faults, which our middle fault tier bounds by $f$.

**Update Start:** Each replica packages up its pending ADDS (denoted by $A$) and the latest stable checkpoint (i.e., $2f + 1$ matching MAS attestations of the previous round) (denoted by $C_s$) obtained from the MAS slot $q_c$ into a $\langle \text{BATCH}, i, C_s, A \rangle_i$ message, which it submits to PBFT's `Invoke`.

**Application State Machine:** The update state machine, executed on BATCH requests as ordered by PBFT, stores BATCH requests in order, until it receives the $2f + 1$-st of them (from distinct replicas). Subsequent batches are ignored to ensure liveness (there is no way for replicas to know how many batches they can expect beyond the $2f + 1$-st). Note that if there are at most $f$ faults during the S phase, there is at least one correct replica submitting every request whose client received $2f + 1$ tentative acknowledgments during the S phase, precluding starvation. As long as there are such S phases with no more than $f$ faults from time to time, the system makes progress.

In receiving the $(2f + 1)$-st batch, the application state machine picks the latest stable checkpoint, and the union of all ADDS across all $2f + 1$ batches. It orders the ADDS according to a consistent order (e.g., by $h(k\|v\|c)$), verifies the client's signature, and inserts all valid pairs into the AST in that order, ignoring keys that already exist. The replica computes the new AST digest $s_i^*$ for sequence number $r_i^*$ $(= r + 1)$, stores it into the $q_s$ slot of its MAS, and multicasts to $R$ a $\langle \text{UCHECKPOINT}, \langle \text{LOOKUP}, q_s, \langle s_i^*, r_i^* \rangle, t, m \rangle_{i'} \rangle$ message.

When a replica receives $2f + 1$ matching UCHECKPOINT messages, it stores the set at the MAS slot $q_c$ as a new stable checkpoint certificate. If a replica's old service state is not the latest one, it will have to perform state transfer, as described below.

When the replica obtains a new stable checkpoint certificate, it resets its watchdog timer to $\mathcal{D}$, which is the remaining time until the next U phase, and exits into its S phase by opening up communication with nodes other than replicas and resetting the mode variable. In the beginning of the new S phase, the replica sends REPLY messages for all newly inserted ADDS as described earlier.

**State Transfer:** Up-to-date replicas missing actual service state (e.g., because some of the AST nodes were corrupted) can apply the same repair process used during the S phase to obtain the AST nodes required for their batched ADDS.

Before the phase can end, the MAS of a replica must contain the latest stable checkpoint. A slow replica may be behind to obtain its checkpoint by executing the agreed-upon write operations. However, the stable checkpoint broadcast by those replicas that were up to date allows a slow replica to append that state digest into its MAS, thereby catching up with others and entering to the next S phase.

**Single-agreement Optimization:** The design described requires at least $2f+1$ PBFT invocations, one per BATCH, during every U phase. In the worst case, each invocation requires 3 network roundtrip times, potentially increasing the latency of the U phase tasks, which increases the minimum duration of the U phase, which in term reduces the availability of the system. Instead, the update process can do preprocessing to create a PROPOSE message containing at least $2f+1$ BATCH messages and only submit that proposal to PBFT. This optimization duplicates the functionality of the PBFT primary by introducing a leader to collect the BATCH messages. At the cost of greater complexity, this optimization can make use of all available BATCH messages, not just the first $2f + 1$ messages generated by replicas, and also reduce the worst-case number of roundtrip times required. Our implementation uses this optimization.

Each replica packages up its pending ADDS $A$ and the latest stable checkpoint $C_s$ obtained from the MAS slot $q_c$ into a $\langle \text{BATCH}, i, C_s, A \rangle_i$ message, filtering out those ADDS for already assigned keys, and broadcasts the BATCH to $R$. Once a *leader* replica (defined below) collects at least $2f + 1$ such messages including its own, it packages them into a PROPOSE message, which it submits to PBFT's `Invoke` for Byzantine agreement. During the `Execute` callback of PBFT, a replica ensures the PROPOSEd set contains at least $2f + 1$ batches from distinct replicas. If so, it runs the function to update the service state, and the rest of the U phase is the same.

During each U phase, the leader described above is the replica ($l \equiv r \mod N$), where $r$ is the current U phase round number. A leader may misbehave, either by delaying the transmission of a PROPOSE message, or by transmitting an incorrect such message. The latter case can be detected during the `Execute` PBFT callback, as described above. A non-faulty replica can detect the former case by setting a timer as soon as it multicasts its BATCH message, which it uneventfully stops when it encounters

its own BATCH as one of the batches included in a proposal during the `Execute` callback; if the timer expires, then the replica initiates a leader change. To avoid unnecessary leader changes due to transient slowness, the replica does exponential backoff for consecutive leader change initiations.

Leader change is similar to proposal formation: to initiate change, a replica multicasts a LEADERCHANGE message, which the *next* leader ($l + 1 \mod N$) listens for. When that leader has collected $2f + 1$ such requests, it packages them into a single LEADERCHANGEREQUEST, which it submits to PBFT; execution of this request increments $l$, completing the leader change. Note that the leader role is similar but unrelated to the PBFT primary role; PBFT's internal operation, including primary assignment and view changes, is opaque to the U phase functionality.

## 3.7 Correctness

Under the tiered fault assumption, Bonafide provides the integrity property. Briefly, it is sufficient to show that any binding committed is guaranteed to be safe in the future (if returned, it is the correct binding) and live (if there are at most $f$ faulty replicas, as long as $2f + 1$ replicas have acknowledged receiving the ADD request, the binding will be added). We show this by connecting what the client knows ($2f + 1$ tentative acknowledgments) to a starting condition for the U phase, and from there to the steps of the state machine replication. We defer the detailed argument to Appendix A.

# 4 Experimental Evaluation

In this section, we present the implementation of Bonafide and evaluate its performance.

## 4.1 Implementation

To validate our design, we developed a prototype Bonafide implementation. We implemented the add/get, background audit and repair, and the optimized version of the update process of Bonafide (excluding leader election) in C/C++ on Fedora Core 6. The client and server communicate with SFS's asynchronous implementation of SUN RPC [47] in the sfslite library [3]. Client-server communications are authenticated by signatures; we use NTT's ESIGN with 2048-bit keys.

The client uses a proxy, its Bonafide local stub code, to perform `Add`/`Get` operations. The server maintains a MAS, an AST, and a log for buffering ADDs. MAS is implemented as a library and it uses NTT's ESIGN with 2048-bit keys for signatures as well. We use SHA-1 as a secure hash function.

For Byzantine agreement during the U phase, we use the PBFT library [14] ported to Fedora Core 6. PBFT uses MACs for message authentication. During update, every node runs an update server and a PBFT replicated state machine. The leader's update server creates a PROPOSE message and invokes PBFT agreement on the proposal to get consensus across the population on a hash of the proposal. When consensus is achieved, every replica fetches the proposal from the leader, and validates against the agreed upon hash.

We store an AST and a log using Berkeley DB 4.5.20 [1].[5] We use a binary AST to minimize the size of membership witnesses [35, 58]. An AST is stored as a Berkeley database with a BTREE format. Each AST node is stored as a Berkeley DB record, which contains a key, a value, a hash of its left child, and a hash of its right child. The primary key of this DB is the key, and the secondary key is the hash of the entire node content. To search for a value given a key in the AST and to insert a (key, value) binding to the AST while generating a membership witness, we traverse the AST using secondary keys.

(a)

| Operation | Time (ms) Mean (std) |
|---|---|
| `Get` | 3.1 (0.24) |
| `Add` | 1.0 (0.21) |

(b)

| Data loss (%) | Time (s) Mean (std) |
|---|---|
| 0 | 554.5 (54.6) |
| 1 | 612.9 (30.3) |
| 10 | 1147.6 (33.3) |
| 100 | 3521.5 (201.6) |

**Table 2**: (a) `Get` and `Add` time. (b) Audit and repair time.

## 4.2 Performance

We evaluate how the fault-tolerance improvement of Bonafide affects performance and availability. We ran our experiments with four Bonafide replica nodes and one client node. The nodes are outdated PCs with 1.8GHz–3.2GHz Pentium 4 processors, 1GB RAM and 3Com 3C905C Ethernet cards. They are connected over a dual speed 10/100Mbps 3Com switch. On a 1.8GHz machine, ESIGN signature creation and verification of 20 bytes take on average $256\mu s$ and $194\mu s$, respectively. We did not opt for a more up-to-date infrastructure since, by its nature, our application need not be deployed on the bleeding edge of hardware. As we see next, even with this obsolete collection of servers and switch, performance is adequate.

Our experiments initially populate server ASTs with one million bindings of a 128-byte key to a 20-byte SHA-1 hash as the value.

**ADD/GET Time:** We use a simple micro-benchmark client that sends 1000 ADD or GET requests. For ADDs, servers store bindings to their logs and return tentative acknowledgments. For GETs, servers search their ASTs and return values, AST witnesses, and MAS attestations.

We measured Bonafide's GET and ADD response times, by averaging over 1000 requests of each types

| Action | Time (s) Mean (std) |
|---|---|
| Reboot | 86.6 (2.1) |
| Proposal creation | 8.0 (4.0) |
| Agreement | 5.2 (1.0) |
| AST update/Checkpoint | 271.1 (24.8) |
| Total | 370.9 (24.0) |

**Table 3**: U phase duration with 1000 new committed bindings.

with randomly selected keys. In average, GET takes 3.1ms, and ADD takes 1.0ms (Table 2(a)). GET takes more time than ADD does since it involves accesses along an AST path, which incurs multiple disk block accesses. There is a start-up effect in processing GET requests, roughly 100 requests' long, while Bonafide caches top AST levels.

**Audit and Repair Time:** We measured the average time of a basic audit that does not perform any repair over five runs. The disk drive we used was an IBM 40GB IDE disk drive with rotation speed 7200 rpm, average seek time 8.5ms, and buffer size 2MB. The mean audit time of the entire AST is 554.5 seconds and the standard deviation is 9.9% of the mean.

To measure audit and repair time, we simulate random data loss. We delete a fraction of AST nodes randomly at a Bonafide replica and run an audit process. When the audit process finds a lost AST node, the process repairs it synchronously by fetching the AST node from a randomly-chosen remote replica. Table 2(b) shows the mean audit and repair time when a fraction of AST nodes are lost. The more data loss, the longer the repair time due to more access to remote nodes.

Note that our current prototype implementation is not optimized. Several optimizations can improve our prototype performance. For example, more intelligent layout of stored key-value bindings may reduce the random disk access [58], thus improving audit time. Also, an audit and repair process can collect missing AST nodes by fetching them in parallel while performing auditing.

**U Phase Duration:** We measured the duration of the U phase when 1000 new bindings were committed. Table 3 shows the mean and standard deviation of the U phase duration of the leader averaged over five runs; the leader has the highest computation and network bandwidth overhead. Proposal creation indicates time to collect a BATCH certificate and to create a PROPOSE message. Agreement indicates time to run a PBFT agreement, and AST update/Checkpoint indicates time to update the AST with new bindings and to execute remaining COMMIT protocol. Several optimizations can improve the U phase duration. We can reduce reboot time, for example, by using fast boot from the LinuxBIOS project [2].The project claims three seconds boot time from power-on to Linux console. Intelligent caching of AST nodes may reduce the AST update time.

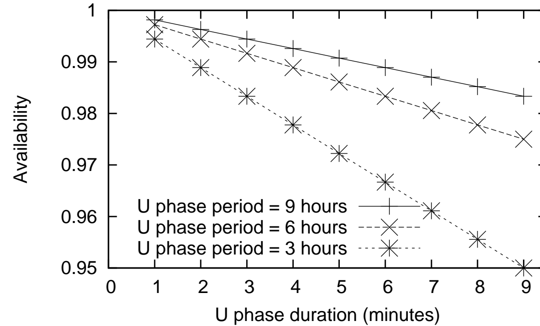


**Figure 6**: Bonafide availability varying the U phase duration and period. Note that the $y$-axis starts at $0.95$.

**Availability:** Finally, we analytically show that Bonafide availability (the ratio of service time to service time plus update time) is high enough for varying U phase duration and period in Figure 6. When the update period is 9 hours, availability is 0.998 and 0.983 for one-minute and nine-minute U phase durations, respectively. Availability decreases linearly as update duration increases. In addition, as we perform update more frequently (i.e., update period decreases), availability decreases more rapidly. For example, when update duration is nine minutes, availability drops from 0.983 to 0.950 as update period changes from 9 hours to 3 hours. However, when we perform update frequently, its duration may decrease since fewer additions are collected, mitigating the effects of unavailability. With one-minute update duration, availability becomes 0.994 despite three-hour periods.

# 5 Discussion

In this section, we discuss the tradeoffs between safety and availability and extensions of Bonafide.

## 5.1 Safety, Availability, and Durability

Our approach implies an operation model that trades off availability for safety, first by containing state changes during small portions of the system's timeline, and by closing off access to the system by its clients while those state changes are incorporated. Though different applications might fare differently with such a trade-off, we believe that applications like Bonafide, including also notarizing documents [23] and auditing for accountability [24, 58] are appropriate practical candidates.

Bonafide can have tradeoffs between durability and availability. This can be tuned with the frequency of update. Frequent update improves durability since it reduces the probability of $N$ replica faults in an S phase, but it reduces availability since the Bonafide service is not available to clients during update.

Availability can be improved in two ways: (1) somehow removing safely the exclusion of service requests

during U phases and (2) increasing the frequency of U phases without halting the service process.

First, it is possible to run the U phase at the same time as a S phase, if the processes executing each can be adequately isolated. For instance, a combination of virtualization and trusted execution (e.g., Intel's LaGrande technology or AMD's Presidio extensions) can ensure that a new operating system image can be late-launched (i.e., "booted") in isolation of any currently running S phase software in a separate execution domain. While the U phase is running, the S phase executing in a separate domain can still handle requests for the previous snapshot of the service. The same effect could be obtained in perhaps less complexity by separating the U and S phases into different physical machines.

Second, it is possible to increase the update frequency without halting the service process by making update unsynchronized, that is, without requiring that all replicas enter the U phase at the same time. Without the need for clock synchronization among all replicas, the duration of U phases can be much shorter (since we no longer need to accommodate global bounds on clock drift across all nodes) and, as a result, U phases can be much more frequent. We give a sketch of an alternative design for unsynchronized U phases in the next section.

## 5.2 Extensions

**$fT$-bound:** The fault threshold for a single U phase in Bonafide can be extended to a multi-phase $fT$-bound model, in which the cumulative number of faults in $T$ consecutive U phases is bounded by $fT$ for some fraction $f$, but there can be phases in which more than fraction $f$ replicas are faulty. Such a failure model may require multi-phase recovery and an extension of MAS to hold, instead of individual registers, an *append-only queue* with at least $T$ positions, akin to an A2M [16].

**Early Commitment:** Bonafide does not guarantee that a mapping for which a client collects $2f + 1$ tentative acknowledgments in any S phase is committed during the following U phase when there are more than $f$ faults during the S phase. By extending MAS, we can provide early commitment that guarantees a mapping is committed during the following U phase. The attested storage needs to bound the number of the entries appended during a single S phase. Once this storage reaches its bound, it does not accept more appends until it is flushed out during the next U phase.

In early commitment, during the S phase, a replica appends ADDs to the *bounded* attested storage and sends a TENTREPLY message with a MAS `Lookup` attestation for each ADD. Essentially, the MAS is used as a trusted un-erasable ADD buffer during an otherwise untrusted S phase. As a result, unlike the protocol described earlier, when the client collects $2f + 1$ tentative `Add` acknowledgements containing buffering MAS attestations, it can complete the request immediately, since that ADD is guaranteed to be reflected in the next AST addition.

**Advanced Search:** To focus on a tiered fault framework, we present a long-term key-value service with a minimal search interface. Extending the main data structure for advanced search is possible. For example, recent research shows a way for running generalized SQL queries on authenticated databases [21].

**Caching for S phases:** Bonafide can employ caching replicas that serve `Get` requests to increase throughput and availability. These caching nodes can serve requests continuously, i.e., they are available during both S and U phases. They use digital signatures that are created by Bonafide replicas to vouch for bindings whose freshness is approximately guaranteed with timestamps. The caching nodes can grow and shrink dynamically depending on workload without manual intervention.

**Unsynchronized U Phases:** Our current design requires a synchronized execution of all U phases in the entire population, which requires bounds on the drift of all nodes' clocks, which in turn requires a long U phase (to accommodate realistic clock drift bounds). As a result, U phases are infrequent to be mostly off-line.

We are exploring an alternative design that does not require a synchronized execution of all U phases. At a high level, when a replica $x$ in its U phase wishes to send a message $m$ to another replica $y$ that is not guaranteed to be in its U phase, replica $x$ asks $y$'s S phase to store that message in an untrusted "mail box" for $y$'s subsequent U phase. When at a later time replica $y$ enters its U phase, it checks its "mail box" for messages from other replicas' U phases, which it uses to make progress.

There are several challenges with this approach. First, the mail box is untrusted (it lives in the address space of the S phase and is subject to the bottom tier of the fault model) which means that messages stored in it may be lost or corrupted. Second, whereas our current, synchronized design executes an entire agreement protocol exchange within a single U phase, this unsynchronized design would have to take multiple rounds of U phases at all involved replicas to complete each agreement protocol (one U phase at each replica to process all messages in its mail box and to transmit the next set of messages). On the other hand, given that there would be no need for clock synchronization, an unsynchronized design can have more frequent but shorter U phases at each replica (say one-second-long U phases every few minutes or so). We are currently adapting a protocol akin to Byzantine Disk Paxos [7], a shared-memory version of Byzantine Paxos that models well communication via unreliable mail boxes.

**Upgrades:** Bonafide does not require that the cryptographic tools it uses (hash functions, digital signatures)

remain inviolate forever; as long as it is migrated to a new algorithm for hashing or signing before the old algorithm has been completely compromised, it can retain its guarantees. Upgrades require an agreement (via the U phase), using a special `Upgrade` request, handled similarly to `Add` requests (buffered, then committed, then executed). Upgrades can include hardware upgrades (e.g., the migration of one MAS device on a particular replica to a newer device, updating the replica membership and public keys to all those who receive the upgrade), software upgrades (e.g., the installation at a replica of a software module for a new cryptographic function), regular membership updates (switching public keys or locations for a replica), or algorithms in use. The latter case requires that all replicas have the new software for a new algorithm; the system cannot migrate from RSA signatures to a (fictional) new RSA++ algorithm until at least a strong quorum of replicas speak RSA++. As a result, an `Upgrade` request to algorithm RSA++ executes only if all replicas in the membership list already have software for the algorithm; otherwise, the request is replaced by a no-op. For hash function upgrades, in particular, the service state must be upgraded as well. This can be done gradually, a small number of AST subtrees per U-phase, by replacing node labels of nodes from tree leaves up to the root. While this upgrade takes place, some tree nodes will have labels computed with the old algorithm, and some labels with the new algorithm, but that is not a problem, until the old algorithm is ultimately and completely compromised.

# 6 Related Work

## 6.1 BFT Systems

Byzantine-fault tolerant state machine replication has received much attention in the systems community since PBFT [13]. Systems such as PBFT-PR [14] and COCA [59] have employed proactive recovery to reduce the vulnerability window. In these systems, a node is periodically rejuvenated, checking and repairing service state. COCA, in particular, shares a similar goal with Bonafide, i.e., the maintenance of a mapping from names to authenticators, but does not account for long-term operation in its structure or assumptions.

Researchers have proposed few improvements on PBFT to improve the 1/3 fault bound. Like PBFT, BFT2F [34] provides safety and liveness with up to 1/3 faulty replicas and then only fork* consistency (a weaker property than linearizability) when faults grow up to 2/3 of the population. Although closer in spirit to our work, since it acknowledges that multiple fault thresholds might be useful towards different guarantees, BFT2F requires state at the clients, which is unreasonable for long-term preservation services, and offers fork* consistency at its weakest fault threshold, which is inappropriate for an archival lookup service. Finally, our own A2M-enabled BFT protocols [16] improve fault bounds by using A2M. In particular, A2M-PBFT-EA provides both safety and liveness with up to fewer than 1/2 faulty replicas. In this work we use some of the insights we gained in the A2M work, but focus instead on the notion of the tiered fault framework in a long-term service.

A2M [16] is a trusted primitive that removes the ability of faulty components to *equivocate*—tell different lies to different peers. Though a powerful primitive for BFT protocols, A2M is lacking MAS's mode bit, which is critical for ensuring phase separations. In addition, A2M has more complicated internal structures and interfaces to account for linearizing requests and handling view changes. A2M has a set of trusted, undeniable, ordered logs, and it gives attestation of any entry or the last entry in the log or attestation vouching for some sequence numbers are skipped. In contrast, MAS has a set of storage slots with a simple write/read interface.

Recent work in Byzantine-fault tolerant storage systems has focused on developing efficient erasure-coding based block storage protocols [12, 22, 26] that reduce storage overhead. Hendricks, Ganger, and Reiter [26] developed the state-of-the-art BFT ($m$, $n$) erasure-coding block storage protocol to optimize reads and large writes. To tolerate $f$ faults, the protocol requires $m \geq f+1$ out of $n = m+2f$ servers. These block storage protocols do not differentiate components of the systems and are not designed for long-term operation.

## 6.2 Differentiating Trust Levels

Researchers have differentiated trust levels on system components, failure types, and failure thresholds. The wormholes model is a *hybrid system model* where the system is decomposed into payload subsystems with weak assumptions and wormhole subsystems with strong assumptions and the two communicate through wormhole gateways [51, 53]. Wormholes such as the Timely Computing Base (TCB) [52] and the Trusted Timely Computing Base (TTCB) [17, 18, 41] provide concrete services such as timely execution and trusted block agreement to payload subsystems. TCB and TTCB are synchronous and fail by crashing. Similarly, we hybridize system components using tiers with different functionalities but we distinguish components explicitly for long-term operation and do in a finer granularity with different fault thresholds and in a more general way.

*Hybrid fault models* differentiate failure types on homogeneous systems: some nodes can have benign faults and others can have Byzantine faults [38, 50]. Furthermore, Byzantine faults are classified into malicious symmetric and malicious asymmetric faults [50]. Modified versions of the classic agreement algorithms can lead to

more flexible fault tolerance guarantees [9, 29, 50].

There has also been research on applying *different fault thresholds* to different sites or clusters. The multi-site threshold model differentiates two types of failures — site failures and process failures — in multi-site systems [28]. The model uses a fault threshold for the number of sites and a vector of fault thresholds, each of which is assigned to a site to account for a different process-failure probability depending on sites. In our tiered fault framework, each site is a tier. Yin et al. [57] proposed an architecture that separates execution from agreement: two groups of replicas—$N$ agreement and $M$ execution replicas—by dividing functionalities. This architecture can tolerate $\lfloor \frac{N-1}{3} \rfloor$ faults and $\lfloor \frac{M-1}{2} \rfloor$ faults, thus assigning different thresholds for the clusters. This partition is done based on functionalities. In our framework, each cluster is a tier. In Bonafide, we differentiate components based on functionalities but do at a finer level.

## 6.3 Long-term Stores

Self-certifying bitstore systems such as Glacier [25], PAST [44], OceanStore [31], Carbonite [15], and Antiquity [55] have addressed durability comprehensively. Authenticity is addressed by expecting all stored data to be *self-certifying*: the name of the datum is an *authenticator* for that datum, and can be used to verify its contents (e.g., via a cryptographic hash). However, such systems leave out of scope where those authenticators come from. It is precisely this gap that Bonafide seeks to fill: providing a long-term store for *non-self-certifying* information.

The LOCKSS system [36] is a digital preservation system not requiring an inviolable 1/3 fault bound. However, this system is probabilistic in nature and does not provide hard safety or liveness guarantees as Bonafide does. POTSHARDS [48] is a long-term storage system that relies on multiple *separately-managed* archives. It uses secret splitting and stores shares into the archives to prevent accidental disclosure. Each object has a mapping between its object identifier and a hash for integrity. In POTSHARDS, each replica in an archive site is trusted or untrusted in its entirety. In comparison, in our work, only a small component at each replica is trusted (MAS), the update software can fail at up to a third of the population at the same time, and the service software can briefly fail everywhere at the same time without affecting safety.

CATS [58] is a single-server service that provides strong accountability of actions done by the server in a single authority and clients. Its approach is not to mask faults through replicated servers, but to detect faults and punish actors responsible for the faults. Its auditing scheme catches server rollback attacks probabilistically. In comparison, Bonafide provides *hard* safety and liveness guarantees under its fault assumption and considers replicated servers.

# 7 Conclusion

Long-term services that operate reliably are hard to construct. This work represents a step towards understanding better system structuring for long-term services that can lead to safer solutions. We present a tiered fault framework that partitions system components of nodes in different tiers, each enjoying a different fault threshold. We have designed and implemented Bonafide, a long-term key-value store that provides integrity under a three-tier Byzantine fault model. We hope that our work provides a framework for building more dependable long-term storage services.


# Acknowledgments

We would like to thank the anonymous reviewers for their comments and our shepherd, Alina Oprea, for her guidance.


# References


[1] Berkeley DB. `http://www.oracle.com/database/berkeley-db/index.html`.

[2] Linuxbios. `http://linuxbios.org`.

[3] sfslite. `http://www.okws.org/doku.php?id=sfslite`.

[4] Trusted Computing Group (TCG). `http://www.trustedcomputinggroup.org/`.

[5] 104th Congress, United States of America. Public Law 104-191: Health Insurance Portability and Accountability Act (HIPAA), Aug. 1996.

[6] 107th Congress, United States of America. Public Law 107-204: Sarbanes-Oxley Act of 2002, July 2002.

[7] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Byzantine Disk Paxos: Optimal Resilience with Byzantine Shared Memory. In *PODC*, 2004.

[8] T. W. Arnold and L. P. V. Doorn. The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer. *IBM Journal of Research and Development*, 48(3/4), 2004.

[9] M. H. Azmanesh and R. M. Kieckhafer. New hybrid fault models for asynchronous approximate agreement. *IEEE Trans. on Parallel and Distributed Systems*, 45(4), 1996.

[10] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale. A Fresh Look at the Reliability of Long-term Digital Storage. In *EuroSys*, 2006.

[11] A. Buldas, P. Laud, and H. Lipmaa. Accountable certificate management using undeniable attestations. In *ACM CCS*, 2000.

[12] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. In *IEEE DSN*, 2006.

[13] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI*, 1999.

[14] M. Castro and B. Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *OSDI*, 2000.

[15] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *NSDI*, 2006.

[16] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested Append-Only Memory: Making Adversaries Stick to their Word. In *SOSP*, 2007.

[17] M. Correia, N. F. Neves, L. C. Lung, and P. Verissimo. Low complexity byzantine-resilient consensus. *Distributed Computing*, 17(3), 2005.

[18] M. Correia, P. Verissimo, and N. F. Neves. The design of a COTS real-time distribued security kernel. In *European Dependable Computing*, 2002.

[19] C. De Cannière and C. Rechberger. Preimages for Reduced SHA-0 and SHA-1. In *CRYPTO*, 2008.

[20] M. Factor, D. Naor, S. Rabinovici-Cohen, L. Ramati, P. Reshef, and J. Satran. Preservation datastores: Architecture for preservation aware storage. In *IEEE MSST*, 2007.

[21] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Super-efficient verification of dynamic outsourced databases. In *RSA Conference—Crypto Track*, 2008.

[22] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *IEEE DSN*, 2004.

[23] S. Haber and W. S. Stornetta. How to Time-stamp a Digital Document. *J. of Cryptology*, 3(2), 1991.

[24] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical Accountability for Distributed Systems. In *SOSP*, 2007.

[25] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *NSDI*, 2005.

[26] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-Overhead Byzantine Fault-Tolerant Storage. In *SOSP*, 2007.

[27] G. Hunt, M. Aiken, M. Fähndrich, C. Hawblitzel, O. Hodson, J. Larus, S. Levi, B. Steensgaard, D. Tarditi, and T. Wobber. Sealing OS processes to improve dependability and safety. In *EuroSys*, 2007.

[28] F. P. Junqueira and K. Marzullo. The virtue of dependent failures in multi-site systems. In *HotDep*, 2005.

[29] R. M. Kieckhafer and M. H. Azamanesh. Reaching approximate agreement with mixed mode faults. *IEEE Trans. on Parallel and Distributed Systems*, 3(1), 1994.

[30] P. C. Kocher. On certificate revocation and validation. In *Financial Cryptography*, 1998.

[31] 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.

[32] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2), 1998.

[33] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM TOCS*, 1992.

[34] J. Li and D. Mazières. Beyond One-Third Faulty Replicas in Byzantine Fault Tolerant Systems. In *NSDI*, 2007.

[35] P. Maniatis and M. Baker. Secure History Preservation Through Timeline Entanglement. In *USENIX Security*, 2002.

[36] P. Maniatis, M. Roussopoulos, T. Giuli, D. S. H. Rosenthal, and M. Baker. The LOCKSS Peer-to-Peer Digital Preservation System. *ACM TOCS*, 23(1), 2005.

[37] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, 1987.

[38] F. Meyer and D. Pradhan. Consensus with dual failure modes. In *IEEE Fault-Tolerant Computing*, 1987.

[39] M. Naor. Bit commitment using pseudorandomness. *Journal of Cryptology*, pages 151–158, 1991.

[40] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI*, 1996.

[41] N. F. Neves, M. Correia, and P. Verissimo. Solving Vector Consensus with a Wormhole. *IEEE Trans. on Parallel and Distributed Systems*, 16(12), 2005.

[42] C. Preimesberger. Intel Faces Up to E-Mail Retention Problems in AMD Lawsuit. `eWeek.com`, Mar. 2007. Fetched on 10/9/2007 from `http://www.eweek.com/article2/0,1759,2101674,00.asp`.

[43] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *FAST*, 2002.

[44] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *SOSP*, 2001.

[45] R. Sekar, V. N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *SOSP*, 2003.

[46] P. Sousa, N. F. Neves, and P. Veríssimo. Proactive Resilience through Architectural Hybridization. In *ACM Symposium on Applied Computing*, 2006.

[47] R. Srinivasan. RPC: Remote procedure call protocol specification version 2. RFC 1831, Network Working Group, 1995.

[48] M. W. Storer, K. Greenan, E. L. Miller, and K. Voruganti. POTSHARDS: Secure Long-Term Storage Without Encryption. In *USENIX ATC*, 2007.

[49] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST*, 2008.

[50] P. Thambidurai and You-keun Park. Interactive consistency with multiple failure modes. In *SRDS*, 1988.

[51] P. Verissimo. Uncertainty and Predictability: Can they be reconciled? *LCNS: FuDiCo*, 2584, 2003.

[52] P. Verissimo, A. Casimiro, and C. Fetzer. The timely computing base: Timely actions in the presence of uncertain timeliness. In *IEEE DSN*, 2000.

[53] P. E. Verissimo. Travelling through wormholes: a new look at distributed systems models. *ACM SIGACT News*, 37(1), 2006.

[54] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *SIGCOMM*, 2004.

[55] H. Weatherspoon, P. Eaton, B.-G. Chun, and J. Kubiatowicz. Antiquity: Exploiting a secure log for wide-area distributed storage. In *EuroSys*, 2007.

[56] Y. Xie and A. Aiken. Saturn: A SAT-Based Tool for Bug Detection. In *International Conference on Computer Aided Verification*, 2005.

[57] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement from Execution for

Byzantine Fault Tolerant Services. In *SOSP*, 2003.
[58] A. R. Yumerefendi and J. S. Chase. Strong Accountability for Network Storage. In *FAST*, 2007.
[59] L. Zhou, F. B. Schneider, and R. V. Renesse. COCA: A secure distributed online certification authority. *ACM TOCS*, 20(4), 2002.

## Notes

[1] We use the terms *fault bound* and *fault threshold* interchangeably.
[2] The window of vulnerability varies depending on system conditions. For example, if some replicas' state is corrupted, the window becomes large.
[3] This metaphor is usually attributed to Reagan Moore.
[4] With our recent A2M-PBFT-EA protocol [16], we can improve this fault bound from $1/3$ to $1/2$. We leave the details out of this paper to keep the exposition simple.
[5] We chose Berkeley DB since it was readily available, but our design would be compatible with any block-store such as Venti [43].

# A Correctness Arguments

In this appendix, we prove that Bonafide provides the integrity property under the tiered Byzantine-fault model. In the proof, we denote by $s(r)$ the S phase of round $r$ and by $p(r)$ the U phase after $s(r)$. Without loss of generality consider a binding $(k, v)$.

**Lemma A.1.** *For every* $\texttt{Add}(k, v)$ *request accepted by* $2f + 1$ *replicas during a S phase in which there is no more than* $f$ *faulty replicas out of* $3f + 1$ *total replicas,* $(k, v)$ *appears in any valid* BATCH *certificate.*

*Proof.* We say an $\texttt{Add}(k, v)$ request is accepted if there are at least $2f+1$ replicas that receive the request; a client can ensure that the request is accepted by checking authenticated tentative ADD responses. Let $Q_a$ denote this set of replicas. At the start of $p(r)$, each replica multicasts a BATCH message to other replicas. The leader collects $2f + 1$ distinct BATCH messages that form a BATCH certificate. Let $Q_b$ denote the set of replicas that form this certificate. $Q_a \bigcap Q_b$ includes at least one non-faulty replica that receives the ADD request since in the S phase the number of faulty replicas is no more than $f$. Therefore, the accepted request is contained in the BATCH certificate. □

**Lemma A.2.** *A stable checkpoint certificate of the previous round appears in any valid* BATCH *certificate.*

*Proof.* We show that at $p(r)$ the BATCH certificate contains the stable checkpoint ($2f + 1$ matching MAS attestations) of $p(r - 1)$. At $p(r - 1)$, there are at least $2f + 1$ replicas, each of which creates a stable checkpoint certificate and puts the certificate to its MAS. Let $Q_p$ denote this set of replicas. $Q_p \bigcap Q_b$ intersects at at least $f + 1$ replicas and hence includes at least one common non-faulty replica between two quorums. This replica ensures that the stable checkpoint of the previous round is included in the BATCH certificate. Therefore, the BATCH certificate of $p(r)$ contains the correct stable checkpoint of $p(r - 1)$. □

**Theorem A.3.** *If a binding* $(k, v)$ *is accepted at* $s(r)$ *and* $k$ *is not in the AST, the binding is correctly read (or temporarily unavailable) at all* $s(r')(r' > r)$.

*Proof.* From Lemmas A.1 and A.2, we know that a PROPOSE message with a valid BATCH certificate contains a correct stable checkpoint certificate of the previous round and bindings received during a S phase in which there is no more than $f$ faulty replicas. When the leader invokes PBFT with the PROPOSE message, PBFT ensures that all non-faulty replicas agree on the PROPOSE message. Each such replica checks that $k$ does not exist; if necessary, the replica may perform state transfer for this validation. If $k$ does not exist, the replica inserts $(k, v)$ into the AST, computes a new AST digest, and appends it to MAS. Finally, each replica creates a stable checkpoint certificate by collecting $2f + 1$ matching UCHECKPOINT messages and appends the certificate to its MAS.

Now, suppose a client gets a reply certificate ($f + 1$ matching MAS attestations) of $\texttt{Get}(k)$ at $s(r + 1)$. The reply certificate contains at least one *up-to-date* replica since a non-faulty replica enters $s(r + 1)$ only after creating a stable checkpoint certificate. Therefore, a client correctly reads value $v$ when it queries with $k$ at $s(r+1)$.

Once $(k, v)$ is inserted into Bonafide at $p(r)$, it is clear that $p(r + 1)$ carries $(k, v)$ from $p(r)$ correctly with the same argument we make for $p(r - 1)$ and $p(r)$ above since we have a correct AST digest. We can inductively argue the same holds for $p(r + i)$ and $p(r + i + 1)$ for all $i \geq 0$. Therefore, when a client gets a reply certificate for $\texttt{Get}(k)$ at all $s(r + i)$ ($i > 0$), the client receives correct $(k, v)$. □