The Case for Byzantine Fault Detection

Andreas Haeberlen[†‡] Petr Kouznetsov[†] Peter Druschel[†]

[†]*Max Planck Institute for Software Systems* [‡]*Rice University*

1 Introduction

Distributed systems are subject to a variety of faults and attacks. In this paper, we consider general (Byzantine) faults [13], i.e. a faulty node may exhibit arbitrary behavior. In particular, a faulty node may corrupt its local state and send arbitrary messages, including specific messages aimed at subverting the system. Many security attacks, such as censorship, freeloading, misrouting, and data corruption, can be modeled as Byzantine faults.

Systems can be protected with Byzantine fault tolerance (BFT) techniques, which *mask* a bounded number of Byzantine faults, e.g. using state machine replication [4]. BFT is a very powerful technique, but it has its costs. In a practical system that needs to tolerate up to f concurrent Byzantine faults, BFT cannot be implemented with less than $3f+1$ replicas [3]. Moreover, BFT scales poorly to large replica groups; as more servers are added, the throughput of the system may actually decrease [7].

In this paper, we explore an alternative approach that aims at *detecting* rather than masking faulty behavior. With this approach, the system does not make any attempt to hide the symptoms of Byzantine faults. Rather, each node is equipped with a detector that monitors other nodes for signs of faulty behavior. If the detector determines that some node has become faulty, it notifies the application software, which can then take appropriate action. For example, nodes can cease to communicate with the faulty node; once all correct nodes have followed suit, the faulty node is isolated and the fault is contained.

Fault detection is weaker than masking. For instance, detection is insufficient for dealing with faults that have serious and irreversible effects, such as deletion of all copies of an important document. However, detection may offer an efficient and scalable alternative to BFT for faults that have limited or recoverable effects, including freeloading, censorship, and denial-of-service.

We are interested in fault detectors that provide *accountability* [17]. With such detectors, each action is undeniably associated with the identity of the node that has performed the action, allowing the system to gather *irrefutable evidence* of faulty behavior.

The fault detection systems we consider should guarantee at least two properties. The system should be *complete*: whenever a correct node observes the effects of faulty behavior, the system eventually generates evidence against at least one faulty node. Also, the system should be *accurate*: it never generates valid evidence against a correct node.

Adding accountability to a distributed system has several important advantages, regardless of whether the systems uses BFT or not: first, any faulty behavior by a node is guaranteed to be detected. Second, the evidence produced by the detector can be used to convince third parties that a fault has occurred. Third, the evidence enables the system to resolve he-said-she-said situations in which two nodes accuse each other of being faulty. Lastly, the presence of accountability alone *deters* certain types of attacks on a system, because it identifies and exposes faulty nodes.

Our goals in this paper are threefold: first, we examine the trade-offs between fault detection and traditional BFT. Second, we give a precise definition of the class of Byzantine faults that can be detected with our approach. Finally, we briefly sketch a practical system that implements a Byzantine fault detector.

1.1 The case for fault detection

Techniques that mask Byzantine faults are perhaps easier to use than fault detection systems, since they provide the application designer with the abstraction of a system in which failures simply do not occur. So what reasons are there to opt for fault detection?

Detection requires less replication: If a practical system can suffer up to f concurrent faults, BFT cannot be implemented with less than $3f+1$ nodes [3]. As we will

show, detection can be accomplished with only $f + 1$ nodes[1]. The complementary view of this point is that BFT requires the *fraction* of faulty nodes in the system to remain below 33% at all times, while a correct node can reliably detect faults irrespective of the fraction of faulty nodes.

Detection systems need only be provisioned for the average load: In a BFT system, all replicas must process each request promptly, since the client cannot make progress before most of them have responded. In a detection-based system, however, a *single* replica can process each request and respond immediately; the other replicas can later check the response during a period of light load. Hence, a BFT system must be provisioned such that each machine can handle the *peak* load, while in a detection system, each machine must merely be able to handle the *average* load.

Detection is cheaper: Detection avoids the consensus required by BFT, and it enables extensive aggregation of messages, state and processing associated with detection. Also, there is no need for strong consistency among the replicas, which makes it much easier to handle changes to the replica group.

Detection systems do not only require fewer resources than BFT, they also have some functional advantages that benefit distributed systems whether or not they use BFT:

Detection enables a timely response to faults: In a system that does not use BFT, once correct nodes obtain evidence of a fault, they can stop communicating with the faulty node and thus isolate it. Also, correct nodes can initiate recovery, e.g. by creating additional replicas of any objects affected by the fault, or by alerting a human operator who can repair the faulty node. Timely repair can also help BFT-based systems to stay within their bound for the number of concurrent faults.

Detection provides a deterrent: The mere presence of a detection system can reduce the likelihood of certain faults. For example, it can discourage freeloading and censorship in peer-to-peer systems by creating a disincentive to cheating, since a faulty node risks isolation and expulsion from the system. Furthermore, if the system maintains a binding from node identifiers to real-world principals, then the owner of a faulty node can be exposed and held responsible. Reducing the frequency of certain faults also benefits a BFT-based system, allowing it to more easily maintain its error bound.

1.2 Uses of fault detectors

Next, we sketch general application areas for fault detection systems, along with some specific examples.

[1]This does not contradict the impossibility results of [3] because detection systems are not based on agreement.

Systems with recoverable state: Network file systems and distributed information systems typically perform periodic backup snapshots to ensure data durability. It is usually acceptable for these systems to revert to the latest snapshot in case of a serious malfunction or attack. However, faults must be discovered quickly in order to prevent corrupt data from spreading to the backups. By adding a detector, these systems can bound the time to detection for a very general class of faults.

Redundant systems: Decentralized systems and systems based on BFT mitigate or mask the effect of a limited number of faults through redundancy. However, if faulty nodes are not discovered and removed quickly, they can accumulate over time and eventually lead to a system failure. Using a detector, faulty nodes can be identified and isolated before they can cause any serious harm.

Systems that span multiple administrative domains: Such systems can benefit from accountability to keep the players honest and to apportion blame, e.g. in federated databases and hosted Web services, or to discourage freeloading and censorship in peer-to-peer systems. The Internet's inter-domain routing system is another example. In the case of a malfunction, detectors could not only identify which party is at fault; they would also produce evidence that would allow other parties to prove to their customers that they are not to blame.

2 Definitions

A perfect detection system would immediately detect any Byzantine fault. The power of a practical, efficient detection system, however, is necessarily limited. In this paper, we will assume that the detector on a correct node can observe all messages sent and received by that node. This clearly means that some Byzantine faults are not observable and therefore *cannot* be detected.

2.1 Examples of detectable behavior

Before defining formally the classes of Byzantine faults that can be detected using observable messages, we discuss a simple example protocol that has only two methods: A `put` method, which is used to store an object on a node, and a `get` method, which is used to retrieve it. Figure 1(a) shows a simple example of a message exchange in which node B receives an object from node A and later delivers it to another node C.

Now assume that node B is faulty and wants to prevent node C from obtaining the object. There are two ways for B to achieve this. One is to break the protocol and deny C's request, as shown in Figure 1(b); we call this behavior *detectably faulty*. The other is to pretend

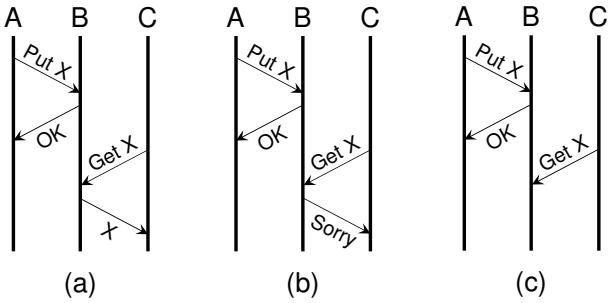

Figure 1: *In this simple message exchange, node B is (a) correct, (b) detectably faulty, and (c) detectably ignorant, provided that nodes A and C are correct.*

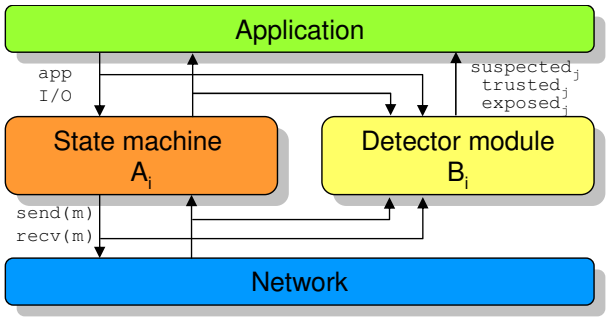

Figure 2: *Information flow between application, protocol, and detector module on node i*

that it has not received the request message, as shown in Figure 1(c); we call this behavior *detectably ignorant*.

In both cases, the fault is detectable because it affects the message exchange observed by the correct nodes. However, a node might become faulty but continue to follow the protocol exactly as if it were correct. Such a fault *cannot* be detected with our approach. Similarly, if a fault is completely internal to one node or affects only messages sent to other faulty nodes, it is not observable by any correct node and therefore cannot be detected.

In the rest of this section, we provide a formal definition of detectable faultiness and ignorance, and we introduce a set of guarantees that can be given by a detection system.

2.2 System model

We consider a set Π of *nodes*. Every node i is modeled as a state machine A_i and a detector module B_i (Figure 2). Informally, we say that a node i is correct if it respects the specifications of both A_i and B_i. Otherwise, the node is faulty.

Nodes communicate with each other through message passing. We assume that messages are uniquely identified. For a message m, let *sender*(m) and *receiver*(m) denote the sender and the receiver of m, respectively. For the moment, we do not put any restrictions on local processing time and communication delays. However, we assume that, after some number of retransmissions, a message sent from a correct node to a correct node is eventually received.

An *event* is either $\mathit{send}_i(m) \in O_i$, where $i = \mathit{sender}(m)$, or $\mathit{receive}_j(m) \in I_j$, where $j = \mathit{receiver}(m)$, or an application-specific input or output.

An *execution* E is a sequence of events such that in E, each m is sent and received at most once, and each $\mathit{receive}_i(m)$ is preceded by the corresponding $\mathit{send}_j(m)$. We distinguish events associated with the state machine A_i and events associated with the detector module B_i. $E|A_i$ denotes the subsequence of E that consists of all events associated with A_i in E, and $E|B_i$ denotes the subsequence of E that consists of all events associated with B_i in E. We say that a node i is *correct* in E if (1) $E|A_i$ (respectively, $E|B_i$) *conforms* to A_i (respectively, B_i), i.e., if the sequence of outputs produced in $E|A_i$ ($E|B_i$) is legal, given A_i (B_i) and the sequence of inputs in $E|A_i$ ($E|B_i$), and (2) if E is infinite, then both $E|A_i$ and $E|B_i$ are also infinite. Otherwise we say that i is *faulty* in E.

2.3 Detectable faultiness and ignorance

We define a *history* of a node i as a sequence of events of A_i. A history h of a node i is *valid* if it conforms to A_i, i.e. if, given the sequence of incoming messages and application-specific inputs in h, A_i could have produced the sequence of outgoing messages and application-specific outputs in h. A pair (h_1, h_2) of histories of i is *consistent* if h_1 is a prefix of h_2, or vice versa. If i is a correct node, one trivial example of a valid history is $E|A_i$.

Let $\mathcal{M}(E)$ denote the set of messages received by the nodes in an execution E. We assume that there exists a *history map* φ that associates every message $m \in \mathcal{M}(E)$ with a history of *sender*(m). For a correct node, $\varphi(m)$ is the prefix of the local execution $E|\mathit{sender}(m)$ up to and including *send*(m). Thus, for any message m sent by a correct node, $\varphi(m)$ is valid, and for every pair of messages m and m' sent by a correct node, $\varphi(m)$ and $\varphi(m')$ are consistent.

We say that a message m is *observable in* E if there exists a correct node i and a sequence of messages $m_1, \ldots, m_k$ such that

(i) $m_1 = m$,

(ii) *receive*(m_k) belongs to $E|A_i$,

(iii) for all $j = 2, \ldots, k$: *receive*(m_{j-1}) belongs to $\varphi(m_j)$.

In other words, m is observable if it causally precedes at least one event on a correct node.

We say that a node i is *detectably faulty* with respect to a message m sent by i in an execution E if m is observable in E and satisfies one of the following properties:

(1) $\varphi(m)$ is not valid (for i)

(2) There exists a message m' that was also sent by i and is observable in E, such that $\varphi(m)$ is inconsistent with $\varphi(m')$

The set of nodes causally affected by m and m' (if m' exists) are called *accomplices* of i with respect to m.

We say that a node i is *detectably ignorant* in E if i is not detectably faulty in E and there exists a message m sent to i by a correct node, such that, for all observable messages m' sent by i, $receive_i(m)$ does not appear in $\varphi(m')$.

2.4 Guarantees

When the detector module B_i on a correct node i has seen evidence of faulty behavior on another node j, it sends a *failure indication* to its local application process. We define three different types of indications: $trusted_j$, $suspected_j$ and $exposed_j$. Intuitively, if the module B_i outputs $suspected_j$, there is evidence that j is ignoring certain inputs, e.g. by refusing to accept a service request from a correct node. If it outputs $exposed_j$, there exists a *proof* that j is faulty, i.e. that it has deviated from the specification of its state machine A_j. Finally, B_i outputs $trusted_j$ while none of the other conditions hold.

We can use a definition similar to that of [5, 11] to describe these properties. Thus, the detection system guarantees that the following properties hold in every execution:

- **Eventual strong completeness:** (1) Eventually, every detectably ignorant node is suspected forever by every correct node, and (2) if a node i is detectably faulty with respect to a message m, then eventually, some faulty accomplice of i (with respect to m) is exposed or forever suspected by every correct node.
- **Eventual strong accuracy:** (1) No correct node is forever suspected by a correct node, and (2) no correct node is ever exposed by a correct node.

Note that the detector need not guarantee that a correct node is always trusted by another correct node; it can jump from *trusted* to *suspected* and back, e.g. due to long message delays. Further, if a set of colluding faulty nodes includes at least one detectably faulty node, then *at least one* of them will eventually be exposed or suspected forever; we chose this weaker guarantee to facilitate an efficient implementation of φ. Nevertheless, if there are only finitely many faulty nodes in the system, correct nodes can be affected by their behavior only finitely long.

3 A practical detector for Byzantine faults

To show that detection systems are practical, we now briefly sketch the design of *PeerReview*, a system that can provide the guarantees stated in Section 2.4. A proof can be found in [9]. We have implemented PeerReview and initial results suggest that it is practical and efficient. An experimental evaluation is the subject of a future, full paper.

3.1 Assumptions and goals

For PeerReview, we assume that the system can be modeled as described in Section 2.2, with two additional assumptions: First, the protocol is *deterministic*, i.e. it produces the same outputs given the same sequence of inputs. This is a fairly common assumption in state machine replication [4,12]. Second, nodes have *strong identities* and hold a cryptographic keypair that can be used to sign messages. This can be accomplished, for instance, by giving each node an identity certificate, signed by a certification authority, that ties its public key to its node identifier.

We also make the common assumption that the attacker does not have the ability to break cryptographic signatures. Other than that, the Byzantine nodes may behave arbitrarily and collude with each other.

3.2 Secure histories and commitment

Each node is required to keep a log of all the inputs and outputs of its local state machine A_i. The log is organized as a hash chain, similar to a secure history [15], such that the top-level hash covers the contents of the entire log. Furthermore, each node must frequently *commit* to the contents of its log by publishing an *authenticator*, i.e. a signed copy of its top-level hash value. This makes the log *tamper-evident* and ensures that nodes cannot revise their history [15].

Nodes must sign all messages they send and acknowledge all messages they receive. If a message is not acknowledged after several retries, it is broadcast to the other nodes, who then challenge the node to accept the message. This ensures that a node is suspected by all correct nodes if it refuses to accept a message.

Each message or acknowledgment m contains an authenticator, as well as a short proof that $send(m)$ or $receive(m)$ was the top-level entry of the corresponding log. The recipient extracts the authenticators and,

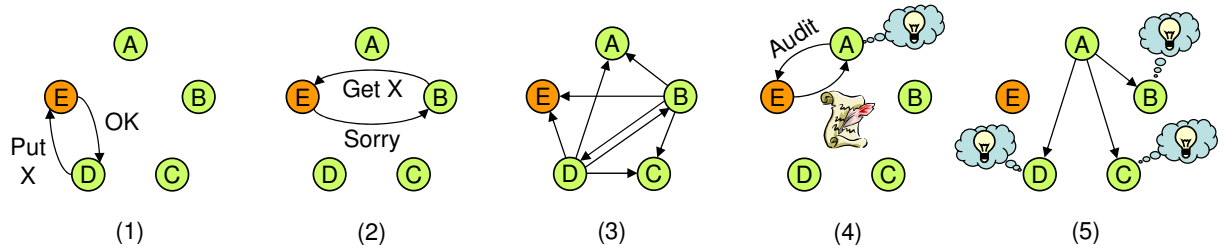

Figure 3: *Node E stores an object for client D (1) and then tries to hide it from client B (2). The two clients broadcast the authenticators they have obtained from E (3). Later, A audits E, discovers the inconsistency, and exposes E (4). Finally, node A broadcasts its evidence against E, so the other nodes can expose E as well (5).*

once in a while, forwards them to other nodes. Thus, interested nodes eventually learn of all authenticators that have been signed by a correct node.

3.3 Auditing

Each node i is periodically *audited* by other nodes. During an audit, the auditor j first asks i for a signed log segment that covers all entries since the last audit. j then validates the log against the most current authenticator it has obtained for i. If i refuses to comply, j begins to suspect i.

Next, j performs a *consistency check* to see if the log matches all the recent authenticators it has obtained for i. If two or more authenticators do not match, then i has either forked its log or is keeping multiple copies. The mismatched authenticators are then made available to other nodes as evidence, who can thus mark i as exposed.

In a third step, j extracts all authenticators from the log segment and forwards them to other nodes. This ensures that, even if i is faulty, other nodes will eventually be aware of all relevant authenticators.

Finally, j performs a *conformance check*. It instantiates a local copy of the state machine A_i and initializes it with a recent checkpoint from the log. Then it replays all the inputs from the log and checks whether the corresponding outputs match the ones in the log. Thus, j can check protocol conformance without an explicit protocol specification. If it detects a divergence, it has obtained a signed confession and can thus expose i.

In a naive implementation of the algorithm, every node audits every other node, requiring $O(N^2)$ messages and computation. Though the messages are small and can be heavily aggregated, this does not scale to large systems. In practice, however, if at most f nodes can be faulty at any given time, it suffices if $f+1$ different nodes audit a given node, which reduces the complexity to $O(f^2)$. The details of this optimization and a full evaluation will be provided in a subsequent full paper.

3.4 Checking evidence

If a node j detects a fault on a node i, it obtains one of two types of evidence. If i is detectably faulty, j obtains either a) an authenticator and a log, both of which are signed but do not match, or b) a signed log segment that fails the conformance check. Both constitute a signed confession. If i is detectably ignorant, j obtains a challenge (e.g. a request for a certain log segment) that i cannot answer, except by providing a signed confession.

Both types of evidence can be distributed to the other nodes, who can verify them independently, either by repeating the checks performed by j (in case of a signed confession) or by contacting i and checking its response (in case of a challenge). PeerReview ensures that such a check will always fail for a correct node, since it never produces a signed confession and can respond to any challenge.

The output of the PeerReview failure detector on a given node is reliable if, and only if, the node has a valid copy of the state machine to be run by all the nodes in the system. A node can ensure this, for instance, by obtaining a signed binary program from a trusted authority.

To bound the space required for logs, nodes may be allowed to discard old log entries, e.g. after a month. In this case, older evidence can no longer be verified and must be discarded as well, which eventually allows faulty nodes to return to the system. This is acceptable as long as the system has ample time to respond to the failure and initiate repair. If repair is not an option, e.g. in a large decentralized system, the log must be kept long enough to create a serious disincentive for attackers. Alternatively, the remaining nodes could use Byzantine consensus to permanently revoke faulty nodes' certificates prior to truncating the log.

4 Related work

Our concept of a detection system is based on the failure detectors by Chandra and Toueg [5]. These were defined for crash failures, but Malkhi and Reiter [14] later ex-

tended them to special classes of Byzantine failures. In a more general manner, Doudou et al. [8] have introduced *muteness* failure detectors dealing with nodes that prematurely stop sending algorithm messages. Kihlstrom et al. [11] have introduced several classes of failure detectors that expose *detectable* Byzantine failures. However, they consider classes of algorithms in which all messages are broadcast, and in which processes know when to expect messages from other processes. PeerReview does not require these assumptions.

State machine replication [12, 16] is a classical technique for masking a limited number of Byzantine faults. Today's state-of-the-art BFT techniques, e.g. [4], are based on this idea. Although these techniques perfectly protect the system from Byzantine failures, they are usually not intended to detect the faulty nodes, and they are inherently expensive and not scalable. The BAR model [1] extends the BFT approach to tolerate selfish behavior of rational nodes while providing a mechanism for detecting certain application-specific misbehavior. Our approach is more general and, unlike BAR, it does not inherit the algorithmic complexity of BFT.

Alvisi et al. [2] introduced a technique that monitors quorum systems and raises an alarm if the failure assumptions are about to be violated. This technique is probabilistic and, unlike PeerReview, cannot identify which nodes are faulty.

Intrusion detection systems [6] can detect certain types of protocol violations; however, unlike PeerReview, the heuristics used in IDS tend to produce either false positives, false negatives, or both. Reputation systems such as EigenTrust [10] can be used against Byzantine failures, but, unlike PeerReview, they cannot prevent a coalition of malicious nodes from denouncing a correct node. Finally, trusted computing platforms like TCG/Palladium can detect failures that involve software modifications, but force users to exclusively run certified software. PeerReview merely checks protocol conformance but otherwise allows diverse implementations.

Yumerefendi and Chase [17] proposed accountability as a first-class design principle for dependable distributed systems, but mentioned that general, application-independent techniques were still elusive. We believe that detection systems, and PeerReview in particular, are a major step towards this goal.

5 Conclusion and future work

We have discussed an alternative approach to handling Byzantine faults, in which the system does not mask faults but rather detects and responds to them. We have formally specified the class of faults that can be detected with this approach, and we have sketched the design of a practical system that implements it. To the best of our knowledge, it is the first practical, general-purpose algorithm for detecting Byzantine faults.

Detection promises to reduce the cost of robustness to Byzantine faults, and to increase dependability of systems in which BFT is infeasible or prohibitively expensive. For example, detection offers a relatively efficient defense for freeloading and censorship attacks in large-scale distributed systems. It can provide accountability in systems that span multiple administrative domains, such as federated databases, hosted web services and peer-to-peer systems.

We believe that further research in detection systems will yield a variety of new detectors with different tradeoffs. For example, more powerful detectors could be constructed by adding more sensors, such as attestation, and hybrids between detection and BFT could allow more fine-grained tradeoffs between protection and overhead.

References

[1] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proceedings of SOSP'05*, Oct 2005.

[2] L. Alvisi, D. Malkhi, E. T. Pierce, and M. K. Reiter. Fault detection for Byzantine quorum systems. *IEEE Trans. Parallel Distrib. Syst.*, 12(9):996–1007, 2001.

[3] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4), 1995.

[4] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of OSDI'99*, pages 173–186, 1999.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[6] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.

[7] J. R. Douceur and J. Howell. Byzantine fault isolation in the Farsite distributed file system. In *Proc. of IPTPS'06*, Feb 2006.

[8] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In *EDCC*, pages 71–87, 1999.

[9] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Detecting faulty behavior in distributed systems. Technical Report Max Planck Institute for Software Systems 2006-1, Jul 2006.

[10] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The EigenTrust algorithm for reputation management in p2p networks. In *Proc. 12th International WWW Conference*, May 2003.

[11] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, 2003.

[12] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Prog. Lang. Syst.*, 6(2):254–280, 1984.

[13] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[14] D. Malkhi and M. K. Reiter. Unreliable intrusion detection in distributed computations. In *CSFW*, pages 116–125, 1997.

[15] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, Jan 2002.

[16] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[17] A. R. Yumerefendi and J. S. Chase. The role of accountability in dependable distributed systems. In *Proceedings of the 1st Workshop on Hot Topics in System Dependability*, Jun 2005.