Byzantine Fault Tolerance in Long-Lived Systems

Rodrigo Rodrigues and Barbara Liskov
MIT Computer Science and Artificial Intelligence Laboratory

1 Introduction

Byzantine fault tolerance comprises a set of techniques for building fault-tolerant systems where no assumptions are made about the behavior of faulty nodes. This makes Byzantine-fault-tolerant systems particularly attractive as a defense against malicious attacks that may cause faulty nodes to exhibit arbitrary behavior.

A service that tolerates Byzantine failures (e.g., [1, 8]) must store the service state at a set of replicas. The replicas carry out a protocol that tolerates failures of a subset of them. Usually the system contains $3f + 1$ replicas, and the protocols guarantee correct behavior provided no more than f of them are faulty at the same moment.

These systems do well provided the assumption about the number of simultaneous failures is valid. But if more than f replicas fail, the system fails and no guarantees can be made about its behavior. The question we address in this paper is: what can be done to increase the probability that no more than f replicas are faulty simultaneously?

We address this question using the following simple model. For any attack that could be mounted, there is an *attack window*, A; this is the length of time needed to compromise more than f replicas using that attack. Different attacks have different attack windows. Some attacks require a very small A. For example, if the code on the replicas has a deterministic software bug that allows an attacker to exploit a buffer overrun, then an attacker can launch such legal calls simultaneously and bring the system down in a very short time. Other attacks require more time. For example, an attack that relies on a non-deterministic error might take quite a while.

A system has a *window of vulnerability*, W, during which it allows an adversary to mount an attack. We would like to have $W < A$ since this means that the system cannot be compromised by an attack. This condition is unlikely to be satisfied in a long-lived system with no defenses against the accumulation of faulty nodes; in this case W is infinite (or at least equal to the system lifetime) and therefore we can expect that ultimately an attack will succeed.

This paper proposes counter-measures that can be deployed as part of a replicated system to reduce the size of W, and thus reduce the class of attacks to which the system is vulnerable. Obviously it will not be possible to withstand all attacks via this technique, in particular attacks with very small A. But we will propose techniques that can reduce W to quite a small value.

In the remainder of this paper, we discuss how to lower the value of W. We begin by discussing attacks. Then we discuss some prior work in this area and why it is insufficient. The final section describes the approach we propose.

2 Attacks

This section presents a rough categorization of attacks, as a basis for deciding what countermeasures are needed. We looked at existing taxonomies from the security community (e.g., [6]) but they make distinctions that aren't relevant to us. Additionally they are concerned about both privacy and integrity of stored information while we are concerned only with the latter.

In a distributed system, there are two things that can be attacked: the nodes and the network.

Attacks on the network can remove, insert, delay, and corrupt messages. They can also detect hidden information, e.g., the key used to sign a message. We assume the standard defenses against such attacks, e.g., the use of encryption, signatures, and nonces. In addition, we assume replication algorithms are designed to work in asynchronous networks that may lose or duplicate messages. These defenses imply that attacks on the network cannot cause a node to malfunction: an attack can affect liveness but not safety. For example, an attack that corrupts or discards all of a node's messages, or an attack on a router that cuts off part of the network, can prevent a node from communicating with others. Or a denial of service attack might flood a node with bad requests, thus preventing it from making much progress on good ones.

The mechanisms to deal with these attacks are outside the scope of this paper. We do assume, however that such an attack has a certain duration, i.e., it ends eventually.

Our concern is with attacks that cause nodes to malfunction. Such attacks can corrupt one or more of the following:

- Code – An attacker can corrupt or replace the running service code, or any other code that surrounds the service application (e.g., the OS code).
- Data – An attacker can wipe out or modify the data stored by the service.
- Hardware – An attacker can modify the hardware so that it no longer performs as expected, e.g., no longer car-

This work is supported by the NSF under cooperative agreement ANI-0225660. Rodrigo Rodrigues is supported by a fellowship from the Calouste Gulbenkian Foundation, and was previously supported by a Praxis XXI fellowship.

ries out the protections relied on by the access control mechanisms.

Nodes can be attacked via the network or physically. A physical attack is one involving a person at the physical location of the node. A network attack either exploits a bug in the code (either OS or application code), or it relies on discovering a secret, e.g., learning the node's root password. Either kind of attack can corrupt the code or data at the node. Only a physical attack can corrupt the hardware. Also a physical attack can exploit avenues not open to a network attack, e.g., replacing the disk that stores the code with a different disk.

An attacker might use several different attack methods in one attack, e.g., corrupt one node via a physical attack, and another by stealing a root password.

Another point is that it is necessary to assume very clever attacks: this is the nature of the game in Byzantine fault tolerance. For example, the attacker might launch a *lying in wait attack*, in which corrupted code continues to perform properly until a sufficient number of nodes has been corrupted, at which point all corrupted nodes destroy their stored state.

3 Prior Work

In this section we discuss two earlier proposals for handling attacks.

First, there have been several *remote attestation* proposals [3, 4]. These proposals rely on secure hardware and a small, trusted OS kernel base to produce a certificate that enables a machine to prove to a remote system what local programs are running.

Code attestation is both insufficient and overkill for a replicated system. It is insufficient because it contains no mechanism for recovering from an attack, and therefore, corrupted nodes can accumulate over time. It is overkill because the client of a replicated service relies on getting enough valid matching replies and doesn't actually need to know about the status of an individual replica.

The other approach is proactive recovery [2]. Here each replica is assumed to have a secure coprocessor that holds its private key and a watchdog timer that periodically (e.g., every 10 minutes) interrupts processing and hands control to a recovery monitor, which restarts the node. When the node is restarted, the coprocessor reinitializes the code running on the main processor from a copy on a read-only disk. The idea is that at this point the code is correct. Then the node runs a restart protocol that corrects its data if that has been corrupted.

Proactive recovery takes care of some of our concerns but not all. It is vulnerable to a physical attack that corrupts the hardware of the replica or that replaces the disk that stores the code copy used at restart. A further point is that the node isn't monitored between restarts, and therefore it can run in a corrupted manner for some time period. Although this period is short (on the order of a few minutes), it might still be large enough to allow an attack to succeed in corrupting more than f nodes simultaneously.

4 Defenses

We can sum up the previous work as follows. Remote attestation detects problems, but provides no provision for reacting to them other than by rejecting a reply. Proactive recovery doesn't detect problems, but does recover from certain kinds of attacks. What is missing in both cases is a way of removing a node that cannot be recovered, e.g., it is dead or the code disk has been replaced with a bad copy.

Removal implies that there must be a way of replacing a bad replica with some other node since otherwise at best it turns a Byzantine failure into a failstop failure. The other node might be a spare kept around for just this purpose. Or there might be a pool of nodes used to replicate many services and replacements are chosen from this pool. In either case, when a node becomes responsible for running a service that wasn't previously its concern, it carries out a state transfer protocol by interacting with the nodes that used to be responsible for that service. Provided state transfer can occur before more than f of these nodes become faulty, we can guarantee that the service continues to operate correctly.

Of course to remove a node we must first detect that it can't be recovered. One possibility is to have a person take on this responsibility: monitor nodes, decide when they are faulty, and remove them. But this isn't a very robust approach: human errors can be a significant source of disruption in computer systems [7]. Therefore we propose instead to use a *configuration management* (CM) system that probes nodes periodically and removes those that don't respond properly; a way of accomplishing removal is discussed in [9]. The exact form of the CM is not of concern here; it could run at a single highly secure node, or at a (BFT) replica group that is either separate from the replicas being monitored or that runs on some subset of these replicas.

Now we can describe our approach.

We assume the hardware used in these nodes is tamper-resistant: The processor has a private key that changes in the case of an attack on the hardware [5]. We also assume proactive recovery but we augment it in one respect: when the node restarts, a fingerprint of the software that started when the node rebooted is stored in a special register of the secure coprocessor. This change will allow us to detect a corrupted code copy. This fingerprint should be computed over all modules used in the boot process (e.g., BIOS, boot loader, operating system) and also over the service code that is started at reboot.

The CM probes all nodes periodically. Each probe contains a nonce. The node sends a reply containing the nonce and the stored code fingerprint, signed by the node's key.

There are three possible responses to a probe:

- A valid reply. This contains a valid signature for the nonce and the correct code fingerprint. In this case the CM assumes the node is functioning properly.
- An error reply. This is signed with the expected key and contains the nonce, but it contains an incorrect fingerprint. In this case, the node has been compromised and the CM immediately removes it from service, causing some other non-compromised node to take over.
- An invalid or missing reply. This covers the case of a reply that is not signed correctly, or that doesn't contain the nonce. One missing or invalid reply means nothing. But if the CM gets this kind of response after repeated probes over a long period, it will conclude the node has failed and evict it from the system. Note that it is important for the system to be slow to remove a node due to invalid or missing replies because otherwise we open an avenue of attack: a denial-of-service against good nodes could cause the replies from these nodes to take longer, and then the nodes would be evicted and the fraction of compromised nodes in the system would increase.

Note that the scheme permits software upgrades. The CM must be told the fingerprint of the new code in advance and would allow nodes running either the old or new code. It could monitor upgrade progress and enforce deadlines, e.g., require all nodes to upgrade within some time period.

The use of an outside entity that does probing seems necessary. Nodes cannot monitor themselves and cause themselves to be replaced when necessary: a node that is dead, or that has been tampered with, cannot be counted on to notice that it is bad. Clients also cannot monitor nodes. Clients can notice error responses and bring them to the attention of an authority that would remove that node. But we also need to handle missing and invalid responses, and clients can't do this: clients can be corrupted too, and therefore their information about missing and invalid responses cannot be trusted. (Client information about valid and error responses can be trusted since they must produce the signed response.) A final point is that attestation is expensive and not doing it on every client request is a win. Probes are relatively infrequent and therefore the cost of signing them is small compared to the cost of useful work being done by the node.

Our approach protects us against many of the possible attacks. It allows us to remove failed or corrupted nodes that are unable to respond to probe requests. It also allows us to remove corrupted nodes that can respond to probe requests, but where a physical attack has corrupted the node or changed the code that is reloaded at a restart. In essence it supports the assumption in proactive recovery that when a node restarts it is running correctly.

But the approach is can't handle all attacks. In particular there is still a window of vulnerability equal to the restart period; we can't recognize attacks that corrupt a node's code or date between restarts. Bad code could be detected by computing a fingerprint of the code that is actually running on the machine as part of a probe response. The problem here is that it isn't clear exactly what constitutes the code, since the runtime environment matters too (e.g., the stack) and an attack might leave the code unchanged but modify its runtime environment.

Finally, we need to guard against an attack that modified the data (e.g., files that are part of the service state stored on disk) but leaves the code unchanged. The BFT library already contains a checkpointing mechanism whereby all nodes periodically exchange certificates of what they believe the correct fingerprint of the service state should be. This fingerprint is computed by maintaining an in-memory data structure with the hashes of all the objects (e.g., files and directories) that form the service state. So all we need to do is ensure that the data on disk matches the hashes used to compute the service state fingerprint. This can be done with a background process that reads data from disk and checks it against the expected hashes. When this process detects an incoherence, it could trigger state transfer, and/or warn the CM that an error in the data has occurred.

References

[1] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Feb. 1999.

[2] M. Castro and B. Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, Oct. 2000.

[3] B. Chen and R. Morris. Certifying program execution with secure processors. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, May 2003.

[4] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles*, October 2003.

[5] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Controlled physical random functions. In *Proceedings of the 18th Annual Computer Security Conference*, Dec. 2002.

[6] J. Howard and T. Longstaff. A common language for computer security incidents. Sandia Report SAND98-8667, Oct. 1998.

[7] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *Proc. 4th USITS*, Mar. 2003.

[8] M. Reiter. The Rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems*, pages 99–110, 1995.

[9] R. Rodrigues and B. Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. Technical Report MIT CSAIL TR/932, Massachusetts Institute of Technology, Dec. 2003.