A Replication- and Checkpoint-Based Approach for Anomaly-Based Intrusion Detection and Recovery

Adnan Agbaria*
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
adnan@crhc.uiuc.edu

Roy Friedman
Computer Science Department
Technion - Israel Institute of Technology
Haifa 32000, Israel
roy@cs.technion.ac.il

Abstract

The common approach to detecting anomaly-based intrusion is by replicating the computation and running a Byzantine agreement protocol among all replicas. However, Byzantine agreement incurs high communication overhead and also requires the use of more than $2t$ replicas (in fact, more than $3t$ replicas in asynchronous systems) in order to overcome t such failures. However, for many applications, and in particular scientific computation, it is possible to achieve the same goal with much lower average communication and replication overheads. This paper presents a new approach for detecting an intrusion by combining checkpoint/restart with replication. The main benefit of the approach is that we replicate the execution into only $t+1$ replicas, and invoke a Byzantine agreement only if we suspect an anomalous behavior that could be observed using checkpointing techniques. If a failure occurs, it is detected using any Byzantine agreement protocol that can agree on a recent valid system's state. Such a Byzantine agreement protocol also identifies the compromised nodes and eliminates them, so the computation can proceed with only $t+1$ replicas until the next failure occurs.

1. Introduction

Anomaly-based intrusions are defined as arbitrary deviations of a process from the behavior that is expected based on the algorithm it is supposed to be running and the inputs it receives. Such intrusions can occur due to a malicious attack. Detecting and recovering from intrusions is becoming increasingly important for several reasons. Increasingly, we depend on the accuracy and correctness of computations in daily life, and the damage caused by an intrusion can have a sever monetary cost. Also, as most computers are connected to the Internet, they are exposed to hackers, which increases the likelihood of malicious attacks. We agree with the opinion that preventing malicious attacks to intrude a system is an impossible task [9, 10], therefore, detecting and recovering intrusions could mitigate the effect of such attacks. Finally, recent initiatives, such as GRID computing and peer-to-peer computing, rely on the ability to run computations on foreign computers that cannot be trusted.

Replication is one of the key techniques for overcoming node failures. A typical way of obtaining such fault tolerance is by executing the application on several replicas while running agreement protocols either on the state of the system, or on the updates to the state [26]. The ability to use this approach thus largely depends on the ability to reach agreement among the replicas. It is known that reaching agreement in fully asynchronous environments is impossible even when it is assumed that there are only benign failures [13]. However, by adding some minimal synchrony assumptions, it is possible to solve these agreement problems even despite up to t Byzantine failures by running the protocol on at least $3t+1$ nodes [7, 11, 18].

Such an approach is required if each action is irrevocable, but is overly expensive in other settings, both in its communication costs and in the number of replicas [24]. There are many intrusion detection systems (IDSs) [16, 19] that are not based on replication. Upon detection, the corrupt process can be removed and may be replaced by another correct process. However, those IDSs are not always applicable either, since an intruder may be able to gain his/her goals before being detected and removed. Moreover, there are many experimental results showing that IDSs can detect only a small fraction of intrusions [25]. In addition, IDSs may produce false alarms that lead to removal of correct processes [22].

In particular, for scientific applications, it is possible to

*This material is based upon work supported by the National Science Foundation under Grant No. CNS-0406351.

simply run the computation on multiple nodes and choose the results that repeat more than a threshold number of times. A similar idea is used, e.g., by the SETI@HOME project [1]. However, such an approach is very wasteful, especially if we assume that most computations are failure-free. Moreover, corrupt nodes are detected only at the end of the computation, which further wastes system resources, and the mechanism relies on a single trusted entity to decide which computations are correct.

In this paper, we propose a new approach, which combines Byzantine agreement and checkpointing for introducing a new intrusion detection approach. Specifically, our approach allows detection and removal of malicious behavior, as is done in IDSs. However, our approach uses Byzantine agreement to agree on which nodes are corrupt, and also on a periodic valid state of the system, so that even if a node was malicious, it is possible to contain and eliminate the damage by rolling back the execution to the previous known valid state. In addition to the new detection mechanism that our approach proposes, the recovery is done to the latest valid state of a computation to reduce the amount of losing computation.

Our approach combines checkpoint/restart [3, 12, 23] with replication techniques, and allows a computation to detect intrusions in deterministic computations assuming operations are revocable. In our approach, each computation is carried out by only $t + 1$ independent replicas that must also periodically take a checkpoint of their state and store it in a globally accessible location. We separate the Byzantine agreement from the actual execution by having a different set of nodes, called *auditors*; the auditors are used to agree that all the $t + 1$ replicas (*workers*) have taken the same checkpoint. Otherwise, the auditors detect the correct checkpoint, eliminate the nodes that reported false checkpoints, and restart the computation from the last correct checkpoint. By doing this, we tolerate intrusions and remove such corrupt nodes for the sake of continuous correct computation and efficiency. Clearly, the auditors are shared by many computations; thus, assuming a large pool of nodes and computations, each failure-free computation requires only $t + 1$ nodes. Moreover, even faulty computations require extra nodes only during the phases of the computation in which the failures occurred. Another interesting feature of our approach is that we do not rely on assumptions such as "well-formed message" [17, 20] and "unstolen private keys" [24].

The remainder of this paper is organized as follows. Section 2 describes the system model and basic definitions. Our approach of anomaly-based detection and recovery is presented in Section 3. In Section 4, we describe previous related work, and conclude our work in Section 5.

2. System Model

We consider a distributed system consisting of two different sets of processes. The first set, called *auditors* and denoted by $\mathcal{A}$, consists of n processes, denoted by $A_1, A_2, \cdots, A_n$. The second set, called *workers* and denoted by $\mathcal{W}$, consists of m processes, denoted by $W_1, W_2, \cdots, W_m$. In this paper, we use the term "process" to refer to either an auditor or a worker when we do not distinguish between them in a given context.

Each process is modelled as an automaton with a predefined initial state, and a deterministic transition function from its current state to the next state based on the current state and the occurred *event*. The possible events are `send`, `receive`, and `computation`. In addition, a worker process can perform the `checkpoint` event. A *local history* of a process is a sequence of such events. An *execution* is a collection of local histories, one for each process. Processes also have access to a local clock, but this clock does not necessarily reflect the real time.

We assume that the model is *partially synchronous* as defined by Dwork et al. [11]. That is, processes communicate with each other by sending messages over a network. The system can behave in an asynchronous manner for a while in the sense that messages may take arbitrary length of time to arrive and might even be dropped during that period, and that the local clocks of processes may not be synchronized. However, there exists a *global stabilization time*, which is unknown to the processes, after which all messages arrive within a known delay Δ. The rate at which one process's clock can run faster than another process's clock is bounded by a known upper bound Φ. Notice here that for the communication among the auditors themselves, we could weaken the partial synchrony assumption and replace it with a mute failure detector, similar to the ones described in [17].

We assume that a process is either *correct* or *corrupt*, where a corrupt process is one that might completely halt its execution, or even deviate arbitrarily from its specification (due to an intrusion). A process is correct if it is not corrupt. We assume that corrupt processes cannot impersonate other processes (practically, this means that we have either private communication links or authentication at the network level). In addition, we assume that the number of corrupt auditors is bounded by f. Thus, by setting $n = 3f + 1$, the auditors can reach agreement despite f intrusions by running a Byzantine agreement protocol, such as [7, 11, 18].

We define a *computational task* to be a deterministic function and an associated set of input values for that function. The goal of the system is to compute the correct results of as many computational tasks as possible. The auditors send computational tasks to workers, which must compute the correct result of applying the function to the corresponding input values. However, completing a computational task may require multiple computational steps. In this work, we assume that a computational task can be completed without exchange of messages with other processes. Extending these ideas in an efficient manner to distributed applications, in which the same task is split among multiple processes that need to communicate among themselves in order to calculate the result, is left as an open question.

We assume that the number of corrupt workers is bounded by t (we do not assume any relation between t and f). Therefore, to ensure that a set of workers includes at least one correct worker, the auditors send the same computation task to $t+1$ workers (replicating the computation task). To reduce the attack propagation, we recommend that each worker run on a different physical machine [24, 28]. Please notice here that m could be more than $t+1$, but for every computation task we only need $t+1$ *active* workers. However, we will see later that for detection purposes, we may use up to t workers from the worker pool. Therefore, the total number of workers (active and non-active) is $2t+1$.

Along with replication, we use the checkpoint/restart technique for the anomaly-based intrusion detection. A *checkpoint* is the act of generating a file that includes a process state during the course of execution of an application. *Restart* is the act of restarting the process from a checkpointed state/file. We assume that the computation tasks are such that a task can be restarted from any checkpoint on any process in the system.

Each checkpoint taken by a worker is assigned a unique sequence number. The ith checkpoint of worker w_j is denoted by $C_{w_j,i}$. We use C_i to denote the set of the ith checkpoints from every worker, namely, $C_i = \{C_{w_1,i}, \cdots, C_{w_m,i}\}$. Moreover, the computation carried out by a worker w_j between $C_{w_j,i}$ and $C_{w_j,i+1}$ is referred to as the ith *checkpoint interval* of w_j.

3. Our Approach

We need to guarantee that the auditors expect the checkpoints from the correct workers every predefined period of time. Otherwise, a corrupt worker may send a checkpoint before the other workers, causing the auditors think that the correct workers are corrupt due to a timeout expiration. Therefore, we define a *common* checkpoint interval between the workers. The length of the common checkpoint interval, denoted by T seconds (the time unit is not important here), is known by the workers and the auditors. During initialization, the workers are asked to send their checkpoints to the auditors every T seconds.

On the other hand, we assume that the exact same sequence of computational steps is performed by each correct worker. In order to monitor the computation, after each predefined number of computational steps, each worker takes a checkpoint of its state, and sends this checkpoint to all auditors. We assume that checkpoints taken at different workers after the same number of computational steps are exactly the same[1].

By the system model, we assumed that the upper bound of the rate between every two different workers' clock is Φ. Therefore, to insure that the checkpoints of two different correct workers are the same, every correct worker takes a checkpoint every $\lfloor \frac{T}{d\Phi} \rfloor$ computational steps, where d is the time (in seconds) of a computation step in the fastest worker's machine. Then, every worker waits the elapsed of T seconds until it sends the checkpoint to the auditors.

For each checkpoint C_i, the auditors run a Byzantine agreement protocol in which they decide if all workers submitted their checkpoints and if all checkpoints are the same. If the answer is positive, the auditors commit checkpoint C_i, and the workers proceed with their computation. Otherwise, either the auditors did not receive all checkpoints, or some checkpoints did not agree. If $1 \leq l < t+1$ workers failed to send their checkpoints in time to enough auditors, those workers are considered corrupt, and the entire computational task, along with the previously committed checkpoint C_{i-1}, is sent to l new workers to continue from there. Otherwise, if some checkpoints did not match, the entire computational task, along with C_{i-1}, is sent to t additional workers. If that happens, after the new t workers submit their version of C_i, the auditors agree on which set of at least $t+1$ workers has submitted the same checkpoints. The corresponding checkpoint is committed as the official C_i. The workers that send different checkpoints are considered corrupt, and the auditors never send computational tasks to them again. The computational task is then resumed from C_i on $t+1$ unsuspected workers.

Note that for efficiency, the workers can initially send only a summary (e.g., MD5) of their checkpoint, and then the chosen checkpoint can be retrieved from one of the correct workers. The auditors then need to compute the summary of the checkpoint they retrieve from the worker to verify that it is as expected. For simplicity, we present the protocol without this obvious optimization. In addition, if a corrupt replica sends its checkpoint to a subset of the auditors, it will not affect the decision made by the auditors. Since the auditors run a Byzantine agreement to reach consensus, the decision remains correct as long as there are no more than t corrupt auditors.

3.1. The Protocol Details

Our protocol works as follows. Based on the common checkpoint interval T, whenever there is a checkpoint trigger, every active worker w takes a checkpoint $C_{w,i}$. Then, every active worker broadcasts the checkpoint to the audi-

[1]If the computation task involves invocation of a pseudo-random number generator, we assume that all workers apply the same seed in it.

tors. After every auditor A_i has received all the checkpoints from the active workers, the auditors agree on C_i for each worker w_j, $1 \le j \le t+1$. If all the checkpoints are valid and equivalent, then all the workers are believed to be correct.[2] Otherwise, the auditors try to identify the corrupt worker(s) in order to remove them from $\mathcal{W}$. In order to find the corrupt workers, the auditors contact new workers from $\mathcal{W}$ in order to *replay* the last *checkpoint interval*, which allows them to identify the corrupt processes.

The protocol is activated upon a new checkpoint taken by active workers. During the checkpoint request, the following steps are performed.

1. Each active worker w_i takes a temporary checkpoint $C'_{w_i,j}$ as described in the previous section and then broadcasts it to the auditors $\mathcal{A}$.

2. Whenever an auditor A_j receives a checkpoint $C_{w_i,j}$ (after T) from an active worker w_i, it sets a timeout equal to $\Delta + d\Phi$ for expecting the other t checkpoints.

3. The auditors perform Byzantine agreement on the received checkpoints, C_j. If they agree that the checkpoints are equivalent, then the checkpoints are assumed to be from correct processes. Then, this C_j is committed to be valid.

4. Otherwise, the auditors suspect that there is at least one corrupt worker in $\mathcal{W}$. Then, they try to detect the corrupt worker by applying the following steps:

   (a) The auditors send the computation task to another t workers asking them to restart the task from the last committed checkpoint C_{j-1}.

   (b) Each replica A_i collects the checkpoint from the new t workers.

   (c) The auditors agree on the correct checkpoint file C_j. Since they have input from different $2t+1$ workers, they can agree on the set of identical (equivalent) $t+1$ checkpoints.

   (d) The auditors then agree on the corrupt active workers, denoted by F, by comparing C_j to $C'_{w_i,j}$ for all $1 \le i \le t+1$.

5. The auditors stop the corrupt active workers F.

6. C_j is committed as the correct checkpoint file.

7. $|F|$ workers are restored in place of the corrupt ones (on different nodes) from C_j.

[2]Note that a worker might continue to generate good results for some time after being corrupt. As long as such a worker reports correct results, there is neither a need to exclude it, nor a way to detect that it was corrupt.

Figure 1 presents the pseudo-code of the main function of the protocol. This function is called by every auditor. In fact, it is the only thing that the auditors do in the context of our protocol. For simplicity of the presentation, in this figure we ignore the dealing with the common checkpoint interval (T).

```
verifyW ()
1: repeat
2:         receive($C_{w_i,j}$)  {Receiving the $j$th checkpoint}
3:         if $C_j == \emptyset$  {This is the first element of $C_j$}
4:            timeout($\Delta + d\Phi$)  {Set the timeout}
5:         if $C_{w_i,j} \notin C_j$,  $C_j = C_j \cup C_{w_i,j}$
6: until (timeout or ($|C_j| == t+1$))
7: if (($|C_j| == t+1$) and (ByzAgreement($C_j$)==true))
   {all the auditors decided that $C_{w_i,j}$-s are equivalent}
8:    commit($C_j$)
9:    return  {There are no suspected corrupt processes in $\mathcal{W}$ }
 {$A_i$ suspects that some processes in $\mathcal{W}$ are corrupt }
10: $F$ = detectFaulty($\mathcal{W}$)
11: $\mathcal{W} = \mathcal{W} \setminus F$  {Remove the corrupt workers}
12: $N$ =active($|F|$)  {Activate another $|F|$ workers}
13: $\mathcal{W} = \mathcal{W} \cup N$  {Add the new active workers}
14: restart($\mathcal{W}, C_j$)
```

Figure 1. The auditor A_i verifies $\mathcal{W}$

As depicted in Figure 1, the function **verifyW** uses several functions. Below we explain each function and present the pseudo-code of some functions for illustration.

- **ByzAgreement** - Upon invoking the function **ByzAgreement**(v), the auditors apply any consensus protocol (e.g., [17, 21]) to agree on the value of v. This function returns **true** only if an agreement is reached.

- **commit** - This function commits the newest checkpoint files of $\mathcal{W}$.

- **detectFaulty** - This function returns the set of the corrupt workers as described before. Please notice here that the auditors invoke Byzantine agreement to agree on the set of corrupt workers.

Notice that after detecting the corrupt workers, we can restart the computation task using a new $\mathcal{W}$ from the last committed checkpoint on a different machine to reduce the risk of malicious attacks. In addition, the add/remove operation of workers is done upon agreement among the auditors. Technically, by removing a worker from $\mathcal{W}$, the auditors ignore its messages. On the other hand, by adding a worker to $\mathcal{W}$, the auditor start to consider its messages and inputs.

Figure 2 presents an execution example for which $t = 1$. In the first checkpoint C_1, we assume that there are no corrupt workers, where the auditors reach an agreement and do not interfere with the workers' executions. However, we assume that before taking the checkpoint C_2, one of the workers was corrupt. Thus, after taking C_2, the auditors suspect that there is a corrupt worker. By the protocol, a new worker is activated to take another copy of C_2. Then, the auditors can detect the corrupt worker and ask the new correct workers to take over.

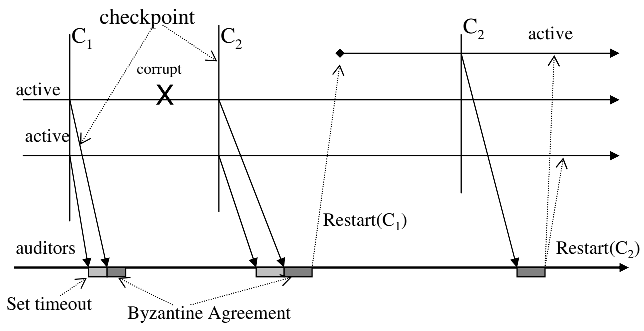

Figure 2. An execution example

In fact, the rate of successful malicious attacks is relatively slow [27]. So, we can apply the following optimization. The workers can keep running while the auditors try to detect any corrupt worker. This way, the workers need not wait for the auditors, and the two tasks of computation and verification can be pipelined.

3.2. The Protocol Properties

In this section, we prove the correctness of the protocol. In order to do so, we slightly refine the definitions of correct and corrupt process to match the structure of our protocol. Specifically, a checkpoint generated by a correct process is called a *valid checkpoint*. A node is considered *correct during a checkpoint interval* if the checkpoint it generated during this checkpoint interval is the same as the corresponding valid checkpoint for that interval.[3] A node is said to *behave in an observable corrupt manner during a checkpoint interval* if it reported a checkpoint that is different from the corresponding valid checkpoint for that interval to at least $f+1$ auditors. Based on these definitions, we show that for each checkpoint interval, the auditors identify the valid checkpoint for this interval. Based on this, the auditors can identify all workers that behaved in an observable faulty manner during the interval, and all workers that were correct during that interval.

[3]Note that since we assumed that the computation is the same in all replicas, all valid checkpoints for the same interval are identical.

Lemma 3.1: For each checkpoint interval, the auditors can identify the corresponding valid checkpoint.

Proof sketch: Since we have $t + 1$ workers running a computation task, there is always at least one correct worker running. If during a checkpoint interval some workers behave in an observable corrupt manner, then at least $f + 1$ auditors will receive checkpoints different from the valid checkpoint. Since at most f auditors can be corrupt, at least one correct auditor should receive the different checkpoint. Therefore, the correct auditors cannot reach a consensus on the valid checkpoint (as depicted in Line 7 of Figure 1).

If they are unable to reach a consensus, by the protocol, the auditors will ask up to an additional t workers to execute this interval as well, for a total of $2t+1$ workers. As at most t can be corrupt, at least $t + 1$ checkpoints reported to the workers will be the same. Each worker will pick this checkpoint as the valid checkpoint. Moreover, as $t < (2t + 1)/2$, the chosen checkpoint file for each correct worker must be the same, and must indeed be a valid checkpoint. □

Lemma 3.2: Every correct auditor identifies every worker that behaves in an observable corrupt manner in any checkpoint interval.

Proof sketch: By Lemma 3.1, each correct auditor can detect the valid checkpoint for any checkpoint interval. Thus, by comparing the checkpoints reported by all workers to a valid checkpoint, a correct auditor can identify the workers that behaved in an observable faulty manner. Moreover, as there are $3f + 1$ auditors, and as at most f of them can be corrupt, any of the known Byzantine consensus protocols, such as [17, 21], always terminate and have all correct auditors agree on the same set of corrupt workers. Moreover, these protocols ensure that if all correct processes propose the same value, then this value will be decided on. Thus, in our case, as all correct auditors suspect the same workers, these workers will indeed be identified as corrupt, and removed from the set of workers. □

Lemma 3.3: A worker that is correct during a checkpoint interval is not suspected of being corrupt by any correct auditor during that interval.

Proof sketch: The proof of this lemma follows the same arguments as the proof of Lemma 3.2. □

4. Related Work

There are many work of anomaly-based detection that based on Byzantine agreement with at least $3t + 1$ replicas [17, 21, 24] as well as intrusion detection systems [16, 19]. By the best of our knowledge, our work is the first that requires only $t + 1$ replicas when there are no intrusions. However, when an intrusion occurs due to a malicious attack, our scheme employs more processes as worse as the other techniques. However, in terms of the number of hosts, our approach does not exceed the traditional approaches if both a worker process and an auditor process run simultaneously on the same host. Moreover, when the system executes multiple computational tasks, the same set of auditors can be used for all computational tasks, thereby amortizing their cost.

In [17] Kihlstrom et al. extended the work of Chandra and Toueg [8] on unreliable fault detectors for crash faults by considering unreliable fault detectors for Byzantine faults. They used those detectors to reach consensus in an asynchronous distributed system. Similarly, Malkhi and Reiter [20] solved the consensus problem in asynchronous distributed systems, but unlike the algorithm of Kihlstrom et al., their consensus algorithm relies on a reliable broadcast service. As in [17, 20] and other papers, their algorithms rely on $3t + 1$ replicas. Other papers that solve Byzantine agreement with unreliable failure detectors that can only detect mute failures include [5] and [14].

In [24] Ramasamy et al. extended the Ensemble group communication toolkit [15] to support intrusion tolerance. Originally, Ensemble was designed for crash fault tolerance; Ramasamy et al. re-implemented the reliable multicast and the group membership protocols to support intrusion tolerance that incorporates intrusion detection by $3t+1$ replicas.

There was earlier work on combining checkpoint/restart and replication for overcoming crash failures. For example, the Starfish system [4] uses such a combination to provide crash fault tolerance for both serial and message-passing applications.

Several papers used checkpoint/restart and duplication for detecting transient faults. Ziv and Bruck [30] adapted a transient fault-detection mechanism by taking a checkpoint of two replicas and then comparing their states. In addition, Black et al. [6] presented a technique dealing with fail-silent processes in the Voltan application programming environment using self-checking process pairs. Their work was based on hardware-based replication.

With respect to use of Byzantine agreement as a black box for other computation, Yen et al. [29] separated the Byzantine agreement from the execution in order to mask t Byzantine faults using $3t + 1$ auditors, while the actual execution can be accomplished using $2t + 1$ replicas. This separation of the execution from the agreement is similar to our approach. However, we employ only $t + 1$ processes in the fail-free case, and also apply Byzantine agreement as an intrusion detection and elimination mechanism in order to reduce the continuous effect of such faults.

Unlike our work, most intrusion detection protocols focus on detecting an intrusions, but not on removing it. We believe that removal of a corrupt process is very important, since a successful malicious attack can propagate to the other processes in the system very quickly [25], to the point that tolerating such attacks could be impossible according to such protocols. Our detection mechanism is incorporated into the intrusion tolerance mechanism, yet does not depend on third-party techniques [2].

5. Conclusions

We have presented a scheme that allows deterministic computations to overcome up to t intruders using $t + 1$ active replicas and $3t + 1$ auditors. The protocol is based on periodic checkpoints, and assumes that it is permissible to roll back the execution to a previous checkpoint. In a nutshell, our approach combines checkpointing and replication in order to detect and then remove an intrusion. In addition, we showed how our work can be used for collaborative applications among several computing processes that are not replicas of the same program.

Considering the fact that intrusions occur rarely relative to the time it takes to perform the computation, our approach is designed to behave efficiently in the intrusion-free case. Therefore, we only need $t+1$ replicas to carry out the main computation. In addition, once a checkpoint has been taken by a replica, the replica continues its computation task immediately, and need not worry about intrusion detection and recovery issues. It can thus be simple.

Practically, our approach can be viewed as both a Byzantine fault-tolerant protocol and an intrusion detection mechanism. For the former, due to replication and Byzantine agreement, the main computation task can proceed correctly despite up to t Byzantine failures. For the latter, our mechanism detects any faulty process (or intrusion) that violates the specification of the problem, and prevents the faulty nodes from handling future computation tasks.

Acknowledgments

We would like to thank Jenny Applequist for her editorial assistance.

References

[1] SETI@home Home Page. http://setiathome.ssl.berkeley.edu.

[2] The Snort Home Page. http://www.snort.org.

[3] A. Agbaria, H. Attiya, R. Friedman, and R. Vitenberg. Quantifying Rollback Propagation in Distributed Checkpointing. In *Proceedings of the 20th Symposium on Reliable Distributed Systems (SRDS'01)*, pages 36–45, New Orleans, USA, October 2001.

[4] A. Agbaria and R. Friedman. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 6(3):227–236, July 2003.

[5] R. Baldoni and J. M. Helary. Strengthening Distributed Protocols to Handle Tougher Failures. Technical Report #1477, IRISA, Université de Rennes, France, 2002.

[6] D. Black, C. Low, and S. K. Shrivastava. The Voltan Application Programming Environment for Fail-silent Processes. In *Proceedings of the Distributed Systems Engineering*, pages 66–77, June 1998.

[7] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, February 1999.

[8] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.

[9] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet Security, Second Edition: Repelling the Wily Hacker*. Boston: Addison-Wesley, 2003.

[10] Y. Deswarte, L. Blain, and J. C. Fabre. Intrusion Tolerance in Distributed Computing Systems. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 110–121, May 1991.

[11] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, April 1988.

[12] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-99-148, Department of Computer Science, Carnegie Mellon University, June 1999.

[13] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):372–382, April 1985.

[14] R. Friedman, A. Mostefaoui, and M. Raynal. Simple and Efficient Oracle-Based Consensus Protocols for Asynchronous Systems. Technical Report 1556, IRISA - Institute de Recherche en Informatique et Systemes Aleatoires, September 2003.

[15] M. Hayden. *The Ensemble System*. PhD thesis, Department of Computer Science, Cornell University, January 1998. Also published as Cornell Dept. of CS Technical Report no. TR98-1662.

[16] H. S. Javitz and A. Valdes. The SRI IDES Statistical Anomaly Detector. In *Proceedings of the IEEE Conference on Research in Security and Privacy*, pages 316–376, Oakland, CA, May 1991.

[17] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Solving Consensus in a Byzantine Environment Using an Unreliable Fault Detector. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, pages 61–75, December 1997.

[18] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[19] Ulf Lindqvist. *On the Fundamentals of Analysis and Detection of Computer Misuse*. PhD thesis, Department of Computer Engineering, Chalmers University of Technology, Goteborg, Sweden, 1999.

[20] D. Malkhi and M. Reiter. Unreliable Intrusion Detection in Distributed Computations. In *Proceedings of the 10th Computer Security Foundations Workshop (CSFW97)*, pages 116–124, Rockport, MA, June 1997.

[21] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *The Journal of Distributed Computing*, 11(4):203–213, 1998.

[22] D. Newman, J. Snyder, and R. Thayer. Crying Wolf: False Alarms Hide Attacks. In *Network World*. Network World, Inc., June 2002.

[23] J. S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee, July 1997.

[24] H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders. Quantifying the Cost of Providing Intrusion Tolerance in Group Communication Systems. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 229–238, Washington, DC, June 2002.

[25] W. H. Sanders. CDR Validation Report. Technical Report CDRL A007-R2, BBN Tehnologies, 2003.

[26] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[27] F. Stevens, T. Courtney, S. Singh, A. Agbaria, W. H. Sanders, J. F. Meyer, and P. Pal. Model-Based Validation of an Intrusion-Tolerant Information System. In *Proceedings of the 23rd Symposium on Reliable Distributed Systems (SRDS 2004)*, pages 184–194, Florianöpolis, Brazil, October 2004.

[28] T. Wu, M. Malkin, and D. Boneh. Building Intrusion-Tolerant Applications. In *Proceedings of the 8th USENIX Security Symposium*, pages 79–91, 1999.

[29] J. Yin, J-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement from Execution in Byzantine Fault-Tolerant Services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 253–268, October 2003.

[30] A. Ziv and J. Bruck. Efficient Checkpointing Over Local Area Networks. In *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 30–35, June 1994.