A Fault-Tolerant Network Intrusion Detection System

Tiago Sampaio de Faria Picado

Departamento de Engenharia Informática,

INSTITUTO SUPERIOR TÉCNICO

tiago.picado@ist.utl.pt

http://dei.ist.utl.pt

Abstract. Network intrusion detection systems can play a determinant role in providing adequate security to an organization, and so it is important that those systems are given the chance to analyze as much of the available data as possible. However, it is known that under certain conditions, in particular when under heavy load, they may lose some network packets. In this paper, the possibility of minimizing intrusion detection systems omission failures by using multiple replicas of an intrusion detection system was researched. The underlying idea is the fact that the probability of loss of a given packet by the set of replicas should be lower than the probability of loss of that packet by a system composed by a single intrusion detection instance. For this, a synchronization layer, sitting between the packet capture layer and the intrusion detection layer was developed. While the synchronization protocol is usually required precisely when conditions are more demanding and the ability to recover packets may for that reason be limited, results suggest that at least on some scenarios, such as in the occurrence of traffic bursts or of short processing overloads, the synchronization mechanism could prove to be useful.

Keywords: Network intrusion detection system, omission failures, replication, synchronization protocol, diff

1 INTRODUCTION

From country-sized corporations to home and mobile users, today's world is increasingly reliant on information systems and on the communications networks which connect them. The Internet in particular, and its related set of technologies have become nothing short of ubiquitous, and increasing convergence between Information Technology and Telecommunications worlds is taking this reality even further. Hand in hand with this usage growth, came an increase in the number of attacks to those systems and networks, making protection from attacks increasingly important. Intrusion detection systems (IDSs) can play an important role in the defense arsenal.

To fulfill this role, it is important that they are provided with the chance to analyze as much of the available data as possible. However, network intrusion detection systems (NIDSs) are known to miss packets is some situations, namely in those situations where they are unable to process with sufficient speed the traffic stream that is presented to them for analysis. Even if the rate of missed packets is low, packets that were lost could prove to be important in detecting a relevant intrusion or intrusion attempt.

In this paper, the possibility of minimizing NIDS omission failures by using multiple synchronized replicas of an IDS was researched. The underlying idea is the fact that the probability of loss of a given packet by the set of replicas should be lower than the probability of loss of that packet by a system composed by a single IDS instance. For this purpose, the popular *open-source* Snort [1] IPS/IDS was used. The goal was to develop a synchronization protocol able to detect misaligned replicas in regard to the stream of packets each of them received from the network for analysis, to determine which specific packets were different across replicas, and to recover required packets, so that all the replicas could analyze the same set of packets. For this, a synchronization layer sitting between the network traffic capture library and the IDS was developed.

2 BACKGROUND

2.1 Intrusion Detection Systems

When guarding computer systems or networks against attacks, the conventional approach is to deploy a number of protective mechanisms in order to secure them. However, this approach has some limitations [2]: it is difficult to build systems which are absolutely secure; it may be impractical to replace a vast existing and possibly insecure infrastructure in favor of a new one; the prevention-based approach constrains user's activities, making them less productive; crypto-based systems cannot defend against lost or stolen keys or passwords; and secure systems can still be vulnerable to insiders. These limitations justify the use of other approaches. Intrusion detection systems can provide a second line of defense, by [3] enabling early detection of intrusion activities, dissuading intruder's intentions or enabling the collection of information about intrusion techniques, that can be used to strengthen the prevention facilities.

IDSs may be classified according to many characteristics. The most used ones are their detection principle and audit data source. Regarding the first, two main types of IDSs exist: those who try to detect a deviation from the normal usage pattern of a system (*anomaly*, *behavior-based* or *statistical* systems), and those who use knowledge about attacks and try to find the presence of those attacks (*misuse*, *knowledge-based* or *signature-based* systems). Regarding the second, there are also two main types: if the analyzed audit data refers to events on a single host, an IDS is said to be a *Host based IDS* or *HIDS*. If the analyzed audit data is gathered from traffic collected on a network, the IDS is said to be a *Network based IDS* or *NIDS*.

2.2 Snort

Due to its popularity, feature set and source code availability, the Snort Intrusion Prevention and Detection System (IPS/IDS) became a natural choice for this project. The heart of Snort as an IDS is its signature-based detection engine, which compares network packets against a set of rules, looking for matches (IP addresses, ports, packet headers, payloads, etc.), but the usage of packet preprocessors for traffic normalization is essential to avoid techniques for evading the IDS itself, such as those that were described in a seminal paper by Ptacek and Newsham [4].

One of the aspects which make Snort appropriate for usage in a research project is its architecture. The software uses a layered architecture, which in some layers is plug-in based. Layers of the IDS include the packet capturing layer, the packet decoding layer, the packet preprocessors, the detection engine and the output plugins. The current stable implementation of Snort can be regarded, for the most part, as single-threaded.

2.3 Fault Tolerance

Fault tolerance is an essential requirement for achieving dependable systems. The original definition of dependability is the ability to deliver service that can justifiably be trusted [5], which in the end is what one intends to obtain. As mentioned in the beginning of this paper, the world is increasingly reliant on information systems and communications networks, and so the need for dependable systems is generally increasing. One possibility for achieving fault-tolerant services is to build software on top of fault-tolerant hardware. However, economic factors have motivated the development of cheaper software-based fault-tolerance solutions, and in particular of solutions based on replication.

Two classical approaches for replication exist: the *primary-backup replication* model and the *active replication* model. In the *primary-backup*, or *passive replication* model, in its purest form, one replica is selected as the primary replica for the service, meaning that this replica receives requests from clients, processes the requests, responds to clients and propagates changes to the backup replicas. If the primary replica fails, one of the backup replicas is promoted to the role of primary. In the *active replication*, or *state-machine* model, replicas are seen as state machines and all of them play equivalent roles. Each replica processes the same requests from clients as the others, and in the same order, and so their state remains synchronized. If one replica fails, there is no availability problem, as others can keep serving requests. Active replication relies on a communication primitive which guarantees required properties of order and atomicity, which is called *total order broadcast* (also known as *atomic broadcast*; see [6]).

The replication scheme implemented in this work bears some resemblance with the active replication model, as in both cases replicas behave as state machines. However, replicas in the proposed system do not use an atomic broadcast primitive, but instead rely on the underlying hardware to ensure ordering, and are required to deal with omission failures.

2.4 The State Saving/Transferring Problem

One issue that would need to be solved in a completely working system would be to have the ability to add new replicas to a running cluster (e.g. to recover from crashes). To align the state of a newly launched replica without processing all previous packets, the state of one of the other replicas could be transferred. One solution is to use a process checkpointing package, also known as process checkpoint/restart, or process C/R, which creates a process image that can be restarted. Due to its features, actuality and maintenance, the stand-alone MTCP component of the Distributed MultiThreaded CheckPointing (DMTCP) project [7] was used in this work.

3 DESIGN

The proposed fault-tolerant system was designed considering the common setup of an intrusion detection system which listens to only one network interface, from which it receives packets to be analyzed. It is assumed that this interface is not used for any purposes other than traffic analysis, and in particular that the system does not send out traffic generated by itself, through that interface. In such a setup, the single IDS system can be replaced by the fault-tolerant one, composed by a set of replicas, each running a synchronization layer which in turn feeds its IDS instance. All replicas should receive the same network packets, sent by some origin interface, and duplicated by some device which behaves as a multiport repeater, such as an Ethernet hub. A second, separate interface is used by each replica for the required system synchronization tasks, and any other functions, such as system control, monitoring or reporting.

3.1 Packet Synchronization Mechanism

Considering that all network components behave as FIFO's, then all replicas synchronization layers should receive the same set of packets, in the same order, unless some component is unable to deliver one or more of them – an omission failure. A common reason for such a failure is that the consumer component is unable to pick up and process packets quickly enough, leading to the filling of buffers of other components, causing them to drop subsequent packets.

In a system composed by multiple replicas, each receiving and processing the same network traffic, omissions, if they occur, will not necessarily happen in the same way on all replicas, due to differences in the load and pace of independent, asynchronous systems, even if the main pace-setting factor, the stream of packets, is the same for all of them. The proposed system attempts to take advantage of this by combining the data received by each of the replicas and trying to detect if packets were lost by one or more of them. If it is able to detect that packet loss has occurred, it tries to infer, as best as it can, what the original packet stream must have been.

To achieve this, a compact representation of each received packet is computed and stored, using a hash function. For every *k* packets, a compact representation of the corresponding set of hashes is computed in the same way, and then exchanged and compared by all replicas. If the system is running in the absence of faults, all replicas are able to receive and process the same packets. Since the hash function is deterministic, the computed hashes will be the same in all replicas, and so will be the hash of a given cluster of *k* hashes. This will also hold true if all replicas miss exactly the same sub-set of original packets, but in this case the fault cannot be detected. If, on the other hand, omissions occur which do not involve exactly the same sub-set of packets, the hash of *k* hashes will differ between replicas, and the fault becomes apparent.

Once detected, replicas exchange the hashes of individual packets and run a deterministic algorithm to determine which packets are common, which are different, and how should the differences be merged. While such algorithm could have been developed during this work, by viewing the sequences of hashes as lines of text files, this task can be left to the `diff` utility.

3.1.1 “Anchor” Definition

An “anchor” is defined as a block of a chosen size of *l* contiguous packets which are present in all replicas. If such a block exists, the replicas are considered to be aligned in that block of *l* packets, and `diff` can be used to determine the best way to merge all hashes of packets which precede it. If such a block does not exist, on the other hand, then replicas are considered to be misaligned in such a way that using `diff` to merge the hashes is not valid, and further *k* packets need to be considered before attempting to realign the replicas. If multiple “anchors” exist, then the one used for alignment is the one closest to the end of the block being evaluated.

Once misaligned, it is necessary to find a new synchronization point, which is equivalent to finding an “anchor”, and to transfer missing packets prior to that “anchor” between replicas, before restarting the process of comparing only the hashes of hashes. Packet hashes could be added one by one, until an “anchor” was found, and comparing the hashes of hashes could restart after that. In the current implementation, however, the synchronization mechanism always moves in blocks of *k* packets. If the first misaligned block of *k* packets cannot be aligned, a second block is added, and a level 1 desynchronization is said to occur. A level 1 desynchronization is equivalent to choosing a size of *2k* for the block of hashes.

3.1.2 Bootstrapping

Bootstrapping the system is performed if the following manner: the first replica is launched as the leader replica. Before starting to process packets, it waits until a specified number of peers have connected to it, or a specified amount of time has elapsed. Each connecting peer is required to start capturing packets before connecting to the leader. This ensures that, once the leader starts collecting packets, other replicas participating in the system bootstrap have at least as many packets as the leader replica. Once the leader starts the boot process, it first collects a specified number of packets, say *m* packets, and then sends a list of all corresponding hashes to the other peers. Each peer attempts to find a sequence of hashes matching the received list, starting from the last *m* hashes in its own list. If the match exists, it is considered to be aligned with the leader up to the last hash of the received list, and the process of comparing hashes of blocks of *k* hashes starts with the first packet captured after that.

3.2 Optimistic Modes

The synchronization layer writes captured or resynchronized packets onto a shared memory segment. The intrusion detection system may consume these packets only up to where they are confirmed to be aligned: This is called optimistic mode 1 or “pessimistic mode”. It may also consume packets in the current block, whose alignment is yet to be confirmed. This is called optimistic mode 2 or “optimistic mode”. This mode requires a checkpoint and rollback mechanism, so that the IDS state can be returned to a point where only aligned packets were consumed, if it is discovered that it (optimistically) consumed misaligned packets.

3.3 The Synchronization Algorythm

The proposed algorithm is implemented as a recursive function, which on each call executes a given synchronization round, that is, it uses the next block of *k available* packets to test for replica alignment and to realign replicas if packet loss has occurred and it is possible to partially or completely execute that task. Figure 1 presents the algorithm pseudo-code:

```
forever do
   cluster_num = cluster_num + SyncWithPeers(cluster_num, 0, 0);


int SyncWithPeers(cluster_num, desynced, credit)
   synced = 1;

   if (credit < CLUSTER_SIZE)
      EnqueuePackets(CLUSTER_SIZE - credit);
      credit = CLUSTER_SIZE;

   UnlockPeersWaitingClusterHash();

   if (PeersSendClusterHash(cluster_num) || desynced)       // replicas not synchronized
      MyHashesSet(cluster_num, desynced);         // unlocks peers waiting to get my hashes
      all_hashes = AllHashesGet(cluster_num, desynced);       // gets hashes of peers

      anchor = DiffRun(all_hashes);       // tries to find an "anchor"

      switch (anchor)
         case -1:       // local replica has not lost packets
            desynced = 0;
            SyncUpperLayer(OK);

         case -2:       // local replica lost packets; one other replica has not lost packets
            credit += EnqueueMissingPackets(CLUSTER_SIZE*(desynced+1) – ANCHOR_SIZE);
            desynced = 0;
            SyncUpperLayer(NOT_OK);

         case -3:       // no "anchor" found
            desynced = desynced + 1;
            SyncUpperLayer(UNKNOWN);

         default case:       // "anchor" found at 'anchor'
            credit += EnqueueMissingPackets(anchor);
            desynced = desynced + 1;
            if ((desynced + 1 - CountResynchronizedBlocks()) > 0)
               desynced = desynced + 1 - CountResynchronizedBlocks();
            else
               desynced = 0;

            SyncUpperLayer([UNKNOWN|NOT_OK]);

   else       // replicas synchronized
      SyncUpperLayer(OK);

   if ((desynced > 0) || (credit > CLUSTER_SIZE))
      synced = synced + SyncWithPeers(cluster_num + 1, desynced, credit - CLUSTER_SIZE);

   return synced;
```

Figure 1 – Synchronization algorithm pseudo-code.

4 IMPLEMENTATION

The fault-tolerant system is comprised of a number of equally behaving replicas, with the exception of the system bootstrapping phase, which uses the first launched replica as a leader, as described earlier. Each replica is composed by two independent processes: one running the developed synchronization layer and another running the network traffic analysis layer. For the later, a modified version of the *open source* industry-standard IDS Snort was used.

It is important to notice that some elements which would be required to exist on a production system, but that were not considered fundamental for the purposes of this work, are absent in the current implementation, such as replica management, alert handling or supporting replica crashes. Also, currently, only two replicas are supported.

4.1 System Components

The major components of the system can be divided in two categories: processes/threads, which execute some task, and information repositories, which keep data that needs to be shared between threads. The following major components exist in the synchronization layer:

Main Thread: Starts and ends replica execution, launches the intrusion detection system, and monitors incoming alerts.
Curses Thread: Interactively presents to users the execution progress of a replica.
Accept Thread: Accepts incoming connections from peers and launches answer threads.
Answer Thread: Processes the requests made by the peer connected to the thread.
Sync Thread: Keeps replicas synchronized by exchanging cluster hashes, hashes of packets and packets, with peer replicas.
Pcap Thread: Captures packets from the network and enqueues them in shared memory.
Restart Thread: Restarts the IDS process when a roll back needs to be performed.
Alert Thread: Receives alerts generated by the IDS and filters out duplicates.
Packet Hashes Database: Stores the computed hashes of captured packets, as well as other important data, such as packet order information and the location of the actual packet data in the shared memory segment.
Connections Manager: Keeps track of peers present in the system, as well as of channels which are open between the local replica and the other peers.

The other major component of the system is of course the analysis layer, composed by Snort. For integration with the synchronization layer, a number of modifications were introduced to the original Snort intrusion detection system source code. The most significant modification was the replacement of the perpetual packet capture loop provided by the `InterfaceThread()` function, so that the IDS could read packets to be analyzed from the shared memory segment, instead of from LibPcap. This modified version also uses the MTCP standalone component of the Distributed MultiThreaded CheckPointing project, which also suffered a few modifications.

5 RESULTS AND DISCUSSION

In this chapter, an evaluation of the system effectiveness and efficiency is carried out. Two classes of hardware were used: a cluster of server-class computers, equipped with two quad-core processors, and a cluster of desktop-class computers, equipped with one dual core processor. Traffic was injected by a computer using two GigabitEthernet interfaces, connected to the capture interfaces of replicas, and using the Linux bonding driver, in broadcast mode. Two datasets were used: a sample of one million packets collected from a live Internet connection, and a sample of artificially crafted UDP packets. The former was used in the correctness effectiveness and performance tests; the later was used in the application scenarios.

5.1 Implementation Correctness

Tests showed that the algorithm could correctly align packets at different packet loss rates, as the resulting ordered files of hashes were equal between replicas. Different levels of packet loss, ranging from 0,05% to 50%, were artificially introduced by a RandomDrop() function.

5.2 Synchronization Protocol Effectiveness

Results showed that the number of packets lost were close to the probabilistically expected values, and that the total differences, considering disordered packets, were still lower than the theoretical differences for a single replica, as depicted in figure 2:

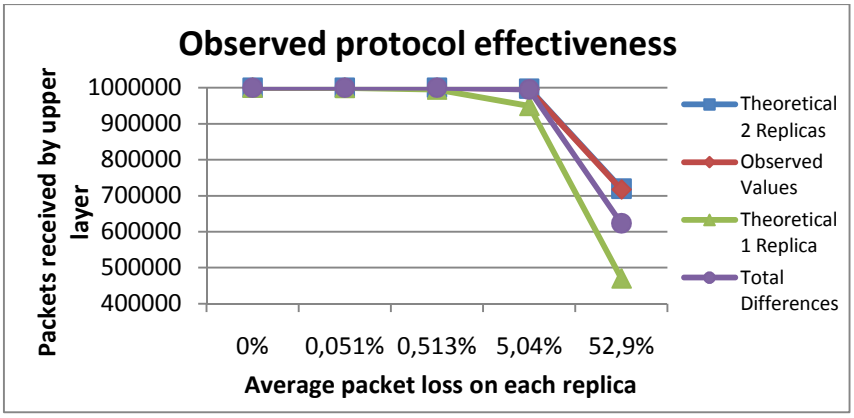

Figure 2 – Theoretical and observed synchronization protocol effectiveness, including data representing the total number of differences between hash files of synchronized replicas which have lost packets and a file from a lossless replica. The results shown are a three-run averages for each packet loss rate condition.

5.3 System Performance

A hash cluster size of 56 proved to be the best choice between values of 8, 56 and 1024 hashes (results not shown). Without packet loss, the system was able to handle speeds close to full capacity of the Gigabit link. With packet loss, the replicas would fail at different speeds, but before failing, they could sustain synchronization at the rate of injected traffic, either with or without running with the IDS. Figure 3 shows throughput of the system without running the IDS:

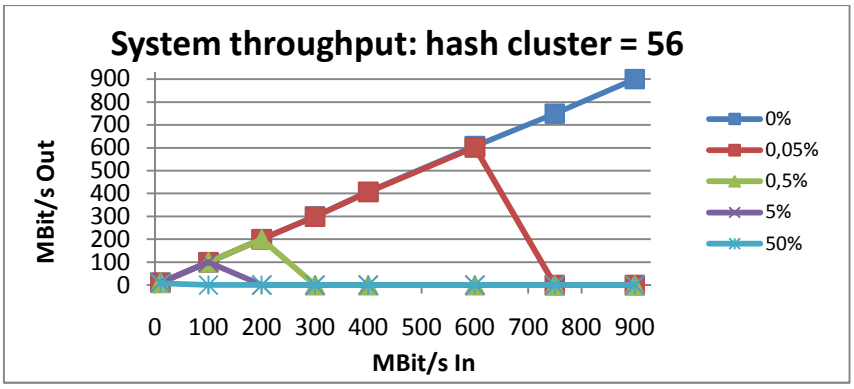

Figure 3 – Replica system throughput for the server-class hardware, running without the IDS, and a cluster size of 56, fount to be the best choice between 8, 56 and 1024.

5.4 Optimistic Mode

Results (not shown) indicate that the number of checkpoints and rollbacks performed by the system decreases with the increase of injected traffic rate, suggesting that at lower rates the IDS is in fact ahead of the last synchronization point, but optimistically feeding it becomes less relevant at higher traffic rates, as the IDS tends to lag behind the synchronization mechanism.

5.5 Application Scenarios

Two application scenarios were tested. Some conditions were found in both cases which suggest that the synchronization protocol could be useful, at least in some situations. Replicas were run with limited memory resources, so that packet loss occurred spontaneously.

5.5.1 Traffic Spikes

The first considered application scenario was the case of network traffic with the occurrence of spikes, where the system is unable to handle all traffic, resulting in packet loss. Since these spikes are followed by periods of much lower traffic rates, the system is given time to recover some of the packets each individual replica has lost, and to empty filled up buffers. In this test, using the server-class hardware, where a stand-alone IDS lost 1070 packets on average, each replica lost 1266 packet on average, but the IDS layer ended up missing just 552, due to packer recovery, while the stand-alone version of course missed all 1070.

5.5.2 Busy Systems

The second application scenario is the case where one system is subjected to a temporary high processor load. This load could be due to some process separate from the intrusion detection process, as there are usually at least a few additional processes running on system dedicated to intrusion detection, for example for system management. In this test, using the desktop-class hardware, a batch of 50 thousand packets was injected at a rate of about 50 Mbit/s, followed by the injection of 8 additional packets, to allow for synchronization round completion (56 x 893 = 5008). The replica with the busy process lost on average 385 packets, but was able to recover all of them from the other replica.

6 CONCLUSION

The loss of packets by a network intrusion detection system may compromise its ability to detect intrusions or intrusion attempts. Even if the number of packets missed by the intrusion detection system is very low, when compared with the number of packets that were analyzed, some dose of misfortune may dictate that the ones that were missed were the ones that were needed for detecting an attack that would cause significant harm to the organization.

In this project, an attempt was made for providing a solution to avoid or at least minimize these omission failures, by developing a system composed by a set of replicas that could together reduce or avoid packet loss by the system, based on the fact that the probability of loss of a given packet by all the replicas should be lower than the probability of loss of that packet by a system composed by a single replica. Results, using a system with two replicas, suggest that this approach could be feasible in practice, given the availability of spare processing capacity which is common in today's multi-core processor computers. These came from a number of performance tests made in conditions where the replicas always had sufficient memory and where packet loss was artificially caused.

Two typical application scenarios were also tested. Those application scenarios were the case of a network link with a usually low traffic rate, but in which an occasional traffic spike would occur, and the case where some process running in one replica would briefly require a large amount of processing resources. Results suggest that, in these situations, which ultimately forced one or both replicas to drop packets, some of the packets missed by each replica were in fact different and recoverable from each other, at least in some specific conditions, which illustrates the protocol usefulness.

7 REFERENCES

[1] Roesch, M. (1999). Snort - lightweight intrusion detection for networks. *LISA '99: Proceedings of the 13th USENIX conference on System administration* (pp. 229-238). Berkeley, CA, USA: USENIX Association.

[2] Mukherjee, B., Heberlein, L. T., & Levitt, K. N. (1994, May/Jun.). Network intrusion detection.

[3] Stallings, W. (2003). *Network Security Essentials: Applications and Standards* (Second Edition ed.). (I. Pearson Education, Ed.) Upper Saddle River, New Jersey: Prentice Hall.

[4] Ptacek, T. H., & Newsham, T. (Jan. 1998). *Insertion, evasion, and denial of service: Eluding network intrusion detection.* Technical report, Secure Networks, Inc.

[5] Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput. , 1* (1), 11-33.

[6] Défago, X., Schiper, A., & Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys , 36* (4), 372-421.

[7] Ansel, J., Aryay, K., & Cooperman, G. (2009). Dmtcp: Transparent check-pointing for cluster computations and the desktop. *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing* (pp. 1-12). Washington, DC, USA: IEEE Computer Society.