SieveQ: A Layered BFT Protection System for Critical Services

Miguel Garcia, Nuno Neves, *Member IEEE,* Alysson Bessani

Abstract—Firewalls play a crucial role in assuring the security of today's critical infrastructures, forming a first line of defense by being placed strategically at the front-end of the networks. Sometimes, however, they have exploitable weaknesses, allowing an adversary to bypass them in different ways. Therefore, their design should include improved resilience capabilities to allow them to operate correctly in highly adverse environments. This paper proposes *SieveQ*, a message queue service that protects and regulates the access to critical systems, in a way similar to an application-level firewall. *SieveQ* achieves fault and intrusion tolerance by employing an architecture based on two filtering layers, enabling efficient removal of invalid messages at early stages and decreasing the costs associated with Byzantine Fault-Tolerant (BFT) replication of previous solutions. Our experimental evaluation shows that *SieveQ* improves existing replicated-firewalls resilience in the presence of corrupted messages by faulty nodes. Furthermore, it accommodates high loads, as it is able to handle sixteen times more security events per second than what was processed by the Security Information and Event Management (SIEM) infrastructure employed in the 2012 Summer Olympic Games.

Index Terms—Intrusion Tolerance; BFT; Application-level firewall; Denial-of-service

1 Introduction

Firewalls are used as the main protection against external threats, controlling the traffic that flows in and out of a network. Typically, they decide if a packet should go through (or be dropped) based on the analysis of its contents. Over the years, this analysis has been performed at different levels of the OSI stack (see [2] for a comprehensive survey), but the most sophisticated rules are based on the inspection of application data included in the packets. State-of-the-art solutions for application-level firewalls include network appliances from several vendors, such as Juniper [3], Palo Alto [4], and Dell [5]. Such appliances are strategically placed on the network borders, and therefore, the security of the whole infrastructure relies on them. A skilled attacker, however, may find weaknesses to compromise the firewall's detection/prevention capabilities. When this happens, critical services under the protection of such devices may be affected, as in the Supervisory Control And Data Acquisition (SCADA) systems targeted in the Stuxnet [6] or Dragonfly [7] attacks.

Most generic firewall solutions suffer from two inherent problems: First, they have vulnerabilities as any other system, and as a consequence, they can also be the target of advanced attacks. For example, the National Vulnerability Database (NVD) [8] shows that there have been many security issues in commonly used firewalls. NVD's reports present the following numbers of security issues between 2010 and 2015: 157 for the Cisco Adaptive Security Appliance; 109 in Juniper Networks solutions; 29 for the Sonicwall firewall; and 24 related to iptables/netfilter. Common protection solutions often have been the target of malicious actions as part of a wider scale attack, e.g., anti-virus software [9], intrusion detection systems [10] or firewalls [11], [12], [13], [14]. Second, firewalls are typically a single point of failure, which means that when they crash, the ability of the protected system to communicate may be compromised, at least momentarily. Therefore, ensuring the correct operation of the firewall under a wide range of failure scenarios becomes imperative. To tolerate faults, one typically resorts to the replication of the components.

Primary-backup replication (also called $1+1$ replication) would suffice to tolerate a single crash fault on a firewall. If the primary replica crashes, the backup replica is able to replace it and deliver the requests. If the system wants to tolerate arbitrary failures, e.g., replicas that suffer intentional or non-intentional Byzantine faults, then it needs a more complex type of replication. For example, if one of the two replicas is misbehaving then the node implementing the critical service will be unable to distinguish which replica is delivering the correct message. To address this difficulty, it is necessary to collect a majority of correct messages, which requires the addition of more replicas. A system that needs to tolerate a single Byzantine fault must have at least 4 replicas [15].

In this work, we propose a new protection system called *SieveQ* that mixes the firewall paradigm with a message queue service, with the goal of improving the state-of-the-art approaches under accidental failures and/or attacks. The solution has a fault- and intrusion-tolerant architecture that applies filtering operations in two stages acting like a sieve. The first stage, called *pre-filtering*, performs lightweight checks, making it efficient to detect and discard malicious messages from external adversaries. In particular, messages are only allowed to go through if they come from a pre-defined set of authenticated senders. Denial-of-service (DoS) traffic from external sources is immediately dropped, preventing those messages from overloading the next stage. The second stage, named *filtering*, enforces more refined application level policies,

- *M. Garcia, N. Neves and A. Bessani are with the LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal. Authors e-mails: mhenriques@lasige.di.fc.ul.pt, nfneves@ciencias.ulisboa.pt, and anbessani@ciencias.ulisboa.pt.*
- *A preliminary version of* SieveQ*'s architecture was previously published in a workshop [1]. This paper presents the detailed protocols, and an extensive evaluation of a* SieveQ *prototype.*
- *This work was partially supported by the EC through projects FP7-607109 (SEGRID) and FP7-257475 (MASSIF), and through funds of the Fundação para a Ciência e a Tecnologia (FCT) with reference to UID/CEC/00408/2013 (LaSIGE).*

which can require the inspection of some message fields or need the enforcement of certain ordering rules.

Different fault tolerance mechanisms are employed at the two stages. Pre-filtering is implemented by a dynamic group of nodes named *pre-SQs*. *Pre-SQs* can be the target of various kinds of attacks and eventually may be intruded because they face the external network. Therefore, we take the conservative approach of assuming that *pre-SQs* can fail in an arbitrary (or Byzantine) way, meaning that they may crash or start to act maliciously. When a failed *pre-SQ* is detected, it is simply replaced by a new one that is clean from errors. Since *SieveQ* needs to support different message loads, e.g., due to additional senders, *pre-SQs* can be created dynamically to amplify the aggregated processing capabilities (within the constraints of the hardware). The filtering stage is performed by a group of *replica-SQ* components, which execute as a replicated state machine [16]. *Replica-SQs* may also fail in an arbitrary way, and therefore, we employ an intrusion-tolerant replication protocol that ensures correct operation in the presence of Byzantine faults.

SieveQ was experimentally evaluated in different scenarios. The results show that it is much more resilient to DoS attacks and various kinds of intrusions than existing replicated-firewall approaches. We also evaluated *SieveQ* considering the protection of a SIEM system. The test environment emulated the setup of the 2012 Summer Olympic Games, where the same sort of security events was generated and transmitted across the network. The experiments demonstrate that *SieveQ* can handle a workload up to sixteen times higher than the observed load in the 2012 Summer Olympic Games, without a noticeable degradation in performance.

The remainder of the paper is organized as follows: in Section 2 we explain what is an intrusion-tolerant firewall; the *SieveQ* architecture and protocol are presented in Sections 3 and 4, respectively; Sections 5 and 6 describe the implementation and present the system evaluation; in Section 7 we discuss the work limitations; in Section 8 we give an overview of the work related with our contributions; and finally we conclude in Section 9.

2 Intrusion-Tolerant Firewalls

In the last decade, several important advances occurred in the development of intrusion-tolerant systems. However, to the best of our knowledge, very few works proposed intrusion-tolerant protection devices, such as firewalls. Performance reasons might explain this, as Byzantine fault-tolerant (BFT) replication protocols are usually associated with significant overheads and limited scalability. Additionally, achieving complete transparency to the rest of the system can be challenging to reconcile with the objective of having fast message filtering under attack.

Figure 1 shows an implementation of an intrusion-tolerant firewall, illustrating existing works in this area [17], [18]. In this design, a sender transmits the messages through the network (e.g., the internet) towards the receiver. As packets reach the firewall, they are disseminated to the replicas. Each replica applies the same filtering rules to decide whether the messages are acceptable. Invalid messages are discarded (and eventually logged). Messages deemed valid are conveyed to the receiver together with a proof of validity, which demonstrates that a sufficiently large quorum of replicas agrees on their validity. The proof of validity is checked by a voter module at the receiver, before delivering the messages to the receiving application. Thus, if a compromised replica produces a message with malicious content, it will be eliminated as it lacks

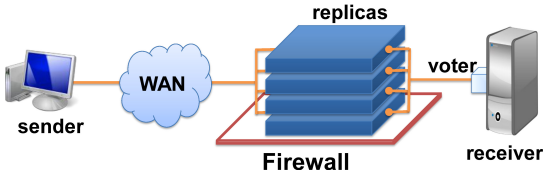


Figure 1: Architecture of a state-of-the-art replicated firewall.

the necessary proof of validity or it is in conflict with the messages transmitted by the other correct replicas.

Although this architecture has interesting characteristics, such as an increased failure resilience, it suffers from some fundamental limitations:

1) The dissemination of a message to all replicas can be detrimental to the proper operation of the firewall. For example, a traffic replicator device (e.g., hub) can be placed at the entry of the firewall to transparently reproduce all messages [17], [18]. An obvious consequence of this approach is that malicious messages from an external attacker are also replicated, and therefore, all replicas have to spend the effort to process them. The effect is an attack amplification caused by the replicator device. Alternatively, a leader replica could receive the traffic and then disseminate the messages to the others [18]. The drawback is that the leader becomes a natural bottleneck, especially when under attack (instead of dispersing the attack load over all replicas [19]).
2) The support for stateful firewall filtering requires that all correct replicas process messages in the same order [16]. As a consequence, to ensure an agreement in a common sequence of messages, replicas need to continuously run a BFT consensus protocol [20] to establish message ordering. A significant amount of work can be wasted with malicious messages since all messages have to be agreed. This is particularly relevant because a consensus protocol consumes both computational and network resources.
3) The creation and check of the proof of validity can be a complex task. For example, one approach requires a trusted component to be deployed in the replicas to generate a Message Authentication Code (MAC) as a proof that a message is valid [17]. The component only returns the MAC when a quorum of replicas accepted the message. Another solution uses threshold cryptography to ensure that every replica can individually produce a partial signature (that corresponds to a part of the proof) [18]. To recreate the full proof, the voter needs to wait for the arrival of a quorum of partial proofs. When building a firewall, it would be useful if a simpler approach could be employed, with no need for specialized trusted components or expensive threshold cryptography.

These drawbacks can have a significant impact on the firewall performance depending on the considered setting. For example, a typical DoS attack can create a substantial decrease in the throughput and several orders of magnitude growth in the latency of message delivery. To illustrate this behavior we implemented the architecture of Figure 1 and launched a DoS attack on this system (see Section 6 for a description of the setup and environment). The results show that the system performance is significantly affected by the attack (see Figure 2).

In this paper, we explore a different design for replicated protection devices, where we trade some transparency on senders

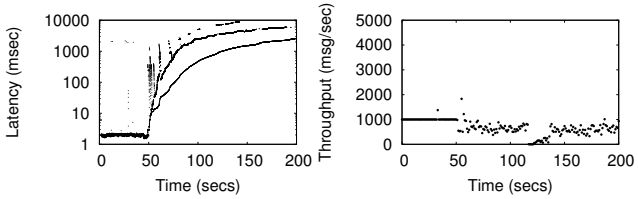


Figure 2: Effect of a DoS attack (initiated at second 50) on the latency and throughput of the intrusion-tolerant firewall architecture displayed in Figure 1.

and receivers for a more efficient and resilient firewall solution. In particular, we propose an architecture in which critical services and devices can only be accessed through a message queue, and implement the application-level filtering in this queue. It is assumed that these services have a limited number of senders, which can be properly configured to ensure that only they are authorized to communicate through *SieveQ*.

3 Overview of *SieveQ*

Typical resilient firewall designs are based on primary-backup replication, and consequently, they are able to tolerate only crash failures. Therefore, more elaborated failure modes may allow an adversary to penetrate into the protected network.

Some organizations deal with crashes (or DoS attacks) by resorting to several firewalls to support multiple entry points. This solution is helpful to address some (accidental) failures, but is incapable of dealing with an intrusion in a firewall. In this case, the adversary gains access to the internal network, enabling an escalation of the attack, which at that stage can only be stopped if other protection mechanisms are in place.

SieveQ provides a message queue abstraction for critical services, applying various filtering rules to determine if messages are allowed to go through. *SieveQ* is not a conventional firewall and we do not claim that it should replace existing firewalls in all deployment scenarios. We are focusing on service- or information-critical systems that require a high-level of protection, and therefore, justify the implementation of advanced replication mechanisms. The system we propose is able to deliver messages while guaranteeing authenticity, integrity, and availability. As a consequence, and in contrast to conventional firewalls, we lose transparency on senders and receivers, since they are aware of the *SieveQ*'s end-points. The rest of the section explains how we address some of the mentioned issues and introduces the main design choices and the architecture of *SieveQ*.

3.1 Design Principles

Our solution was guided by the following principles:

- *Application-level filtering*: support sophisticated firewall filtering rules that take advantage of application knowledge. *SieveQ* implements this sort of rules by maintaining state about the existing flows, and this state has to be consistently replicated using a BFT protocol.
- *Performance*: address the most probable attack scenarios with highly efficient approaches, and as early as possible in the filtering stages; Reduce communication costs with external senders, as these messages may have to travel over high latency links (e.g., do not require message multicasts).

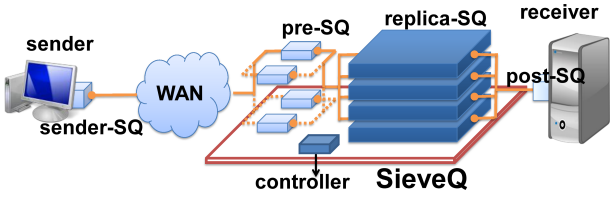


Figure 3: *SieveQ* layered architecture.

- *Resilience*: tolerate a broad range of failure scenarios, including malicious external/internal attackers, compromised authenticated senders, and intrusions in a subset of the *SieveQ* components; Prevent malicious external traffic from reaching the internal network by requiring explicit message authentication.

3.2 *SieveQ* Architecture

A fundamental difference of the *SieveQ* architecture, when compared with other replicated firewall designs (see Figure 1), is the separation of filtering in several stages. The rationale for this change is to gain flexibility in the filtering operations while ensuring better performance under attack, retaining the ability to tolerate intrusions. As observed previously, despite the significant improvements in state-of-the-art BFT implementations, there is an inherent trade-off between the benefits of BFT replication and performance, namely due to the need to disseminate (and eventually authenticate) all messages at the replicas, which includes both valid and invalid messages.

Figure 3 presents the architecture of *SieveQ*. In this architecture, message processing starts with a first filtering layer that implements a message authentication mechanism and is responsible for discarding most of the malicious traffic in an efficient way. This layer is based on a set of *pre-SQ* modules, each of them in charge of the communications with a subgroup of senders. During a normal operation, a sender only interacts with its own *pre-SQ*. The assignment of a *sender-SQ* to a *pre-SQ* is done during the channel setup. Initially, a sender connects with one of a few statically-configured *pre-SQs*. If a *pre-SQ* becomes overloaded, it will request the creation of more *pre-SQs* (see details in Section 4.4.2) and/or hand of the new *sender-SQ* channel to another node.

The messages are sent to the second filtering layer by the *pre-SQ* to perform a more detailed inspection, which can take advantage of state information kept from previous messages and application-related rules. This layer is implemented by a group of *replica-SQs* acting together as a BFT replicated state machine. They receive all accepted messages by the *pre-SQs* and process them in the same order, which guarantees that every *replica-SQ* reaches the same decision (discard or accept a message). However, if f replicas are faulty their output could be different. Consequently, as long as more than two-thirds of the *replica-SQs* are correct, the right decision is taken by the *post-SQ* by performing a voting.

In the following, we describe each module of Figure 3:

- *sender-SQ*: The sender nodes cooperate with *SieveQ* to secure the messages by deploying locally a *sender-SQ* module. Its main role is to secure the messages and assist in the detection of some intruded *SieveQ* components. The module can be implemented inside the sender's operating system (OS) (e.g., as a kernel module or a specialized

device driver) or as a library to be linked with the applications. The decision to have this module corresponds to a trade-off in our design, where we are willing to lose some transparency to improve the system's resilience.

- *pre-SQ*: This is the *SieveQ* front-end, and although it only performs stateless filtering to improve efficiency, it can deter the most common attacks. *Pre-SQ* modules discard invalid messages, while the approved ones are forwarded to the *replica-SQ* using a *Byzantine total ordered multicast (TOM)* protocol [21]. The *pre-SQs* can be deployed, for instance, as virtual machines. The effect of this layer is a significant reduction in the communication and computational overhead caused by malicious packets at the *replica-SQ*.
- *replica-SQ*: These components implement a replicated filtering service that tolerates Byzantine faults. The actual filtering rules can be more or less complex depending on the needs of the critical service. The TOM ensures that *replica-SQs* receive the messages in the same order. Consequently, identical rules are applied across the *replica-SQs*, and therefore, the same decision should be reached on the validity of messages. Approved messages are individually transmitted by each *replica-SQ* to the final receiver (the others are dropped). Overall, this layer allows for sophisticated filtering as the replicas are stateful.
- *post-SQ*: This module runs on the receiver side and it is responsible for the delivery of messages to the application. A *post-SQ* carries out a voting operation on the arriving data because a *replica-SQ* might be intruded and corrupt messages. It delivers a message to the application only after receiving the same approved message from a quorum of *replica-SQs*. From a deployment perspective, this module can be implemented in the OS or as a library, as in the sender.
- *controller*: This module is a trusted component of *SieveQ* that runs with a high privilege. It takes input from the *replica-SQs* to decide on the creation or destruction of *pre-SQs*, if some misbehavior is observed. Depending on the actual *SieveQ* implementation, it can be developed in different ways. This sort of component was used in previous works and it can be implemented both in a centralized [18], [22] or distributed [17] way.

3.3 Resilience Mechanisms

The *SieveQ* architecture is built to tolerate both faults with an accidental nature (e.g., crashes) and caused by malicious actions (e.g., a vulnerability is exploited and a specific module is compromised). To be conservative, we assume that all failed components are controlled by a single entity, which will make them act together in the worst possible manner to defeat the correctness of the system. Therefore, failed components can for instance: stop sending messages, produce erroneous information, or try to delay the system. *SieveQ* performs several mitigation actions to guarantee a valid operation (as long as the number of faults is within the assumed bounds, see Section 4.1).

The most common attack scenario occurs when an external adversary attempts to attack a system that is being protected by *SieveQ*. He can deploy many nodes, whose aim is either to delay the communications or bypass *SieveQ* protection and reach the internal network. *SieveQ* addresses these attacks by discarding unauthenticated or corrupted messages with minimal effort at the *pre-SQ* filtering stage. As with any other firewall, if a DoS attack completely overloads the incoming channels, *SieveQ* cannot handle or react to the attack. The network needs to include other defense mechanisms to deal with this sort of problem [23].

As (authenticated) senders might be spread over many (outside) networks, it is advisable to consider a second scenario where an adversary is capable of taking control of some of these nodes. In this case, we assume that the adversary gains access to all data stored locally, including the *sender-SQ* keys. Thus, he will be able to generate traffic that is correctly authenticated, allowing these messages to go through the first filtering step. The messages are however still checked against the application related rules (namely, the ones defined in the *replica-SQ*), which can cause most malicious traffic to be dropped (e.g., a pre-defined *sender-SQ* can only send messages to a particular *post-SQ* accordingly to a specific application protocol). If the messages follow all the rules, the firewall has to forward them because they are indistinguishable from any other valid messages.

A third scenario occurs when the adversary is able to cause an intrusion in *SieveQ* and compromises a few of the *pre-SQs* and/or *replica-SQs*. When this happens, these components can act in an erroneous (Byzantine) way. However, unlike with an intruded *sender-SQ*, malicious *pre-SQs* cannot generate fully authenticated messages, since they lack all the required keys. They can still perform DoS attacks on the *replica-SQs*, e.g., by transmitting many messages, but this strategy creates an obvious misbehavior allowing immediate discovery. Malicious *pre-SQs* are detected with the assistance of correct *sender-SQ* and *replica-SQs*, eventually leading to their substitution.

Replica-SQs modules are much harder to exploit because they do not face the external network. However, if they end up being intruded, *replica-SQs* can produce arbitrary traffic to the internal network. *Post-SQ* addresses this issue by carrying out a voting step, which excludes these messages. Moreover, an alarm is generated and sent to the *controller*.

The *SieveQ* architecture makes no attempt to recover from intrusions in the *controller* and *post-SQ*. The first is assumed to be trusted, as it is deployed in a separated administrative domain and is to be used only in a few very specific operations. Moreover, its simplicity allows the audit of its code and ensures correctness with a high level of confidence. The *post-SQ* already runs in the internal network, and therefore, *SieveQ* can not preclude its misbehavior.

4 *SieveQ* PROTOCOL

This section details the *SieveQ* protocol, the system model and the service properties. We conclude the section with an analysis of the behavior of the system under different kinds of attacks and component failures, and highlight how the countermeasures integrated in our design mitigate such threats.

4.1 System and Threat Model

The system is composed of a (potentially) large number of external nodes, called *senders*, some internal nodes, called *receivers*, and *SieveQ* nodes. Senders run *sender-SQ* modules to be able to transmit packets through the *SieveQ*, while receivers receive validated messages by using *post-SQ* modules.

Communications can experience accidental faults or attacks. Thus, packets might be lost, delayed, reordered or corrupted, but we assume that if messages are retransmitted, eventually they will

be correctly received by *SieveQ*. The fault model also assumes that *sender-SQ*, *pre-SQ* and *replica-SQ* nodes can suffer from arbitrary (Byzantine) faults. When this happens, failed nodes may perform actions that deviate from their specification, including colluding against the system. However, at most f_{ps} *pre-SQs* from a total of $N_{ps} = f_{ps} + k$ (with $k > 1$), and f_{rs} *replica-SQ* from a total of $N_{rs} = 3f_{rs} + 1$ may fail. Redundant components should fail independently by employing diversity techniques such as software obfuscation [18] or even different off-the-shelf software stacks [24], [25]. A component that is unable to communicate is also considered faulty because from a practical perspective it is indistinguishable from a crashed module.

The cryptographic operations used in the *SieveQ* protocol are assumed to be secure, and therefore, they cannot be subverted by an adversary. Consequently, traditional properties of digital signatures, Message Authentication Codes (MACs), and hash functions will hold as long as the associated keys are kept safe. The deployment of *SieveQ* requires a key distribution scheme to create shared keys between the *sender-SQ* and the *pre-SQ* and to periodically re-issue private-public key pairs for the *sender-SQ*. We assume that the key distribution scheme is similar to solutions that already address this sort of problem (e.g., [26]). If required, the key distribution infrastructure could also be made intrusion-tolerant [27], [28].

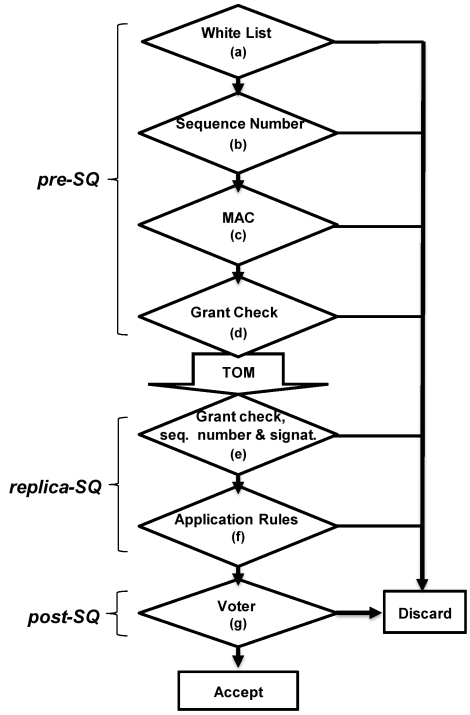


Figure 4: Filtering stages at the *SieveQ*.

4.2 Properties

SieveQ guarantees the following three properties for messages transmitted from a sender to a receiver:

Compliance. *If a message, transmitted by a correct sender, is delivered to a correct receiver then the message is in accordance with the security policy of* SieveQ.

Validity. *If a correct receiver delivers a message msg.DATA, then the message was transmitted by msg.sender.*

Liveness. *If a correct sender sends a message, then the message eventually will be delivered to the correct receiver.*

These properties require *SieveQ* to behave in a way similar to most firewalls while offering a few extra guarantees. Only external messages that are approved by the policies defined in *SieveQ* can reach the receivers and the rest should be dropped (Compliance). *Post-SQ* can use the message field msg.*sender* to find who transmitted the message contents (msg.*DATA*), and accordingly decide if the message should be delivered to the receiver application (Validity). Progress is also ensured, as correct senders eventually are able to transmit their messages (Liveness).

Besides these functional properties (related to message filtering), *SieveQ* also ensures a *resilience* property related with the detection and recovery of components of the system that exhibit faulty behavior:

Resilience. *Every component exhibiting observable faulty behavior will be eventually removed or recovered.*

In the following we present the mechanisms for implementing the *SieveQ* functionalities, i.e., the mechanisms for satisfying the three functional properties stated above. The mechanisms for ensuring resilience will be detailed in Section 4.4.

4.3 Message Transmission

4.3.1 Sender-SQ *processing*

This module gets a buffer with the *DATA* to be transmitted to a certain application placed behind *SieveQ*. The buffer needs to be encapsulated in a message with some extra information required for protection (see (1), below): we add a *sender-SQ* identifier S_i and a sequence number *sn* that is incremented on each message. This information is needed to prevent replay attacks, either from the network or from a compromised *pre-SQ*. Some information is also added to protect the integrity and authenticate the message. A signature sgn_{Si} is performed over the message contents, and a MAC mac_{ski} is computed using a shared key established with the *pre-SQ*. This MAC serves as an optimization to speed up checks [29].

$$msg = \langle S_i, sn, DATA, sgn_{Si} \rangle mac_{ski} \quad (1)$$

After constructing the message, it is sent to the *pre-SQ* assigned to the *sender-SQ*, and a timer is started. If this timer expires before the *sender-SQ* receives an acknowledgement message, it resends the *msg* to another *pre-SQ*.

4.3.2 Pre-SQ *filtering*

The *pre-SQ* determines if an arriving message *msg* should be forwarded or discarded (stages (a)–(d) in Figure 4). It applies the following checks to make this decision:

(a) *White list:* each *pre-SQ* maintains a list of the nodes that are allowed to transmit messages (i.e., which were authorized by the system administrator). Messages coming from other nodes are dropped. This check is based on the address of the message sender, and therefore it serves as an efficient first test, but it is vulnerable to spoofing.

(b) *Sequence number:* finds out if a message with sequence number *sn* from *sender-SQ* S_i was seen before. In the affirmative

case, the message is discarded to prevent replay attacks. Messages are also dropped if their *sn* is much higher than the largest sequence number ever observed from that *sender-SQ* (a sliding window of acceptable sequence numbers is used).

(c) *MAC test:* MAC mac_{ski} is verified to authenticate the message contents, including finding out that the expected S_i was the sender. If the check is invalid, the message is dropped and the sequence number information is updated to forget that this message was ever received (the update carried out in the previous step needs to be undone).

(d) *Grant check:* each *pre-SQ* controls the amount of traffic that a *sender-SQ* is transmitting. Messages that fall outside the allocated amount are dropped to ensure that all senders get a fair share of the available bandwidth (and to avoid DoS attacks by compromised senders). The amount of traffic that is allowed to each *sender-SQ* is adjusted dynamically based on the available and consumed resources.

Finally, the *pre-SQ* invokes the total order multicast primitive to forward the message to every correct *replica-SQ*.

4.3.3 Replica-SQ *filtering*

When a correct *replica-SQ* delivers a message to be filtered, the following checks are applied:

(e) *Grant check, sequence number and signature:* since a *pre-SQ* may have been intruded and may collude with malicious *sender-SQ*, extra checks are required on the amount of forwarded traffic and on the integrity of the message. The grant check and the test on the sequence number are similar to the ones performed by the *pre-SQ*, and the signature ensures that all *replica-SQs* reach the same decision regarding the validity of the message content.

(f) *Application-level rules:* apply the application-defined filtering rules to determine if the message is compliant with the security policy of the firewall.

Although uncommon, it can happen that *replica-SQs* receive messages in a different order from what is defined in their sequence numbers. As a consequence, the *replica-SQ*'s application-level rules may drop some of the out-of-order messages, which later on will have to be re-transmitted by the *sender-SQ*. For example, if messages `A` and `B` should appear in this sequence but are re-ordered, then the rules may consider `B` invalid and then accept `A`. At some point, the *sender-SQ* would consider `B` as lost, and re-transmit it.[1]

To address this issue, each *replica-SQ* enqueues for a while messages with a sequence number greater than the expected (but that do not exceed a threshold above the last processed sequence number). These messages are processed when either: 1) the missing messages with smaller sequence numbers arrive, and then they are all tested in order, or 2) the *replica-SQ* gives up on waiting, and checks the enqueued messages. This last decision is made after processing a pre-determined number of other messages.

Valid messages are encapsulated in a new format (see (2), below) and are sent to their receivers. Basically, *replica-SQ* RS_j substitutes the signature with a new MAC, mac_{skj}. This MAC is created with a shared key between the *replica-SQ* and the *post-SQ*.

$$msg' = \langle S_i, sn, DATA, RS_j \rangle mac_{skj} \quad (2)$$

1. It is important to remark that, independently of any violation on the order of delivery of sender messages, all correct *replica-SQs* receive the messages in the same order.

Algorithm 1: *post-SQ* protocol

```
   input : msg = <Si, sn, DATA, RSj>mac_skj
 1 Initialization : executed only once
 2 snExpect_Si ← 0;
 3 WaitingQuorom_Si,sn ← ∅;
 4 begin
 5     if ( verify_MAC(msg) == FALSE ) then
 6         return errorMAC;
 7     if (snExpect_Si ≤ msg.sn < (snExpect_Si + snThreshold)) then
 8         WaitingQuorom_Si,sn ← WaitingQuorom_Si,sn ∪ {msg};
 9     k ← snExpect_Si;
10     while ( equalMsg(WaitingQuorom_Si,k) ≥ f_rs + 1 ) do
11         deliver(mostVotedMsg(WaitingQuorom_Si,k)) ;
12         snExpect_Si ← snExpect_Si + 1 ;
13         k ← snExpect_Si;
14 end
```

4.3.4 Post-SQ *processing*

Post-SQ accumulates the messages that arrive from *replica-SQs* until enough evidence is collected to allow their delivery.

(g) *vote:* A message can be delivered to the application when it is received from $f_{rs}+1$ *replica-SQs*.

Algorithm 1 presents the *post-SQ* voting protocol. The *post-SQ* starts by checking that the message contains a valid MAC (Lines 5 and 6). Then, it finds out if the message carries an acceptable sequence number before storing it in a *WaitingQuorom* set (Lines 7 and 8). Notice that a different set is used for each sender S_i and *sn* pair. Messages with sequence numbers already delivered or higher than a threshold (*snThreshold*) are discarded.

The expected sequence number is stored in the auxiliary variable k (Line 9). Next, the *post-SQ* tries to find a message with this sequence number and with at least $f_{rs}+1$ votes (by searching the corresponding set and using function `equalMsg` — Line 10). For each different *DATA* value that may exist in the set, the `equalMsg` function counts the number of times its appears, and returns the largest count.[2] Next, while there are *msg*'s with a $f_{rs}+1$ quorum, *post-SQ* delivers *msg*'s in order (Line 10-13). The function `mostVotedMsg` returns the message with the *DATA* value with most votes (Line 11). Then, the `deliver` function delivers *DATA* to the application on the receiver and deletes the *WaitingQuorom* set. Finally, the *snExpect* is incremented (Line 12) and the k index is updated (Line 13). This allows *post-SQ* to deliver messages in order while there are messages with the expected sequence number in its buffer.

4.3.5 *Correctness Argument*

In the following, we show that our design satisfies the three functional properties described in Section 4.2. The proofs work under the assumptions of our system and threat model (Section 4.1).

Validity. *If a correct receiver delivers a message msg.DATA, then the message was transmitted by msg.sender.*

Proof. Assume that a receiver R_j gets a message content msg.*DATA* with the msg.*sender* = msg.S_i. For this to happen, the *post-SQ* of R_j waited for the arrival of at least $f_{rs}+1$

2. Notice that all correct *replica-SQs* transmit messages with the same *DATA* value and that malicious replicas can send at most f_{rs} arbitrary *DATA* values. Therefore, eventually there will be a *DATA* value with at least $f_{rs}+1$ votes because there are at least $2f_{rs}+1$ correct *replica-SQs*.

correctly authenticated messages[3] with equal msg.S_i, msg.sn, and msg.*DATA*. Therefore, at least $f_{rs}+1$ *replica-SQs* received *msg* signed by S_i, and verified the signature sgn_{Si} as valid. This indicates that msg.*DATA* was not modified by the network or by any *pre-SQ*. Only a sender msg.S_i (correct or not) is able to create and authenticate a *msg* with its signature. Therefore, S_i is the msg.*sender*, i.e., the creator of msg.*DATA*. □

Compliance. *If a message, transmitted by a correct sender, is delivered to a correct receiver then the message is in accordance with the security policy of* SieveQ.

Proof. The proof of the *Compliance* property is very similar to the *Validity* proof. The difference is that, we also know that if $f_{rs}+1$ *replica-SQs* sent *msg* to a *post-SQ*, then *msg* was verified against the security policy in at least one correct *replica-SQ*, which approved it. □

Liveness. *If a correct sender sends a message, then the message eventually will be delivered to the correct receiver.*

Proof. For the sake of simplicity, our proof only considers nodes that drop or delay messages. Attacks to the integrity will make the message be discarded (i.e., dropped).

Assume that a sender S_i transmits a message *msg* with the sequence number *sn* to a *pre-SQ* PS_u. Then S_i sets a timer $timer_{sn}$ for msg.sn. If PS_u is correct, after it receives a message, it re-sends *msg* via TOM to the *replica-SQs*. At least $f_{rs}+1$ correct *replica-SQs* will send *msg* to the *post-SQ*, which will deliver msg.*DATA* to the receiver R_j. If the PS_u is faulty, i.e., drops or delays messages, $timer_{sn}$ in S_i will eventually expire. When this happens, S_i re-transmits *msg* to another *pre-SQ*. Eventually, this process will make some correct *pre-SQ* forward *msg* to the *replica-SQs* using the TOM primitive, which will cause msg.*DATA* to be delivered to R_j. □

4.4 Addressing Component Failures

In this section, we discuss the implications of failures in the different components of the *SieveQ*, and how they are handled for ensuring the *Resilience* property stated in Section 4.2.

In the Byzantine model, every failed component can behave in an arbitrary way, intentionally or accidentally. Therefore, *SieveQ* design incorporates mechanisms that are resilient to different failure scenarios. Given the architecture of Figure 3, one has to address faults in authenticated *sender-SQs*, *pre-SQs* and *replica-SQs*, as they are the main components subject to Byzantine failures in our model. Other components of our architecture, such as the *controller* and *post-SQ* are not considered in this section as they are either assumed to be trusted or co-located with the receiver.

In the following, we try to focus on complex scenarios in which faulty components send syntactically-valid messages, avoiding cases that could be easily detected and recovered by existing network monitoring and protection tools (e.g., it is easy to discover that a *pre-SQ* is sending messages to another *pre-SQ*, something our protocol does not allow).

Since *pre-SQs* are directly exposed to the external network, there is a higher risk of them being compromised. Then, to keep the *SieveQ* operational, it is required that failed *pre-SQs* are identified and recovered. We leverage from the *replica-SQ* setup to perform failure detection, and then use the controller to restart erroneous *pre-SQs*. Replacing these components is almost trivial because they are stateless.

Replica-SQs execute as a BFT replicated state machine, processing messages in the same order and producing identical results. Consequently, *replica-SQs* faults can be tolerated by employing a voting technique on the *post-SQ* that selects results supported by a sufficiently large quorum (as explained above, an output with at least $f_{rs}+1$ votes). Below, we discuss in more detail a few failure scenarios.

4.4.1 *Faulty* Sender-SQ

A faulty *sender-SQ* is authorized to communicate while suffering from some arbitrary problem (e.g., intrusion). Therefore, it can produce correctly authenticated messages to attack the firewall. In some scenarios, it is possible to discard these messages. For example, if the *SieveQ* receives a correctly signed message with a sequence number much larger than the expected, then it can be easily detected and eliminated (checks *(b)* and *(e)* in Figure 4).

A more demanding scenario occurs when a *sender-SQ* transmits faster than the allowed rate (verification (d) of Figure 4). In this case, some defense action has to be carried out, as these attacks can lead *SieveQ* to waste resources. To be conservative, we decided to follow a simple procedure to protect the firewall: *SieveQ* maintains a counter per sender that is incremented whenever new evidence of failure can be attributed to it, e.g., the faulty *sender-SQ* is overloading the system with invalid messages or if it is sending messages to *pre-SQs* which it was not assigned. When the counter reaches a pre-defined value, the *sender-SQ* is disallowed from communicating with *SieveQ* by temporarily removing it from the white list and adding it to a quarantine list (failing verification (a) of Figure 4) and by giving a warning to the system administrator. *This ensures that faulty sender-SQs are eventually removed from the system.*

Excluded *sender-SQs* may regain access to the service later on because the counter is periodically decreased (when the counter falls below a certain threshold, the *sender-SQ* is moved back into the white list). Additionally, the administrator is free to update the white/quarantine lists. For instance, he may choose to manually add a faulty sender to the quarantine list or even deploy policies to do that under certain conditions (e.g., if the same sender is in the quarantine list for a certain number of times).

4.4.2 *Faulty* Pre-SQ

Addressing failures in *pre-SQs* is difficult because these components may look as compromised even when they are correct. Notably, when a *pre-SQ* is under a DoS attack, messages can start to be dropped due to buffer exhaustion, and this is indistinguishable from a malicious behavior in which messages are selectively discarded. In the same way, a failed signature check at a *replica-SQ* indicates that either the *pre-SQ* is faulty (it is tampering/generating invalid messages) or that a *sender-SQ* is misbehaving (recall that a *pre-SQ* verifies the message MAC, but not its signature). Finally, a *replica-SQ* may also detect problems if it observes a sudden increase in the arrival of messages (above the grant check), which could indicate a DoS attack by a malicious *pre-SQ* (maybe colluding with a compromised *sender-SQ*). This kind of ambiguity precludes exact failure detection, and consequently, our aim is to provide a mechanism that allows *SieveQ* to recover from end-to-end problems and continue to deliver a correct service.

3. Note that *replica-SQs* send msg' (defined in (2)) instead of *msg* (defined in (1)). Both are similar, but msg' has a MAC instead of a signature to authenticate its contents with *post-SQ* (see Section 4.3.3). For the sake of simplicity we will only use the *msg* notation in the rest of the proof.

An initial step to deal with these complex failure scenarios is to make *pre-SQs* evaluate their own state. This is done by analyzing the amount of arriving traffic and by observing if it could overload the *pre-SQ*. The analysis can be done by measuring the inter-arrival times of messages over a certain period. If those intervals are very small (on average), there is a chance that the *pre-SQ* is working at its full capacity or is even overloaded. When this happens, the *pre-SQ* broadcasts (using the TOM primitive) a WARNREQ message to the *replica-SQs*, so that they may take some action to solve the problem (see below).

When the *pre-SQ* is faulty, the *sender-SQ* and *replica-SQs* need to detect it together. *SieveQ* provides a procedure to find how many messages are being discarded on *pre-SQ*:

1) Periodically, the *sender-SQ* sends a special ACKREQ request to *replica-SQs*, in which it indicates the sequence number of the last message that was sent (plus a signature and a MAC). This request is first sent to the preferred *pre-SQ*, but if no answer is received within some time window, it is forwarded to another *pre-SQ*. The waiting period is adjusted in each retransmission by doubling its value.
2) When the *replica-SQs* receive the request, the included sequence number together with local information are used to find how many messages are missing. The local information is basically the set of sequence numbers of the messages that were correctly delivered since the last ACKREQ.
3) Based on the number of missing messages, the *replica-SQs* transmit through the same *pre-SQ* a response ACKRES to the sender, where they state the observed failure rate and other control information (plus a signature). *Replica-SQs* may also perform some recovery action if the failure rate is too high.

Additionally, if a *replica-SQ* detects that a signature is invalid or that it is receiving more messages than the expected (check (e) in Figure 4), it suspects the *pre-SQ* that sent the messages and takes some action.

Once a problem is detected, the *replica-SQs* should attempt to fix the erroneous behavior by employing one of three possible remediation actions, depending on the extent of the perceived failures:

- *Redistribute load:* if a *pre-SQ* has sent a warning about its load, or a high failure rate related with this component is observed, the first course of action is to move some of the message flows from the problematic component to other *pre-SQ*. This is achieved by specifying, in the ACKRES response to a *sender-SQ*, the identifier of a new *pre-SQ* that should be contacted. At that point, the *sender-SQ* is expected to connect to the indicated *pre-SQ* and begin sending its traffic through it.
- *Increase the* pre-SQs *capacity:* if the existing *pre-SQs* are unable to process the current load, then the *SieveQ* needs to create more *pre-SQs* (depending on the available hardware resources). To do that, *replica-SQs* contact the *controller* informing that an extra *pre-SQ* should be started. When the controller receives $f_{rs}+1$ messages, it performs the necessary steps to launch the new *pre-SQ* (which are dependent on the deployment environment). The new *pre-SQ* begins with a few startup operations, which include the creation of a communication endpoint, and then it uses the TOM channel to inform the *replica-SQ* that it is ready to accept messages from *sender-SQs*.
- *Kill the* pre-SQ*:* when there is a significant level of suspicion on a *pre-SQ*, the safest course of action is for *replica-SQs* to ask the controller to destroy it. Moreover, if the load on the firewall is perceived as having decreased substantially, the *replica-SQs* select the oldest *pre-SQ* for elimination, allowing eventual aging problems to be addressed. The controller carries out the needed actions when it gets $f_{rs}+1$ of such requests (once again, which depend on how *SieveQ* is deployed). The affected *sender-SQs* will be informed about the *pre-SQ* replacement through the ACKREQ mechanism, i.e., they will eventually use another *pre-SQ* to send a request, and get the information about their newly assigned *pre-SQ* in the response. Moreover, the new *pre-SQ* is informed about the expected sequence number for each *sender-SQ*. This information is stored by *replica-SQs*, which contrary to *pre-SQs* are stateful.

Together, these mechanisms ensure that *a faulty* pre-SQ *affecting the* SieveQ *performance will be eventually removed from the system*.

4.4.3 *Faulty* Replica-SQ

A *post-SQ* only delivers a message if it receives $f_{rs}+1$ matching approvals for this message. Given the number of *replica-SQs*, this quorum is achievable for a correct message even if up to f_{rs} *replica-SQs* are faulty. However, a faulty *replica-SQ* can create a large number of messages addressed to other components of the system, effectively causing a DoS attack. Therefore, we need countermeasures to disallow a faulty *replica-SQ* to degrade the performance of the system in a similar way as illustrated in Figure 2. For example, a faulty *replica-SQ* can send an unexpected amount of messages to a *pre-SQ*, making it slower and triggering suspicions that may led it to be killed. Similarly, it can attack other *replica-SQs* to make the system slower. Finally, a faulty *replica-SQ* can also overload the *post-SQ* with invalid messages.

Each of these attacks require a different detection and recovery strategy:

- When a *pre-SQ* is being attacked it complains to the controller, which first requests the *pre-SQ* replacement (assuming it might be compromised) and increases a suspect counter against the *replica-SQ*. If $f_{ps}+1$ *pre-SQs* also complain about the same *replica-SQ*, the controller starts an *individual recovery* in this *replica-SQ* (see below).
- If more than f_{rs} other *replica-SQs* are being attacked, they will probably become slower. In this case, it is possible to detect such attacks if $f_{rs}+1$ *replica-SQs* complain about a single *replica-SQ*. This would be possible because target *replica-SQs* would observe a high unjustifiable traffic coming from a single replica and because such spontaneously generated messages would be deemed invalid. When this attack is detected, the controller starts an individual recovery in this *replica-SQ*.
- If only up to f_{rs} other *replica-SQs* are being attacked it is still possible to make the system slower without being detected by the previous mechanism. This happens because $N_{rs}-f_{rs}$ replicas must participate in the TOM protocol [21], and the target f_{rs} plus the attacker intersect this quorum in a least one component. This intersecting component will define the pace of the messages coming, which typically will be slow. We can mitigate this

attack as each *replica-SQ* periodically informs the *post-SQ* about its throughput (number of processed messages per second), piggybacking this value in some approved messages submitted for voting. The *post-SQ* verifies that there are *replica-SQs* presenting throughputs lower than expected and initiates a *recovery round* on all the *replica-SQs* (as described below).
- When the *post-SQ* is being attacked it detects the abnormal behavior. Then, it requests the individual recovery of the compromised replica.

The previous mitigation mechanisms suggest two kinds of recovery actions. First, an individual recovery, where the machine is rebooted with clean code (possibly using a different OS image to avoid common vulnerabilities [25]) and then is reintegrated in the system. Second, a recovery round is used when a performance degradation attack is detected but there is no certainty about which *replica-SQ* was compromised. Therefore, in this case, we recover all replicas, one after another to avoid unavailability periods in the system, just like in proactive recovery systems [17], [18], [20]. There are several works that present these techniques, and most of them are compatible with our architecture, therefore we refrain from describing them in more detail. Notice that the use of a recovery round is enough to ensure that *a faulty replica-SQ will eventually be recovered in SieveQ*.

5 Implementation

We implemented a prototype of *SieveQ* following the specification of the previous section to validate our design. The *sender-SQ* and *post-SQ* were developed as libraries that are linked with the sender and receiver applications. The libraries offer an interface similar to the TCP sockets, to simplify the integration and minimize changes in the applications. A sender can provide a buffer to be transmitted and the receiver can indicate a buffer where the received data is to be stored. Internally, the *sender-SQ* library adds a MAC and a signature to each message. The MAC is created using HMAC with the SHA-256 hash function and a 256-bit key, and the signature employs RSA with 512-bit keys. Messages are authenticated at the *post-SQ* also using HMAC. The RSA keys were kept relatively small for performance reasons. However, since they should be updated periodically (e.g., every few hours), this precludes all practical brute force attacks [30].

The *pre-SQs* operate as separate processes receiving the messages and forwarding them to the *replica-SQs*. We resorted to BFT-SMaRt, a state machine replication library [21], to implement the TOM and manage the *replica-SQs*. BFT-SMaRt, like most SMR systems (e.g., PBFT [20], Zyzzyva [31], Prime [19]) follows a client-server model, where a client transmits request messages to a group of server replicas and then receives the output messages with the results of some computation. We had to modify BFT-SMaRt because this model does not fit well with the message flow of *SieveQ*. Therefore, we have decoupled the client-server model into a client-server-client model. The client sends messages (but does not wait for responses as in the traditional SMR), and the server forwards them (after the validation) to another client, which is the final receiver. A reverse procedure is carried out for the traffic originating from the receiver. Furthermore, in BFT-SMaRt, the replicated servers are typically designed with a single-thread to process requests. To improve performance, we also modified the system to allow CPU-costly operations (like a signature verification) to occur concurrently with the rest of the checks performed by *replica-SQs*.

In the prototype, a *pre-SQ* can be replaced on two occasions: first, voluntarily by asking for a substitution to the *replica-SQs*, when it is flooded with unauthorized messages (DoS attack); and second, when a *replica-SQ* detects message corruptions by a *pre-SQ*. In both situations, the *replica-SQs* make a request to the *controller*, which will replace a *pre-SQ* instance. Notice that the *controller* has to wait for $f_{rs}+1$ messages, requesting a *pre-SQ* replacement, to ensure that a faulty *replica-SQ* cannot force the recovery of a correct *pre-SQ*. The *replica-SQs* are recovered when the collected information indicates that some component might be attacking other components. The information is collected and sent to the trusted controller to evaluate and decide if the replicas are making progress as expected. The information allows the system to suspect on $f_{rs}+1$ *replica-SQs* and recover them (there is no need to recover all the *replica-SQs*).

Since BFT-SMaRt is programmed in Java, we decided to use the same language to develop the various *SieveQ* components. If the sender and receiver applications are coded in other languages, they can still be supported by implementing specific *sender-SQ* and *post-SQ* libraries.

6 Evaluation

This section focuses on the evaluation of *SieveQ* under different network and attack conditions. We present the results of four types of experiments. In the first one, we evaluate the latency for different *sender-SQ* workloads, assessing the performance of *SieveQ* in the absence of failures. The second experiment assesses the effect of filtering rules complexity on the performance of the system. In the third experiment, we assess the throughput in three scenarios: *i)* the normal case, *ii)* a DoS attack without countermeasures; and *iii)* a DoS with all the mechanisms defined in Section 4.4 enabled (in fact we considered two DoS attacks: external, from a malicious *sender-SQ*, and internal, from a compromised *replica-SQ*). The last experiment considers the capability of *SieveQ* to safeguard a SIEM system under a similar workload as the one observed in the 2012 Summer Olympic Games.

6.1 Testbed Setup

Figure 5 illustrates the testbed, showing how the various *SieveQ* components were deployed in the machines. We consider one *sender-SQ* and one *post-SQ* deployed in different physical nodes, and an additional host acted as a malicious external adversary. The *controller* and *pre-SQs* were located in the same physical machine for convenience but in different VMs. Four *replica-SQs* were placed in distinct physical nodes. Every machine had two Quad-core Intel Xeon 2.27 GHz CPUs, with 32 GB of memory, and a Broadcom NetXtreme II Gigabit network card. All the machines were connected by a 1Gbps switched network and run Ubuntu 10.04 64-bit LTS (kernel 2.6.32-server) and Java 7 (1.7.0_67).

6.2 Methodology

SieveQ acts as a highly resilient protection device, receiving messages on one side and forwarding them to the other side. Therefore, the performance of *SieveQ* is assessed with latency/throughput measurements that can be attained under different network loads. In the experiments, the sender transmits data at a constant rate, i.e., 100 to 10000 messages per second, with three different message

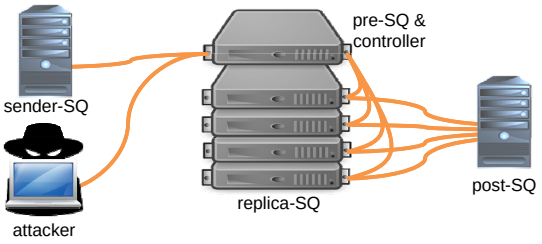


Figure 5: The *SieveQ* testbed architecture used in the experiments.

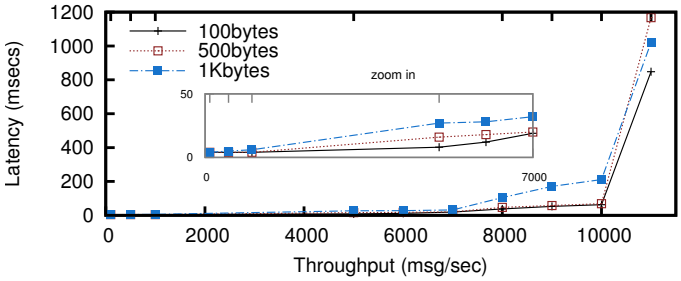


Figure 6: *SieveQ* latency for each workload (message size and transmission rate).

Payload size $(bytes)$	Non-optimized (msg/sec)	Optimized (msg/sec)
100	1360	10682
500	1320	10618
1000	1311	10332

Table 1: Maximum load induced by the *sender-SQ* library with various message sizes

payload sizes, i.e., 100 bytes, 500 bytes and 1k bytes. We used the Guava library [32] to control the message sending rate.

Latency measures the time it takes to transmit a message from *sender-SQ* until it is delivered to the application on the *post-SQ*. The following procedure was employed to compute the latency: *sender-SQ* obtains the local time before transmitting a message. When the message arrives and is ready to be delivered to the receiver application (after voting), the *post-SQ* returns an acknowledgment over a dedicated UDP channel. The *Sender-SQ* gets the current time again when the acknowledgment arrives. The latency of a message is the elapsed time calculated at *sender-SQ* (receive time minus send time) subtracted by the average time it takes to transmit a UDP message from *post-SQ* to *sender-SQ*.

Throughput gives a measure of the number of messages per second that can be processed by *SieveQ*. It was calculated at the *post-SQ* using a counter. This counter is incremented every time a message is delivered to the receiver application, and the counter is reset to zero after one second. Consequently, the server can calculate the number of messages delivered in every second. The throughput is computed as the average value of the individual measurements collected over a period of time (in our case, 5 minutes after the steady state was reached).

In some experiments, we wanted to assess the behavior of *SieveQ* under a DoS attack. The attack was made using PyLoris [33], a tool built to exploit vulnerabilities on TCP connection handling. The tool implements the Slowloris attack method, which opens many TCP connections and keeps them open. The tool allows the user to define parameters like group size of attack threads, the maximum number of connections, and the time interval between connections among others. In our setup, PyLoris was configured to perform an unlimited number of connections, with 0.1 milliseconds between each connection.

In all experiments, measurements were taken only after the Java Virtual Machine was warmed-up, and the disks were not used (all data is kept in memory).

6.3 Performance in failure-free executions

This experiment measures the latency of *SieveQ* with several message sizes and distinct message transmission rates. It demonstrates the overall performance of *SieveQ* in different scenarios, gradually stressing the *post-SQ* side as the workload is slowly increased. Measurements were collected after the system reached a steady state. The experiments were repeated 10 times for every workload and the average result is reported.

Figure 6 shows how the latency is affected by the transmission rate and message size. As expected, when the system becomes increasingly loaded, the latency grows proportionally because resources have to be shared among the various messages. The latency increase is approximately linear for the messages with 100 and 500 bytes until the throughput reaches $10k$ messages per second. The messages with $1k$ bytes have a linear increase on latency until the throughput reaches approximately $7k$ messages per second, and then it has a higher increase as more load is put on the system. This means that very high workloads can only be supported if applications have some tolerance to network delays. Overall, the performance degrades gracefully when varying the message payload sizes, for rates under $7k$ messages per second.

We performed a more detailed analysis of the overheads introduced by the various components of *SieveQ*. We observed that *sender-SQ* performs the most expensive operations, which is interesting because it shows that our design offloads part of the effort to the edges, reducing bottlenecks. The most important overheads were caused by the tasks associated with securing the message payload (which in fact are the most costly operations in the system). Several optimizations were made to mitigate the performance penalties during the message serialization (e.g., creation of the signature), including the use of parallelization to take advantage of the multicore architecture (as done, for example, in [34]). Table 1 shows the gain of using the optimized version of the *sender-SQ* library. Similar optimizations were employed in other components.

6.4 Effect of Filtering Rules Complexity

The results presented in the previous section do not consider any kind of complex filtering rule set. This section presents experiments with a fully loaded system and application-level filtering rules with different complexity at the *replica-SQs*.

Given the diverse requirements imposed by application-level firewalls, we decided to approximate the complexity of the filtering rules by considering a variable number of string matchings on processed messages. It is well recognized that the most costly aspect of message filtering is exactly finding (or not) specific strings in the packet contents (besides crypto verifications, which were included in all our experiments). The high cost of running such algorithms leads for instance to several implementations in FPGAs and GPUs to improve performance in firewalls and IDS (e.g., [35], [36]).

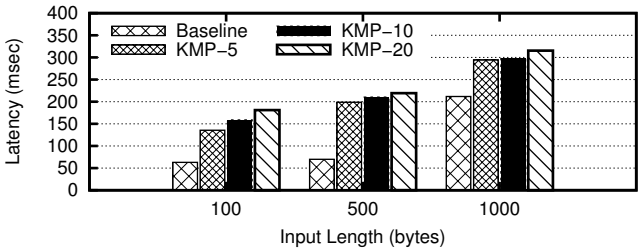


Figure 7: Comparison of the *SieveQ*'s latency between the baseline and adding the filtering rules.

We used a classical algorithm for string matching (KMP [37]) at the *replica-SQs* to implement the filtering rules. This algorithm is employed in IDSs [38] and firewalls like iptables [39]. The algorithm employs a pre-computed table to execute string matching in $O(n+k)$ complexity, where n is the string length and k is the pattern length.

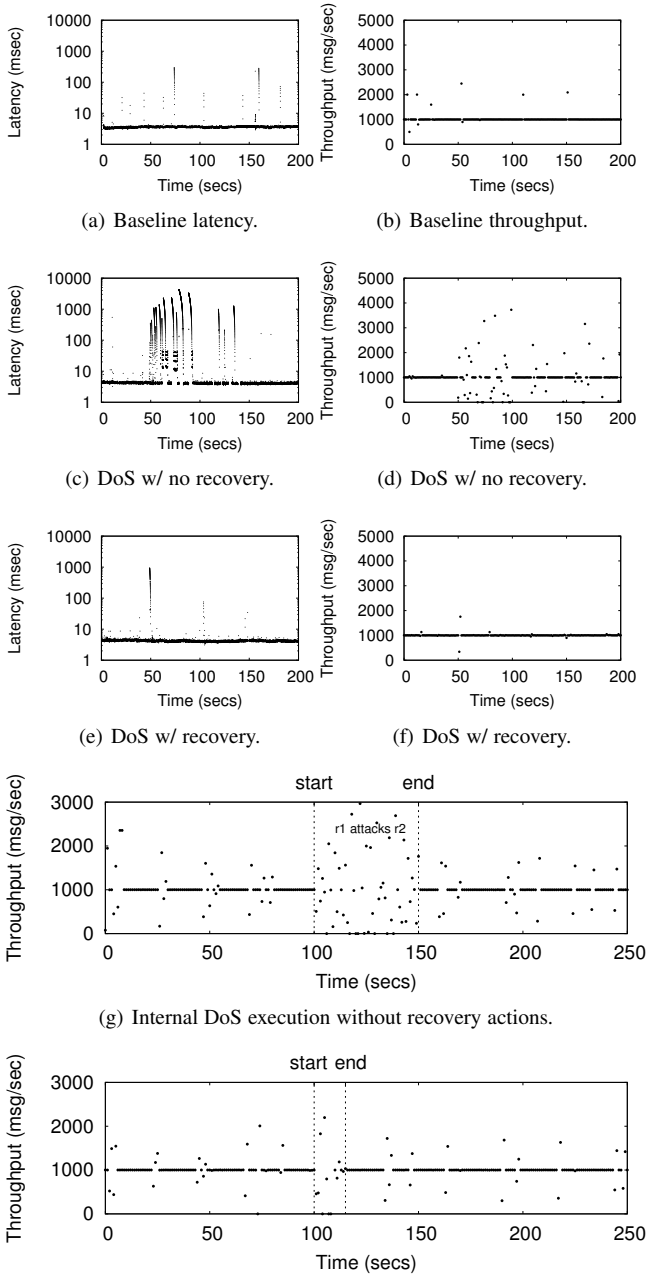
We measured the latency of the system by varying the message size from 100 to 1000 bytes and the string pattern size from 5 to 20 bytes. Both the message content and the strings were randomly generated. The experiments were performed with the maximum throughput of 10000 messages per second, as identified in the previous section. Figure 7 shows the latency of *SieveQ* without message filtering (Baseline) and string matching (KMP) with different sizes. In the last group of experiments, where the message size is 1000 bytes and the pattern is 20 bytes, the latency increases by 50%. In other cases, sometimes higher overheads were observed, e.g., with 500 bytes messages and 20 bytes pattern the overhead is approximately 200%. This result is expected as the string matching is a slow operation and it needs to be performed in the critical path of message processing. Implementations with hardware support (as mentioned before) could be integrated with *SieveQ* to reduce these delays significantly.

6.5 *SieveQ* Under Attack

Our next set of experiments aims to evaluate the system under different attack scenarios. Here, the *sender-SQ* creates a steady load of 1000 500-byte messages per second. We measured the latency and throughput of the system in three conditions: failure-free operation, a malicious external and internal DoS attack, and a malicious attack with remediation mechanisms. The results are reported in Figure 8. Before presenting the results, we need to stress that these experiments must be compared with the results displayed in Figure 2, which were obtained in the same way but with a different architecture (see Figure 1).

Figure 8(a) and Figure 8(b) show the latency and throughput, in the failure-free scenario. One can observe that latency stays on average around 3.4 milliseconds. The throughput is approximately constant during the whole period. It is possible to observe some momentary spikes in the latency and throughput, which happens due to Java garbage collector and a queuing effect from the SMR.

The behavior of the *SieveQ* during a DoS attack is displayed in Figure 8(c) and Figure 8(d). In this scenario, we have disabled the *SieveQ* capability of replacing *pre-SQs* at runtime. The attack consists in stressing the TCP socket interface of the *pre-SQs* by creating many TCP connections, which consumes network bandwidth and wastes resources at system and application levels. When the attack is started it executes for 50 seconds. The latency graph displays a reasonable impact in terms of an increase in the



(a) Baseline latency. (b) Baseline throughput.

(c) DoS w/ no recovery. (d) DoS w/ no recovery.

(e) DoS w/ recovery. (f) DoS w/ recovery.

(g) Internal DoS execution without recovery actions.

(h) Internal DoS execution with recovery actions.

Figure 8: Performance of *SieveQ* under different attack conditions.

delays for message delivery. In some cases, the latency is not too affected but in others, there is a drastic delay, with some messages taking more than 3 seconds. The attack also has consequences on the throughput as it is possible to observe an oscillation between 0 to 4000 messages per second (which correspond to the situation when the *post-SQ* processes a batch of messages that have been accumulated).

Figure 8(e) and Figure 8(f) show the latency and throughput when a similar DoS was carried out, but in this case the *SieveQ* replaced the *pre-SQ* under attack with a new *pre-SQ*. When the *pre-SQ* finds out that it is being overloaded with messages coming from non-authorized senders, it asks for a replacement. After that, the controller replaces the faulty *pre-SQ*, and the existing *sender-SQs* are contacted to migrate their connections. As the figures

show, the impact of the attack is minimized, since only a few messages are delayed and throughput is only affected momentarily while the *pre-SQ* is switched. Once the new *pre-SQ* takes over, the messages lost during the switching period are retransmitted and delivered. In practice, the attack becomes ineffective because, although it continues to consume network bandwidth, there is no longer a *pre-SQ* to process the malicious messages. An adversary could increase the attack sophistication and try to find a new *pre-SQ* target. However, even in this case the attack has limited effect because during an interval of time (while there is a search for a fresh target) the system can make progress.

Figure 8(g) and Figure 8(h) shows the *SieveQ* throughput when an internal attack is carried out by a compromised *replica-SQ*. The experiment was made with a *replica-SQ* ($r1$) launching a DoS to another *replica-SQ* ($r2$). The attack consists in overloading a *replica-SQ* with *state transfer* requests, which are the most demanding request a replica can receive in BFT-SMaRt [40]. Figure 8(g) shows the impact on the *SieveQ* throughput during an attack lasting 50 seconds, without any recovery capability on the system. As can be seen, the performance of the system is severely disrupted during the attack. Figure 8(h) shows the same attack but with the detection and recovery mechanism described in Section 4.4.3. The *post-SQ* detects the problem by noticing that $f_{rs}+1$ *replica-SQs* are sending less messages than the others, and then requests a recovery. When the *replica-SQ* is recovered it requests the state from the other replicas, and then after applying the new state, the replica resumes the normal execution (end line in the figure). In the experiment of Figure 8(h) we show a case in which the faulty *replica-SQ* is the first to be recovered. It could happen that *SieveQ* recovered f_{rs} *replica-SQs* before the faulty one. This would take $(f_{rs}+1)\times$ 3 seconds (in our setup) before the system resumes the normal execution.

6.6 Use Case: *SieveQ* to Protect a SIEM System

SIEM systems offer various capabilities for the collection and analysis of security events and information in networked infrastructures [41]. These systems are being employed by organizations as a way to help with the monitoring and analysis of their infrastructures. They integrate a large range of security and network capabilities, which allow the correlation of thousands of events and the reporting of attacks and intrusions in near real-time.

A SIEM operates by collecting data from the monitored network and applications through a group of sensors, which then forward the events towards a correlation engine at the core facility. The engine performs an analysis of the stream of events and generates alarms and other information for post-processing by other SIEM components. Examples of such components are an archival subsystem for the storage of data needed to support forensic investigations, or a communication subsystem to send alarms to the system administrators.

As part of the MASSIF European project [42], we have implemented a resilient SIEM system, where *SieveQ* was used to protect the access to the core facility (in Figure 9). In the *SieveQ* architecture, the sensors had the *sender-SQ* while the *post-SQ* was placed in the correlation engine. Additionally, we had access to an anonymized trace with the security events collected during the 2012 Olympic Games. Each event corresponds basically to a string describing some observed problem by a sensor. The strings had lengths varying between a minimum of 551 bytes and a maximum of 2132 bytes, with an average length of 1990 bytes

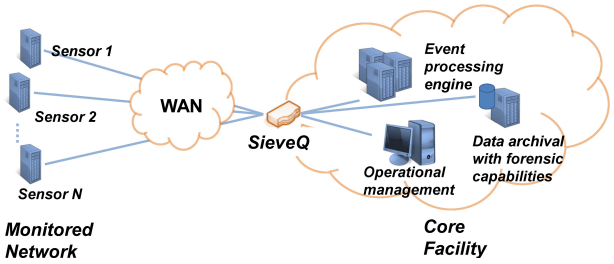


Figure 9: Overview of a SIEM architecture, showing some of the core facility subsystems protected by the *SieveQ*.

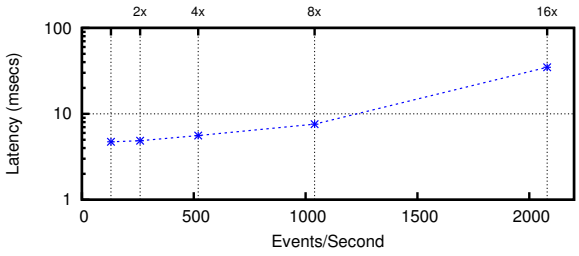


Figure 10: *SieveQ* latency for the 2012 Summer Olympic Games scenario throughput requirement (127 events per second) and how the *SieveQ* scale, we doubled on each experiment the number of events.

(and a standard deviation of 420 bytes). Based on this log, we built a sensor emulator that generates traffic at a pre-defined rate. Basically, when it is time to produce a new event, the emulator selects an event from the trace and feeds it to the *sender-SQ*.

During the 2012 Olympic Games, the workload was approximately 11 million events per day, i.e., around 127 events per second. Figure 10 shows the latency imposed by *SieveQ* for this workload, and when it is scaled up from 2 to 16 times more. As can be observed, the latency is in the order of 4 milliseconds for the emulated scenario. Even with the highest load (16 times) the observed values had a latency below 70 milliseconds. This means that *SieveQ* could potentially deliver on average 176 million events per day, which is more than enough to accommodate the expected growth in the number of events for the next Olympic Games.

7 Discussion

SieveQ differs from standard firewalls like iptables [39] that do not need client- or server-side code modifications. However, our system requires these modifications to ensure end-to-end message integrity and tolerance of compromised components. Complete transparency would be hard to achieve mainly due to the use of voting. Nonetheless, it is worth stressing that *SieveQ* is to be used as an additional protection device in critical systems, not as a substitute to normal L3 firewalls. *SieveQ* provides a protection similar to an application-level/L7 firewall, as one can implement arbitrary rules on the *replica-SQ* module.

State machine replication is a well-known approach for replication [16]. In this technique, every replica is required to process requests in a deterministic way. This requirement traditionally implies in two limitations: (1) replicas cannot use their local clock during request processing; and (2) all requests are executed sequentially. The first limitation can affect the capacity of *SieveQ* to process rules that use time. We remove this limitation by making use of the timestamps generated by the leader replica and agreed upon on each consensus, as proposed in PBFT [20]

and implemented in BFT-SMaRt [21]. The second limitation can constraint the performance of the system, especially when CPU-costly operations such as signature verifications are executed. One of the optimizations we implemented in *SieveQ* was to add multi-threading support in the BFT-SMaRt replicas. More precisely, the signature verification is done by a pool of threads that either accept or discard messages. Once accepted, a message is added to a processing queue following the order established by the total order multicast protocol. A single thread consumes messages from this queue, verifies them against the security policy, updates the firewall state (if needed) and forwards them to their destinations, without violating the determinism requirement.

8 Related Work

Since Castro & Liskov's PBFT [20], several Byzantine fault-tolerant (BFT) message ordering protocols have been described with practical performance in mind (e.g., [31]). This paper does not propose a new BFT replication protocol, but instead uses BFT-SMaRt [21] total order multicast primitive.

Over the past years, there has been an important amount of research in the development of systems that are intrusion tolerant. However, only very little work was devoted to design intrusion-tolerant firewall-like devices. Performance reasons might explain this, as BFT replication protocols are usually associated with reasonable overheads and limited scalability. To our knowledge, only two works can be compared with ours.

Sousa et al. [17] proposed the first replicated intrusion-tolerant system that implements proactive-reactive recovery. The paper presented a firewall for critical services, named CRUTIAL Information Switch (CIS), which was integrated with a trusted component and works under the assumption of an hybrid synchrony model. *SieveQ* shares a few similarities with CIS, but there are two fundamental differences. The CIS needs traffic replicators, one for in-bound and other for out-bound messages, and therefore, an external DoS will be replicated to all replicas; another difference is that the CIS is a stateless firewall.

Roeder and Schneider [18] proposed a replicated intrusion-tolerant device that introduces diversity through software obfuscation techniques on each rejuvenation. The authors main focus was in supporting diversity, a technique that could be integrated into our work. As in [17], the device needs a traffic replicator to send the messages to every replica, and therefore, suffers from the same limitations as the CIS.

Platania et al. [22] did not describe a firewall-like solution, but presented a practical intrusion-tolerant survivable system. Three main contributions came out of this work: a theoretical model that estimates the resiliency of the system based on the rejuvenation rate, the number of replicas and the replica's vulnerability; a protocol for large state transfer between replicas; and a practical replicated system that guarantees a good performance even under attack. Prime [19] was used to support BFT replication, and diversity was implemented by generating different compiled versions of Prime upon recoveries. A trusted component was responsible to trigger the recoveries. This work is complementary to ours, in the sense that it adds diversity and improves the support for proactive recovery, and thus could be used in *SieveQ* by replacing BFT-SMaRt.

There are several approaches to defend systems from DoS/D-DoS (see a survey in [43]). For example, Walfish et al., [44] proposed a solution based on active response to DDoS. Basically, upon the detection of the attack, the server requests the client to send more data. The idea is that by increasing the load, the network congestion management mechanisms will make the channels be used in a more fair way among the correct and incorrect clients. Unfortunately, this solution is not resource efficient because it overloads the channels for the benefit of the client. Jia et al., proposed a solution based on traffic redirection. It works on cloud-based services to solve DDoS. Upon the detection of a DDoS the system redirects the correct client to non-attacked servers [45]. If the network support is available, this kind of technique could be used together with *SieveQ* to ensure *sender-SQs* will always be able to reach a correct *pre-SQ*.

9 Conclusions

We presented *SieveQ*, a new intrusion-tolerant protection system for critical services, such as ICSes and SIEMs. Our system exports a message queue interface which is used by senders and receivers to interact in a regulated way. The main improvement of the *SieveQ* architecture, when compared with previous systems, is the separation of message filtering in several components that carry on verifications progressively more costly and complex. This allows the proposed system to be more efficient than the state-of-the-art replicated firewalls under attack. *SieveQ* also includes several resilience mechanisms that allow the creation, removal and recovery of components in a dynamic way, to effectively respond to evolving threats against the system. Experimental results show that such resilience mechanisms can significantly reduce the effects of DoS attacks against the system.

References

[1] M. Garcia, N. Neves, and A. Bessani, "An intrusion-tolerant firewall design for protecting siem systems," in *Workshop on Systems Resilience in conjunction with the IEEE/IFIP International Conference on Dependable Systems and Networks*, 2013.

[2] K. Ingham and S. Forrest, "A history and survey of network firewalls," University of New Mexico, Computer Science Department, Tech. Rep. TR-CS-2002-37, 2002.

[3] Juniper networks security. [Online]. Available: http://www.juniper.net/us/en/products-services/security/

[4] Palo alto networks. [Online]. Available: http://www.paloaltonetworks.com/products/platforms/PA-5000_Series.html

[5] Sonicwall. [Online]. Available: http://www.sonicwall.com/us/en/products/network-security.html

[6] N. Falliere. (2010, September) Exploring Stuxnet's PLC Infection Process. http://www.symantec.com/connect/blogs/exploring-stuxnet-s-plc-infection-process. Symantec.

[7] Symantec. (2014, July) Dragonfly: Cyberespionage Attacks Against Energy Suppliers. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/Dragonfly_Threat_Against_Western_Energy_Suppliers.pdf. Symantec.

[8] National Vulnerabilities Database. [Online]. Available: https://nvd.nist.gov

[9] J. Chauhan and R. Roy, "Is Firewall and Antivirus Hacker's Best Friend?" iViZ Techno Solutions Pvt Ltd, Tech. Rep., Jan 2011.

[10] R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, 1st ed. John Wiley & Sons, Inc., 2001.

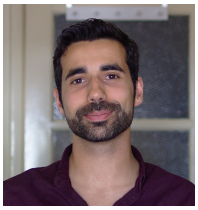
[11] S. Kamara, S. Fahmy, E. Schultz, F. Kerschbaum, and M. Frantzen, "Analysis of vulnerabilities in Internet firewalls," *Computers & Security*, vol. 22, no. 3, pp. 214–232, 2003.

[12] S. Surisetty and S. Kumar, "Is McAfee securitycenter/firewall software providing complete security for your computer?" in *Procs. of the International Conference on Digital Society*, 2010.

[13] Cisco, "Multiple vulnerabilities in firewall services module," Feb. 2007. [Online]. Available: http://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20070214-fwsm

[14] ——, "Multiple vulnerabilities in Cisco firewall services module," Oct. 2012. [Online]. Available: http://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20121010-fwsm

[15] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of ACM*, vol. 32, no. 4, pp. 824–840, Oct. 1985.
[16] F. Schneider, "Implementing faul-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.
[17] P. Sousa, A. Bessani, M. Correia, N. Neves, and P. Veríssimo, "Highly available intrusion-tolerant services with proactive-reactive recovery," *IEEE Transactions on Parallel Distributed Systems*, vol. 21, no. 4, pp. 452–465, Apr. 2010.
[18] T. Roeder and F. Schneider, "Proactive obfuscation," *ACM Transactions on Computer Systems*, vol. 28, no. 2, pp. 4:1–4:54, 2010.
[19] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine replication under attack," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 4, pp. 564–577, 2011.
[20] M. Castro and B. Liskov, "Practical Byzantine fault-tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, Nov. 2002.
[21] A. Bessani, J. Sousa, and E. Alchieri, "State Machine Replication for the Masses with BFT-SMaRt," in *Proc. of International Conference on Dependable Systems and Networks*, 2014.
[22] M. Platania, D. Obenshain, T. Tantillo, R. Sharma, and Y. Amir, "Towards a practical survivable intrusion tolerant replication system," in *Procs. of the IEEE International Symposium on Reliable Distributed Systems*, 2014, pp. 242–252.
[23] A. Mishra, B. Gupta, and R. Joshi, "A comparative study of distributed denial of service attacks, intrusion tolerance and mitigation techniques," in *Procs. of the Intelligence and Security Informatics Conference*, 2011.
[24] I. Gashi, P. Popov, and L. Strigini, "Fault diversity among off-the-shelf sql database servers," in *Procs. of International Conference on Dependable Systems and Networks*, 2004.
[25] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro, "Analysis of operating system diversity for intrusion tolerance," *Software: Practice and Experience*, vol. 44, no. 6, pp. 735–770, 2014.
[26] D. Harkins and D. Carrel. (1998) The Internet Key Exchange. [Online]. Available: http://tools.ietf.org/rfc/rfc2409.txt
[27] D. Kreutz, A. Bessani, E. Feitosa, and H. Cunha, "Towards secure and dependable authentication and authorization infrastructures," in *Procs. of the IEEE Pacific Rim International Symposium on Dependable Computing*, Nov 2014, pp. 43–52.
[28] L. Zhou, F. Schneider, and R. Van Rennesse, "COCA: A secure distributed online certification authority," *ACM Transactions Computer Systems*, vol. 20, no. 4, pp. 329–368, Nov. 2002.
[29] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine fault tolerant systems tolerate Byzantine faults," in *Proc. of the USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2009, pp. 153–168.
[30] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice," Weakdh, Tech. Rep., 05 2015. [Online]. Available: https://weakdh.org/imperfect-forward-secrecy.pdf
[31] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Transactions on Computer Systems*, vol. 27, no. 4, pp. 7:1–7:39, Jan. 2010.
[32] Google. Guava. [Online]. Available: https://code.google.com/p/guava-libraries/
[33] Pyloris. Pyloris. [Online]. Available: https://sourceforge.net/projects/pyloris/
[34] J. Kirsch, S. Goose, Y. Amir, D. Wei, and P. Skare, "Survivable scada via intrusion-tolerant replication," *IEEE Transactions on Smart Grid*, vol. 5, no. 1, pp. 60–70, Jan 2014.
[35] J. Moscola, J. Lockwood, R. Loui, and M. Pachos, "Implementation of a content-scanning module for an internet firewall," in *Procs. of the Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2003, pp. 31–38.
[36] C.-L. Lee, Y.-S. Lin, and Y.-C. Chen, "A Hybrid CPU/GPU Pattern-Matching Algorithm for Deep Packet Inspection," *PLoS ONE*, vol. 10, no. 10, 10 2015.
[37] D. Knuth, J. Morris, and V. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.
[38] K. Prabha and S. Sukumaran, "Improved single keyword pattern matching algorithm for intrusion detection system," *International Journal of Computer Applications*, vol. 90, no. 9, pp. 26–30, March 2014.
[39] Netfilter. (2016) The netfilter.org project. [Online]. Available: http://www.netfilter.org/
[40] A. Bessani, M. Santos, J. Félix, N. Neves, and M. Correia, "On the efficiency of durable state machine replication," in *Procs. of the USENIX Annual Technical Conference*, Jun. 2013.
[41] D. Miller, Z. Payton, A. Harper, C. Blask, and S. VanDyke, *Security Information and Event Management (SIEM) Implementation*. McGraw-Hill Education, 2010.
[42] V. Vianello, V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, R. Torres, R. Diaz, and E. Prieto, "A Scalable SIEM Correlation Engine and Its Application to the Olympic Games IT Infrastructure," *Procs. of the International Conference on Availability, Reliability and Security*, pp. 625–629, 2013.
[43] S. Zargar, J. Joshi, and D. Tipper, "A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks," *Communications Surveys Tutorials, IEEE*, vol. 15, no. 4, pp. 2046–2069, 2013.
[44] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker, "DDoS Defense by Offense," in *ACM Transactions on Computer Systems*, March 2010.
[45] Q. Jia, H. Wang, D. Fleck, F. Li, A. Stavrou, and W. Powell, "Catch Me If You Can: A Cloud-Enabled DDoS Defense," in *Procs. of International Conference on Dependable Systems and Networks*, June 2014.



Miguel Garcia is a PhD student of the Department of Computer Science of the Faculty of Sciences of the University of Lisboa, Portugal, and a member of LaSIGE research unit and the Navigators research team. He received his M.Sc in 2011 by the University of Lisbon. His main research interests are dependable and intrusion-tolerant systems. More information at http://homepages.lasige.di.fc.ul.pt/~mhenriques

Nuno Neves is an Associate Professor with Habilitation of the Department of Computer Science of the Faculty of Sciences of the University of Lisboa, Portugal, and a member of LaSIGE research unit and the Navigators research team. His research interests are in security and dependability aspects of distributed systems. Over the years, he participated in several European and national research projects with a focus on security, such as SUPERCLOUD and SEGRID. His work has been recognized several times, for example with the IBM Scientific Prize and the William C. Carter Award at the IEEE FTCS. More information at http://www.di.fc.ul.pt/~nuno

Alysson Bessani is an Associate Professor of the Department of Computer Science of the Faculty of Sciences of the University of Lisboa, Portugal, and a member of LaSIGE research unit and the Navigators research team. He holds a PhD in Electrical Engineering from Santa Catarina Federal University, Brazil (2006), and was a visiting professor at Carnegie Mellow University (2010), and a visiting researcher at Microsoft Research Cambridge (2014). His main interests are distributed algorithms, Byzantine fault tolerance, coordination, and security monitoring. More information at http://www.di.fc.ul.pt/~bessani