Computer Networks 54 (2010) 2711–2727

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Defense techniques for low-rate DoS attacks against application servers

Gabriel Maciá-Fernández [*], Rafael A. Rodríguez-Gómez, Jesús E. Díaz-Verdejo

Dept. of Signal Theory, Telematics and Communications, E.T.S. Computer and Telecommunications Engineering, University of Granada, c/Daniel Saucedo Aranda, s/n, 18071 Granada, Spain

A R T I C L E I N F O

Article history:
Received 14 October 2009
Received in revised form 15 February 2010
Accepted 4 May 2010
Available online 11 May 2010
Responsible Editor: Christos Douligeris

Keywords:
Denial of service
Low-rate
Defense
Network security

A B S T R A C T

Low-rate denial of service (DoS) attacks have recently emerged as new strategies for denying networking services. Such attacks are capable of discovering vulnerabilities in protocols or applications behavior to carry out a DoS with low-rate traffic. In this paper, we focus on a specific attack: the low-rate DoS attack against application servers, and address the task of finding an effective defense against this attack.

Different approaches are explored and four alternatives to defeat these attacks are suggested. The techniques proposed are based on modifying the way in which an application server accepts incoming requests. They focus on protective measures aimed at (*i*) preventing an attacker from capturing all the positions in the incoming queues of applications, and (*ii*) randomizing the server operation to eliminate possible vulnerabilities due to predictable behaviors.

We extensively describe the suggested techniques, discussing the benefits and drawbacks for each under two criteria: the attack efficiency reduction obtained, and the impact on the normal operation of the server. We evaluate the proposed solutions in a both a simulated and a real environment, and provide guidelines for their implementation in a production system.

1. Introduction

Denial of service (*DoS*) currently poses a severe challenge for researchers. DoS attacks are those which aim at either completely or partially disrupting the availability of a system or network. Traditionally, these attacks have been classified into two categories [1]: (*i*) flooding attacks, *i.e.*, those which send a huge amount of traffic to a victim in order to overwhelm its resources, and (*ii*) vulnerability attacks, *i.e.*, those which take advantage of a certain vulnerability in the victim and send specially crafted messages which cause the denial of service.

Flooding attacks are difficult to counteract, mainly due to their strength and because they are typically launched from a large number of distributed locations (DDoS), what allows the attacker to use IP spoofing and other sophisticated evading techniques. However, this fact also constitutes a certain weakness for the attack, as high-rate traffic is more detectable and also the attacker needs to recruit a high number of zombies to deliver the attack [2].

For this reason, attackers have recently moved to novel strategies that allow them to launch flooding DoS attacks without sending high-rate traffic to the victims. These are mixed approaches between flooding and vulnerability attacks in the sense that they get advantage of a certain vulnerability to reduce the traffic rate directed to the victim. Some authors have warned that these attacks constitute a considerable danger and that they may affect to important Internet services [3].

In this respect, several kinds of low-rate DoS attacks have been reported by researchers. The pioneer was the Shrew attack against TCP [4]. The attacker sends a burst of well-timed packets, creating packet losses in a link and

[*] Corresponding author. Tel.: +34 958241000/20048; fax: +34 958240831.
E-mail addresses: gmacia@ugr.es (G. Maciá-Fernández), rodgom@correo.ugr.es (R.A. Rodríguez-Gómez), jedv@ugr.es (J.E. Díaz-Verdejo).

1389-1286/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2010.05.002

therefore incrementing the retransmission timeout for certain TCP flows. The bursts are sent only around the expiration times of these flows, thus reducing the overall traffic rate employed by the attacker.

Another family of low-rate DoS attacks is constituted of reduction of quality (RoQ) attacks [5], which attempt to degrade the performance of a victim by disrupting the feedback mechanism of a control system. This is done by intelligently sending short bursts of traffic in such a way that the own control system of the target is fooled and causes the denial of service. These attacks have been studied in several scenarios, such as bottleneck queues with Active Queue Management (AQM) employing Random Early Detection (RED) [5], Internet end systems [6], dynamic load balancers [7], ad hoc networks [8] and content adaptation controllers [9].

More recently, we have reported [10] another kind of such attacks: low-rate DoS attacks against application servers (henceforth LoRDAS). A LoRDAS attack is launched against application servers in Internet and is able to reduce their availability in a controlled way by sending short bursts of traffic. These bursts are sent only around specific instants, thus resulting in low-rate traffic. The efficiency of this attack has been verified when it is applied to iterative servers [11], *i.e.*, those running a single thread or process, as well as to concurrent servers [10].

The research community has invested considerable effort in the development of detection and counteracting methods against Shrew and RoQ attacks (see details in Section 2). However, to the best of our knowledge, no solutions have yet been suggested to defeat LoRDAS attacks. In this paper, we focus on this problem and suggest a defense strategy against these attacks. We leverage mechanisms based on specific management of incoming queues to applications and suggest a solution based on slightly modifying the server behavior under attack conditions. A set of different alternatives for the solution is studied, showing both the benefits and the drawbacks of each one. These alternatives are also evaluated in both a simulated and in a real environment. Finally, guidelines for their implementation in a production environment are given.

The paper is structured as follows: Section 2 summarizes related work on suggested techniques to defeat low-rate DoS attacks, as well as existing approaches to defeat DoS attacks against application servers. Next, some fundamentals about the LoRDAS attack are explained in Section 3, while different alternatives for the defense are described in detail in Section 4. Results derived from our evaluation of the suggested techniques are shown in Section 5. In Section 6, these results are also validated in a real environment, and are subsequently discussed in Section 7. Finally, some conclusions are drawn in Section 8.

2. Related work

In order to build defense strategies against LoRDAS attacks, let us first pay special attention to techniques which have been formerly employed to combat other kinds of low-rate attacks. Although a general technique for all these attacks would be desirable, at this moment only approaches designed to defeat specific attacks have been proposed.

Next, we present related work on defenses against both Shrew and RoQ attacks. In addition, we consider other suggested solutions to defend applications against DoS attacks.

2.1. Defenses against Shrew attacks

Due to the importance of these attacks, many authors have concentrated on leveraging mechanisms to defeat these attacks. The main difference between Shrew attacks and LoRDAS consists on the fact that Shrew is TCP targeted, whilst LoRDAS works at the application level. Moreover, they exploit different vulnerabilities, i.e., determinism in the TCP congestion control mechanism timeout (Shrew) and existence of deterministic service times in applications (LoRDAS). Furthermore, Shrew attack attempts to trigger the TCP congestion control mechanism by creating outages in a link, while LoRDAS simply seeks to overflow a service running in a machine, not creating any network congestion at all.

Regarding the defense mechanisms for Shrew attacks, some authors are only interested in detection while others also develop mitigation mechanisms. Sun et al. [12] suggest that the ON/OFF traffic pattern of the attack can be detected by combining the autocorrelation of the traffic rate signal and the use of dynamic time warping (DTW) [13]. They also suggest the use of the same technique in a distributed environment [14].

Others try to detect the attack by analyzing the frequency information of traffic flows, either by spectral analysis [15] or by considering the correlation between the ON/OFF traffic rate signal and the round trip time of the affected flows [16].

In this paper, we focus not only on detecting attacks, but also on developing a response technique that aims at reducing the attack impact. In this line, some previous work for Shrew attacks has been carried out. Yang et al. [17] suggested to randomize RTO[1] timers in TCP flows in order to avoid the synchronization between the periodic arrival of short attack bursts and the timer expiration. As we show later on, some of the mechanisms suggested in this paper for LoRDAS attacks are inspired in the latter idea. The main flaws of this approach are the impact on the protocol performance and the need for modifying the protocol stack, which implies a slow deployment. Sarat et al. [18] studied the evolution of router buffer sizes during the execution of Shrew attacks; they demonstrated that an increase in the router queues sizes would compel the attacker to use high-rate traffic. As shown below, this work has similarities with ours in the sense that it suggests the modification of the queues. Still, there are important differences. First, it focuses on router queues, while ours works with application incoming queues. Second, we modify the queue management process instead of simply increasing a queue size, as this would not be enough to constitute a solution for the LoRDAS attack.

[1] Retransmission TimeOut of TCP.

2.2. Defenses against RoQ attacks

RoQ attacks have also attracted attention from the research community, which has made a big effort to defeat them. The main difference between these attacks and LoRDAS is that RoQ attacks get advantage of the transient mechanisms of the systems, whilst LoRDAS uses an estimation of the service time for the requests employed by an application server. Furthermore, the way that both attacks exploit the vulnerability are different. RoQ attacks send short high-rate traffic bursts to the victim while attack bursts in LoRDAS practically consist of less than five attack packets.

Some strategies have been learnt and are applied here from the previous Shrew attack experience. Chen and Hwang [19] suggested the use of spectral analysis techniques to detect these attacks. Shevtekar and Ansari [20] proposed a router-based technique for detection, which consists in measuring traffic increments during short periods of time and comparing these increments with a threshold. Another approach has been contributed by Argyraki et al. for the detection of RoQ attacks in ad hoc networks [21]. These authors analyse the frequency of RTS/CTS messages in the virtual carrier sensing process to detect a signature that identifies RoQ attacks.

2.3. Defenses against DoS directed against applications

Some authors have addressed the problem of protective applications from the threat of DoS attacks. Srivatsa et al. [22] suggest a set of protecting measures for web servers, mainly based on a congestion control system that uses port hiding and client prioritization. This system focuses on DoS attacks in general and does not consider the low-rate type as a specific case. Ranjan et al. [23] recently presented a solution that analyzes the traffic directed to an application and applies certain policies to the incoming traffic. There are similarities with our work in the sense that given policies are enforced by applying an access control mechanism in the incoming queues. The main difference between our approach and that in [23] is that the latter is not specifically designed for low-rate traffic. While it suggests a detection process based on traffic rate measurements, we detect attack requests by considering temporal and spatial similarities between them. Besides, the solution suggested in [23] is implemented in a proxy, while our proposal is located at the servers themselves. This implies that the two solutions are complementary and may contribute to the concept of security in depth, thus allowing security officers to deal with both external attackers (accessing through the proxy) and insider attackers. Finally, what we suggest is a lightweight alternative which avoids the need to compute of a complex detection algorithm and the need for communication between the application and a proxy as in [23].

3. LoRDAS attack fundamentals

Application servers are the potential victims of LoRDAS attacks. Certain conditions are required for an application to be vulnerable to this kind of attacks, and several different strategies might be followed by the attacker to deny the service. Here, we present the fundamentals of the behavior of application servers, and the overall process followed to deliver a LoRDAS attack. We focus only on the necessary details for understanding the defense techniques explained in Section 4, referring to [11,10], for more details.

The application server model considered in the LoRDAS attack is composed of the following elements (Fig. 1): (*i*) a *service queue* where incoming requests are placed upon their arrival at the server, and (*ii*) one or several *service modules* which are in charge of processing the requests.

The service queue is a finite-length queue where requests are stored. When the queue is full, any new incoming request is rejected (event message overflow MO in Fig. 1). The service modules are really processes or threads that attend those requests queued in the service queue. Requests are extracted from the service queue following a certain queue serving policy, *e.g.*, FIFO, LIFO, etc. Once a service module has finished processing a request, an *answer* is sent to the corresponding client. This is termed an *answer instant.* Note that when an answer is sent by the server, the corresponding service module becomes idle. Then, if any pending request is waiting in the service queue, it is extracted, thus freeing a new position in the queue. We denote the instants at which new positions are enabled in the service queue as *enabling instants*. At first sight, it could seem obvious that *answer instants* and *enabling instants* occur at the same time and, thus, they could be defined as a single concept. However, we prefer to keep these two concepts separate, as they will play different roles when defense techniques are applied.

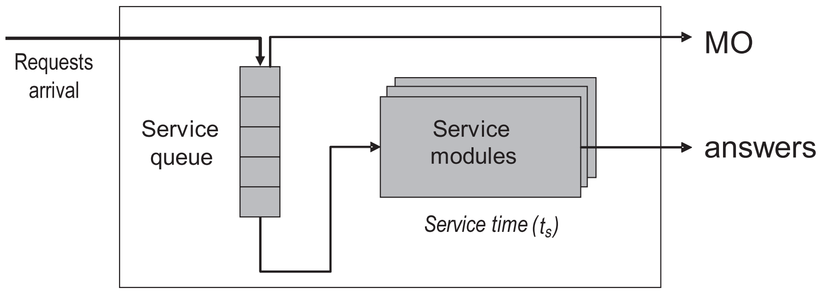


Fig. 1. Application server model in the LoRDAS attack.

From a request perspective, let us define two concepts for subsequent use in the evaluation of defense techniques, as detailed later. Let the *service time,* t_s, be that employed by the service module in processing a given request. As discussed in [10], even if identical requests are considered, the service time for their processing might be different, as several multiple factors influence this time, *e.g.*, memory and disk use, number of CPU interruptions, network traffic, etc. For this reason, the service time for identical requests is modeled in [10] as a random variable, T_S, with a normal distribution. Its mean value is denoted as $\overline{T}_S$, and its variance as $var[T_S]$.

$$T_S = \mathcal{N}(\overline{T_S}, var[T_S]). \tag{1}$$

Let the *in-system time*for a request, t_I, be the time elapsed between its arrival and the generation of the corresponding answer. The *in-system time* is contributed by the time spent by a request while in the service queue, plus the corresponding service time. Let us consider an example consisting of a server with only one service module, *i.e.*, a single process or thread, and a service queue of N positions. Let us assume also that the queue contains $N-1$ requests, and that they are all processed by the service module at a rate of t_s seconds per request. If we evaluate t_I for a new incoming request in this scenario, an interval of possible values is obtained, depending on the amount of time still needed to serve the request located in the service module. This interval is given by:

$$t_I = [Nt_s,\ (N+1)t_s]. \tag{2}$$

The aim of the LoRDAS attack is to send traffic in such a way that it makes the server process only those requests coming from the attacker instead of from legitimate users. This is similar to preventing legitimate users from storing any request in the service queue of the server by keeping it completely full of requests coming from the attacker. In a traditional approach, this could be done by using high rate traffic (flooding DoS attack). However, what the attacker actually does here is to estimate those instants at which free positions are enabled in the service queue (*enabling instants*), and to send traffic only around them. Following this strategy, the amount of traffic used in delivering the attack is considerably reduced (low-rate DoS attack). An overview of the complete process for carrying out the attack is now described.

As a previous step to carrying out the attack, the LoRDAS attacker estimates the service time for one or several types of identical requests (see more details below). Then, using this knowledge, the attacker is able to forecast the *answer instants* at the server, which enables the attacker to know also the *enabling instants*, as they coincide. We provide in the following an example on how this is really done, and refer to [10,11] for more details about different strategies for carrying out this preliminary step.

Consider a web server that implements the persistent connections feature. First, the attacker sends an HTTP request, which is stored in the service queue. Then, when reaching the service module after a queue time, the server sends an HTTP response containing the requested resource. After that, the server waits for the expiration of a persistent connection timeout before shutting down the connection. This timeout is typically a preconfigured fixed value in the server[2] which stands for every TCP connection, no matter what resource is requested. In this process, it is easy for the attacker to estimate the time elapsed between the HTTP response and the connection shutdown, just by sending several requests to probe the server. This time plays the role of the *service time,* t_s, in the LoRDAS attack server model. Next, by using this knowledge, the attacker estimates the *answer instants*. In this example, the *answer instants* are those at which the TCP connections are closed. Considering the functioning of the server, an *answer instant* will occur t_s seconds after an HTTP response is sent. As mentioned previously, this instant is also an *enabling instant*, as the next request waiting in the service queue will be extracted by the recently idle service module.

After forecasting the answer instants, the attacker is able to send traffic in such a way that it intentionally arrives at the server around the estimated *enabling instants*. As previously discussed, although certain requests have a constant service time, *i.e.*, persistent connection timeout in our example, the attacker would perceive this time as a normal distribution variable (Expression (1)). For this reason, the attacker sends not only one request that arrives just after every *enabling instant*, but short bursts of traffic that arrive around the forecasted instants to avoid failing to seize the enabled positions in the queue due to the variance of the service time.

Additionally, the LoRDAS attack incorporates another mechanism for dealing with the possibility that short bursts of traffic may not succeed in seizing the enabled queue free positions. Every time that a new answer, *i.e.*, a connection shutdown in our example, is received from the server, a new attack request is sent to the server. Fig. 2 shows the whole process for acquiring a position in the queue once an *enabling instant* is estimated. First, a short burst of three attack packets is sent to the server. Note that there is a difference between the estimated *enabling instant* (around which the attack burst is centered) and the real answer instant. This difference is due to the fact that t_s really follows a normal distribution as in Expression (1). In the figure, the second packet in the short burst of attack traffic will occupy the free position in the queue. Still, when the answer reaches the attacker, a new attack request is sent to the server.

The LoRDAS attack is designed in such a way that multiple attack threads are executed at the same time. Every attack thread tries to ensure, by sending attack requests, that at least one request is always present in the service queue. This means that, if the attacker would like to completely deny the service, the number of threads on the attacker side should correspond with the number of attacked positions in the service queue. Note that the attacker might not only to completely deny the service, but also to partially reduce the availability of the server, *i.e.*, by dimensioning a number of attack threads lower than the size of the service queue, and also to strike the attack from distributed locations.

[2] In an Apache 2.0 server, the directive KeepAliveTimeout controls this timeout.

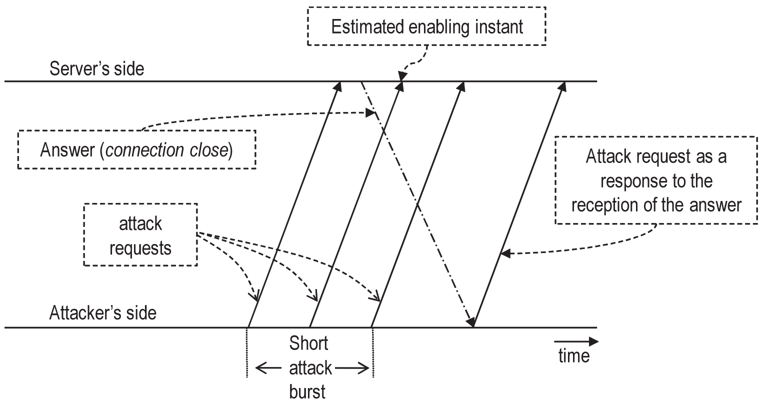

Estimated enabling instant
Server's side
Answer (connection close)
Attack request as a response to the reception of the answer
attack requests
Attacker's side
Short attack burst
time

Fig. 2. LoRDAS attack process for seizing a position in the service queue once an enabling instant has been estimated.

Once an attack thread has inserted a request into the service queue, the process described above can be executed. Thus, the insertion of a request is a pre-requisite. For this reason, whenever an attack thread has not completed this pre-requisite (possibly because a legitimate user has does so instead of the attack thread), the attack thread tries blindly to seize a new position. At this stage, we say that the attack thread is in the *flooding state*.

A good defense technique should try to prevent the attacker from gaining any advantage over legitimate users. Thus, the aim is to make all the attack threads go into the flooding state. In this state, the probability of an attacker inserting a request into the service queue depends only on its traffic rate and not on the attacker's ability to exploit a vulnerability. Obviously, in this situation a low-rate traffic is not enough for the attacker.

4. Proposals on defenses against the LoRDAS attack

As explained above, the success of a LoRDAS attack is based on the exploitation of two different aspects in server behavior: (*i*) the existence of deterministic patterns, *e.g.*, fixed timeouts, which makes it possible to estimate the instants at which answers are sent to the corresponding clients (*answer instants*) and (*ii*) the fact that *enabling instants* concur at the same time as *answer instants*, *i.e.*, positions in the service queue are enabled just at the instants at which the answers are sent. As a consequence, the design of defense techniques against LoRDAS attacks should be based on trying to modify the server behavior in such a way that these two aspects are not exploitable.

For the development of such defense techniques we focus on achieving the following two goals:

- The efficiency of the attack should be reduced as much as possible. We define the *efficiency of the attack*, *E*, as the percentage of service queue positions seized by the attacker over the total number of seized positions during the attack execution. Note that we do not consider a measure of the efficacy of the defense technique based on the observation of the percentage of served requests coming from legitimate users. The reason is that this latter metric depends on the legitimate users' traffic. Our metric measures the percentage of the server capacity that is dedicated to serve attack requests. The remaining capacity is therefore used for serving legitimate users' requests.
- The impact of the defense technique on the normal behavior of the server under no attack conditions should be minimized. We assess this impact by comparing the *in-time system* experienced by requests when none of the defenses is active (Expression (2)), and when the different defense techniques are applied.

In the following, we provide some approaches that constitute defense techniques to reduce LoRDAS attack efficiency. They are presented following an incremental approach, in which every one produces additional benefits, compared to the previously presented techniques. For each one, a motivation of its strategy is explained, the technique itself is described in detail and both the benefits and the shortcomings are discussed. Subsequently, an evaluation study of these techniques in an experimental environment is presented in Section 5.

For all the techniques presented, we assume that the server is in an overflow state, *i.e.*, the service queue is completely full of requests. In Section 7 we discuss the use of these techniques in other scenarios.

4.1. Random service time (RST)

The *Random service time* defense, *RST*, is designed to reduce the predictability of server behavior. This is done by eliminating deterministic patterns from its mode of operation. Whenever a fixed timeout or other exploitable feature is used in the server, this technique aims at randomizing it, in order to make things harder for the attacker, *i.e.*, it hinders the prediction of the *answer instants* and, consequently, also of the *enabling instants*.

A server implementing *RST* works as in the following (see attack process in Fig. 3):

(1) *Service of a request in the queue.* When idle, a service module takes a request from the service queue following an established scheme, *e.g.*, FIFO, LIFO, etc. Then, the service module employs t_s seconds to

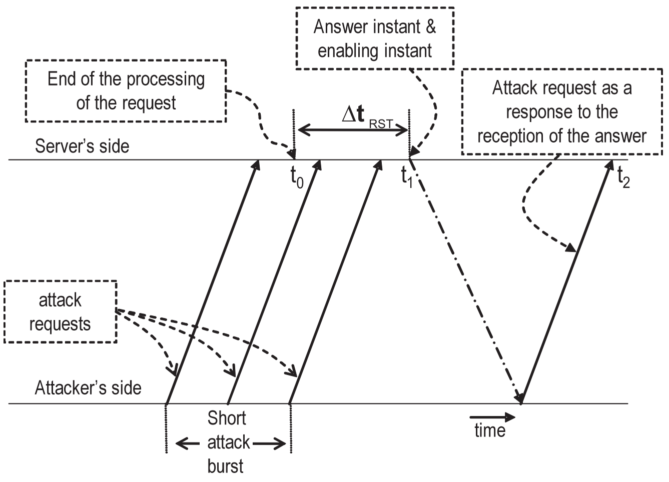

Answer instant & enabling instant
End of the processing of the request
Δt RST
Attack request as a response to the reception of the answer
Server's side
t0
t1
t2
attack requests
Attacker's side
Short attack burst
time

Fig. 3. LoRDAS attack process for seizing a position in the service queue when *RST* defense is active.

process the request. During this phase, the attacker manages to send a short attack burst in such a way that it arrives around the estimated *answer time*.

(2) *Extra delay.* When the processing is finished (t_0 in Fig. 3), the service module remains locked during an additional random time, called *extra delay*, Δt_{RST}. In this case, unlike the normal behavior (depicted in Fig. 2), no additional position is enabled in the service queue during Δt_{RST}, as no answer is generated.

(3) *Sending of the answer.* After the extra delay (t_1 in Fig. 3), the answer is sent to the user that previously requested the service. Due to the functioning of the server, a new free position appears in the service queue just at this instant, thus concurring both the *enabling instant* and the *answer instant*. If Δt_{RST} is long enough, as in the figure (we make this assumption now and discuss other possibilities later), during the period of time from t_1 until t_2, *i.e.*, a period of duration *RTT* (round trip time) seconds, a legitimate user can insert a request in the queue. In t_2, a new attack request arrives and, if the position still remains free, it is occupied.

With respect to the efficiency of the attack, the application of *RST* in the server has two remarkable consequences: first, the *answer instant* is moved to a position not controlled by the attacker; second, an additional variance, compared to that originally perceived by the attacker, is introduced. Let us study both of them in greater detail.

(*i*) First, *RST* shifts the answer time to a position that is not controlled by the attacker. If the extra delay is long enough (as in Fig. 3) and the service queue is full of requests during the attack burst, then all the attack burst traffic will be rejected by the server. In this case, the period of time available for a legitimate user to insert a new request in the queue would correspond to the round trip time between the server and the attacker, *RTT*, *i.e.*, the period of time between the answer instant (t_1 in Fig. 3) and the reception of the attack packet sent as a response to the answer (t_2 in Fig. 3). As shown, it is desirable that *RST* should shift the answer time out of the attack burst arrival in such a way that, during *RTT* seconds, a new position in the queue is available for legitimate users.

However, when an extra delay is added to the original service time t_s, the attacker also perceives an increase in the estimation of t_s equal to the mean extra delay, $\overline{\Delta t}_{RST}$, and, thus, will adjust the attack parameters[3] in order to synchronize the attack bursts in such a way that they arrive around $\overline{T_s} + \overline{\Delta t}_{RST}$. Considering this attack adaptation process, when bursts of B seconds arrive from the attacker, *RST* should ideally be adjusted in such a way that the answer time is shifted to either just after the attack bursts arrive (*condition 1:* $\Delta t > \overline{\Delta t}_{RST} + B/2$) or *RTT* seconds before (*condition 2:* $\Delta t < \overline{\Delta t}_{RST} - B/2 - RTT$). Only in these cases are we assured that legitimate users will have *RTT* seconds for inserting requests in the queue. For this reason, assuming that the server is able to estimate the values for *RTT* of the attacker and B (we discuss how this is done in Section 5.3), Δt is calculated as follows.

Condition 2 cannot be fulfilled if $\overline{\Delta t}_{RST} < B/2 + RTT$. In this case, Δt_{RST} takes values from a uniform distribution with a maximum value Δt^{RST}_{max}:

$$\Delta t = U\left[0, \Delta t^{RST}_{max}\right] \quad \text{if } \overline{\Delta t}_{RST} < \frac{B}{2} + RTT \tag{3}$$

and only in this case will legitimate users not have free positions available for *RTT* seconds but only for $\overline{\Delta t}_{RST}$ seconds.

When $\overline{\Delta t}_{RST} > B/2 + RTT$, Δt_{RST} is a random variable sampled from two different uniform variables, $V_1 = U[0, \Delta t_1]$ and $V_2 = U[\Delta t_2, \Delta t^{RST}_{max}]$ (see Fig. 4), in such a way that

$$\begin{aligned} \overline{\Delta t} &= \frac{\Delta t^{RST}_{max}}{2}, \\ \Delta t_1 &= max\left[0, \overline{\Delta t}_{RST} - B/2 - RTT\right], \\ \Delta t_2 &= \overline{\Delta t}_{RST} + B/2. \end{aligned} \tag{4}$$

[3] The details for this adaptive process are given in [11].

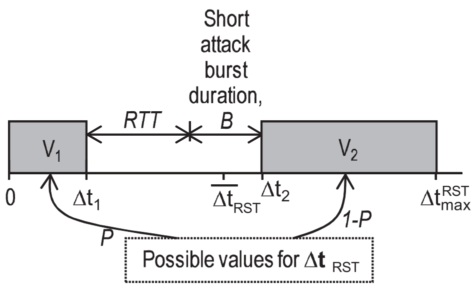


Fig. 4. Intervals in the uniform distribution for $\overline{\Delta t}$.

Note that the value of the mean extra delay should be $\overline{\Delta t}_{RST} = \Delta t_{max}^{RST}/2$ in order to appropriately shift the answer time. For this reason, Δt_{RST} is really sampled from the variables V_1 and V_2 following a probability P in such a way that

$$\Delta t_{RST} = \begin{cases} V_1 & \text{with prob. } P \\ V_2 & \text{with prob. } 1-P \end{cases} \quad \text{if} \overline{\Delta t}_{RST} > B/2 + RTT, \tag{5}$$

where the probability P is calculated from the following condition:

$$\overline{\Delta t}_{RST} = P \cdot E[V_1] + (1-P) \cdot E[V_2] = \frac{\Delta t_{max}^{RST}}{2}, \tag{6}$$

which leads to

$$P = \frac{\Delta t_{max}^{RST} + B}{2(\Delta t_{max}^{RST} + B + RTT)}. \tag{7}$$

(ii) Second, *RST* introduces an extra variance to that originally perceived by the attacker, $var[t_s]$. This variance is given by the implicit variabilility in the extra delay, $var[\Delta t_{RST}]$. For this reason, it is expected that the attack will be less efficient. The only option for the attacker to avoid the effect of this extra variability is to enlarge the duration of the attack bursts, which would not be desirable, especially when the traffic rate needs to be low in order to hide the attack.

In summary, it could be said that *RST* is able to decrease the attack efficiency mainly because it shifts the answer time to a position not controlled by the attacker, and because it adds a source of variability in the server behavior.

With respect to the impact of *RST* on the normal behavior of the server, a notable shortcoming appears when this defense is employed: the *in-system time* increases for every request when *RST* is active. Let us evaluate t_I in the same example considered in Expression (2), *i.e.*, a server with a single service module, $N-1$ equal requests in the queue and another one in the service module, all of them involving a service time t_s. In this case, t_s is incremented by the extra delay for all the requests, thus causing a delay to the other requests waiting in the queue. Taking a mean value $\overline{\Delta t}_{RST}$ for all the requests, t_I becomes

$$t_I^{RST} = [N \cdot (t_s + \Delta t_{RST}),\ (N+1) \cdot (t_s + \Delta t_{RST})], \tag{8}$$

which means that a minimum increase of $N \cdot \Delta t_{RST}$ is experienced in t_I for every request. Obviously, a selection of a high value for Δt_{max}^{RST} implies that *RST* is more efficient in defending against the attack but, as shown, it also implies a higher impact on the service. In Section 5, an evaluation of this configuration value is made.

Finally, note that the configuration value for Δt_{RST}, given by Expressions (3) and (5) depends on the value of *RTT*. Obviously, this value vary between each incoming request to the server. Here, we have assumed a constant value for *RTT*; in Section 7 we discuss the configuration of Δt_{RST} in a realistic scenario where different *RTT* values are present.

4.2. Random answer instant (RAI)

A second option for deploying defense schemes, called *Random answer instant*, *RAI*, is now presented. In this technique, instead of introducing variability in the server behavior as in the *RST* defense, we explore how to defeat the attack by decoupling the *answer instants* and the *enabling instants*. As the attacker sends attack bursts in such a way that they arrive around the *answer instants*, a countermeasure based on forcing the *enabling instants* to happen at a different time will make the attacker fail.

This is done by following these steps (see Fig. 5):

(1) *Service of a request.* As in *RST*, the service module extracts a request from the service queue and processes it during a service time, t_s.
(2) *Non-blocking extra delay.* When the processing is finished (t_0 in Fig. 5), the service module waits for an additional random time (extra delay), before sending the response. However, as a key difference with *RST*, when the service has finished, the service module extracts a new request from the service queue and begins its processing. Thus, we say that the extra delay is non blocking. A new position is enabled in the queue at this instant; therefore, t_0 becomes the *enabling instant*.
(3) *Delayed response sending.* When the extra delay expires, *i.e.*, at $t_0 + \Delta t_{RAI}$, the answer is sent to the corresponding client. This is the *answer instant*, which now differs from the *enabling instant*.

Note in this process that the attacker is estimating a service time given by $\overline{T_s} + \overline{\Delta t}_{RAI}$ and, thus, is sending a short attack burst that arrives around this instant. However, the *enabling instant* is happening before the *answer instant*, so that legitimate users have more time to insert new requests in the queue. In the example in Fig. 5, a new position is available for requests coming from legitimate users during the interval delimited by the enabling instant and the arrival of the first attack packet of the burst, that is, $t_1 - t_0$.

The selection of the value for Δt_{RAI} is now discussed. As previously indicated, it is desirable to make the attacker perceive an increased variability in the service time; for this reason, Δt_{RAI} is chosen from a uniform distribution, $\Delta t = U[\Delta t_{min}^{RAI}, \Delta t_{max}^{RAI}]$. The lower limit of the distribution, Δt_{min}^{RAI}, is selected in such a way that the difference between the enabling instant and the answer instant is higher than half of the attack burst length, *i.e.*, $\Delta t_{min}^{RAI} = B/2$. Thus, we ensure that there is a time interval of duration $\overline{\Delta t}_{RAI} - \frac{B}{2}$ between the enabling instant and the arrival of the short attack burst. In summary:

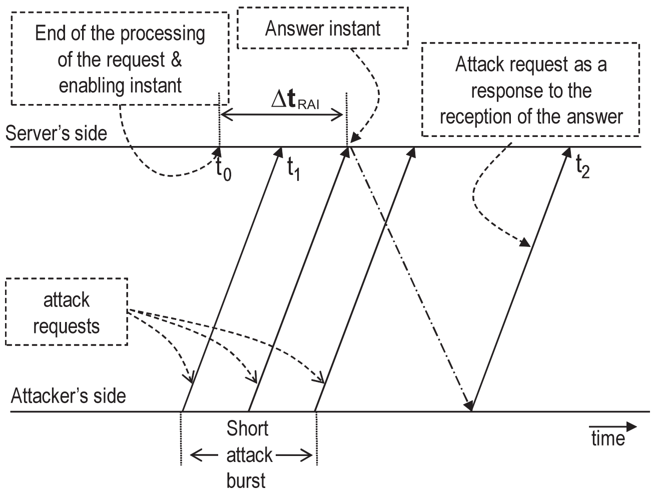

End of the processing of the request & enabling instant
Answer instant
Attack request as a response to the reception of the answer
Δt_{RAI}
Server's side
t_0
t_1
t_2
attack requests
Attacker's side
Short attack burst
time

Fig. 5. LoRDAS attack process for seizing a position in the service queue when RAI defense is active.

$$\Delta t = \begin{cases} 0 & \text{if} \Delta t_{max}^{RAI} < \frac{B}{2} \\ U\left[\frac{B}{2}, \Delta t_{max}^{RAI}\right] & \text{if} \Delta t_{max}^{RAI} \geqslant \frac{B}{2}. \end{cases}$$

As a first approach, the upper limit value, Δt_{max}^{RAI}, should be as high as possible in order to introduce higher variability. However, if we analyze the impact of this measure by evaluating t_I, we obtain an interval of possible values given by

$$t_I^{RAI} = [N \cdot t_s + \Delta t_{RAI}, (N+1) \cdot t_s + \Delta t_{RAI}], \qquad (9)$$

which is really an increment of Δt_{RAI} with respect to the normal behavior given by Expression (2). This means that a higher value for Δt_{max}^{RAI} also generates a higher impact in the normal behavior. Really, Δt_{max}^{RAI} should be configured as a trade-off between reducing the impact and increasing the variability in the server. In Section 5, we evaluate configuration values for this parameter in a wide range of scenarios.

In summary, the main contribution of this technique, compared to *RST*, is the reduction of the impact on the normal behavior of the server, while the effectiveness of the attack is reduced. In Section 5 we provide a comparative study of both the effectiveness and the impact of these techniques.

4.3. Random time queue blocking (RTQB)

As shown before, *RST* and *RAI* are techniques for reducing the effectiveness of the attack. However, they present some limitations. First, the reduction of effectiveness in *RST* is limited due to the fact that the maximum amount of time for legitimate users to seize new positions in the queue is *RTT*. *RAI* does not present this limit but, like *RST*, it implies the cost of introducing an impact on the server behavior. Although the impact could be acceptable, especially if we consider that the server is potentially under an attack or, at least, under overflow conditions, it would be desirable for a defense technique not to cause any impact.

Random time queue blocking, *RTQB*, is a defense technique that aims at reducing the attack efficiency whilst causing no impact on the server behavior. The basic idea behind *RTQB* is as follows. Let us suppose that the attacker is able to accurately estimate the *answer instants* at the server. Then, the short attack bursts will arrive around these instants. If the answers are really sent at these *answer instants*, a response attack message will arrive in *RTT* seconds. In this scenario, while legitimate users are distributing their requests (generally as a poisson process) during this time, the attacker is really concentrating the requests in a time interval $[-B/2, RTT]$ around the *answer instants*, B being the size of the attack burst. Thus, the idea here is to block all the requests that arrive at the server during this time interval.

Note that *RTQB*, as described here, can be thought of as a mechanism that generates DoS itself. Yet, if *RTQB* is not active, DoS intervals are not controlled by the server, while applying *DoS* intervals are concentrated around *answer instants*, thus making the attacker more penalized.

The execution of *RTQB* is done following these steps (see Fig. 6):

(1) *Service of a request.* The service module extracts a request from the service queue and processes it during a service time, t_s.
(2) *Answer generation.* When the processing is finished, the service module sends the corresponding answer (*answer instant*), and extracts a new request from the service queue (*enabling instant*).
(3) *Blocking time.* After the *answer instant*, new incoming requests are rejected during a time interval Δt_{RTQB}. After this time, new incoming requests are accepted again.

Note that, although in this process the *answer instants* and the *enabling instants* may seem to happen at the same time (step 2 of the process), the existence of a blocking time just after the *answer instant* virtually shifts the *enabling instant*, as no new requests are allowed to enter

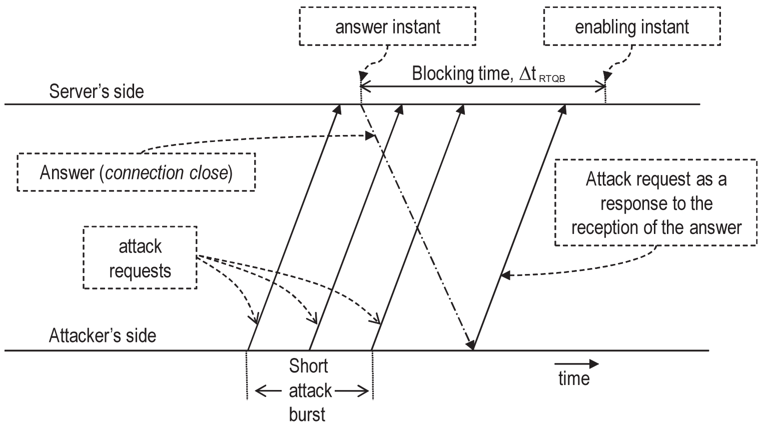


Fig. 6. LoRDAS attack process for seizing a position in the service queue when *RTQB* defense is active.

the queue before Δt_{RTQB} finishes. Thus, it may be said that this technique is based on decoupling *answer instants* from *enabling instants*.

Ideally, the value for Δt_{RTQB} should be *RTT*, as all the attack requests are expected in an interval $[-B/2, RTT]$ around the *answer instant*. However, we suggest configuring $\Delta\ t_{RTQB}$ as a random value taken from a uniform distribution:

$$\Delta t_{RTQB} = U\left[RTT,\ \Delta t^{RTQB}_{max}\right]. \tag{10}$$

There are two reasons for considering a random value. First, the attacker's estimation of both the *answer instants* and *RTT* may not be perfect and, thus, attack packets may arrive even after *RTT* seconds from the *answer instant*. Second, it is recommendable to introduce a certain variability into the process, as previously discussed. Otherwise, the attacker might be capable of estimating the value Δt_{RTQB} and adapting the attack. In Section 5, we evaluate the influence of the parameter Δt^{RTQB}_{max}, as well as the efficiency really obtained by *RTQB*.

Similarly to the case of *RST*, the configuration of $\Delta\ t_{RTQB}$ following Expression (10) depends on the value of *RTT*. Obviously, this value may be different for every incoming request to the server. Although we have assumed here a constant value for *RTT*, in Section 5.3 we discuss the configuration of Δt_{RTQB} in a realistic scenario where different *RTT* values are present or *RTT* could take a large value.

Finally, regarding the impact of *RTQB* on the server behavior, it is clear that once a request enters the service queue, there is no difference in its process when *RTQB* is active. Thus, no impact is present due to the use of this technique. We explore this fact also in Section 5.

4.4. Improved Random Time Queue Blocking (IRTQB)

As previously described, *RTQB* takes advantage of the fact that the attacker uses short bursts of traffic that arrive around the *answer instants*, while any incoming request in a time interval around them is blocked. The main problem with *RTQB* is that it does not selectively choose requests during the blocking interval, but it discards all of them. An improved version of *RTQB*, called *Improved Random Time Queue Blocking*, *IRTQB*, is here introduced. Derived from *RTQB*, *IRTQB* is also based on discarding requests during a time interval around the *answer instants*. However, the main difference is that here an algorithm selectively chooses those requests that potentially comes from the attacker and, consequently, it is only these that are discarded. This algorithm is based on measuring the similarity between incoming requests.

As known, the attacker delivers the attack by sending several attack messages as short bursts[4]. As answers should reach the attacker in order to allow the attack to be readapted and a response attack packet sent, the attacker can only use restricted spoofing mechanisms. This means that spoofing is only allowed within the same network segment; otherwise, the attacker would not be able to sniff answers coming from the server.

In summary, it is expected that the attacker's requests will have a temporal similarity, *i.e.*, they arrive in a time interval around the *answer instants*, and also a spatial similarity, *i.e.*, the attack requests are forced to follow certain communication rules given by transport and network layer protocols.

We define a *spatial similarity metric*, *SSM*, between two requests to measure the probability that they come from the same source. Although many factors could be included in *SSM*, *e.g.*, time-to-live of incoming requests, sequence numbers, source IP addresses, etc., we are only interested in the analysis of the inclusion of an algorithm of this class in the defense technique. Thus, we choose a simple metric that considers only the source IP addresses of incoming requests.

The proposed metric is based on the common prefix length between two considered IP addresses. For two generic IP addresses A_i and A_j, we compute the similarity metric as the number of consecutive bits set to '1' (starting from the most significant bit) in the *bit XNOR* operation of the two addresses:

$$SSM(A_i, A_j) = \#\_consecutive\_bits_1(A_i XNOR A_j). \tag{11}$$

[4] We are assuming that an attack burst is sent from a single location, as it would be almost unfeasible for the attacker to efficiently synchronize the traffic from several locations affected by different network variability

For example, the similarity between the IPv4 addresses 192.168.20.3 and 192.168.24.3 would be 20. This spatial similarity metric is a measure of the probability that two IP addresses are in the same network. In order to decide whether a request comes from the attacker or not, there also exists a similarity threshold, *ST*, in such a way that if $SSM(A_i,A_j) > ST$, A_i and A_j are considered similar, and not otherwise.

The algorithm followed in *IRQTB* is shown in the Algorithm 1 diagram. We explain it now in detail.

IRQTB maintains a list of *answer instants* as long as they are happening in the server. As in *RTQB*, around every *answer instant*, *e.g.*, t_0, a time interval exists, denoted as the *attack interval.* This indicates when attack packets are likely to arrive and, thus, they could potentially be discarded. If the length of an attack burst is *B*, the attack interval is $[t_0 - B/2, t_0 + \Delta t_{max}^{IRTQB}]$. Note that requests coming before an *answer instant* do not enter the service queue, as no free positions exist under the assumption of overflow state in the server. However, these requests give useful information to decide whether requests coming after the *answer instant* are attack requests or not. For this reason, the attack interval begins at $t_0 - B/2$.

IRQTB also maintains a record of the timestamps and the source IP addresses for all incoming requests. Upon the arrival of a new request, the set of attack intervals that comprise the timestamp of the incoming request is obtained. Then, the similarity between the incoming request and every request that arrives within this set of attack intervals is computed. If the similarity is higher than the threshold *ST* for any two requests, then both are discarded. When discarded, no notification is sent to the client, in order to avoid a potential attacker getting information on this process.

Algorithm. Algorithm for the execution of *IRTQB*

```
For every answer
{
  Insert the answer instant in a list
  Compute attack interval for the answer
}

For every incoming request Ri
  {
    Record timestamp, source IP
    AI = attack intervals for Ri->AI
    For all requests Rj in AI
    {
      If SSM (Ri, Rj)>ST then
        Discard Ri and Rj
    }
}
```

In the configuration of *IRTQB* only two parameters appear, Δt_{max}^{IRTQB} and the similarity threshold, *ST*. Δt_{max}^{IRTQB} is configured following the rules previously explained for this parameter in the case of *RTQB* (Section 4.3). Regarding *ST*, this is a parameter that could be used to improve the accuracy of the algorithm. As previously discussed, the optimization of this algorithm is beyond the purpose of the paper, as we are only interested in the design of a defense technique that allows us to incorporate this kind of mechanisms. However, a brief consideration can be made here: the selection of a low value for *ST* is equivalent to considering that all the requests are spatially similar and, thus, to discarding all the requests within an attack interval. This would be the case of *RTQB*, in which all the requests are discarded.

5. Experimental evaluation of techniques

In this section, we experimentally evaluate the defense techniques previously described in this paper. For this purpose, an implementation in Network Simulator 2 [24] of both the attack and the server is used.

We evaluate the performance of the attack by measuring both the *in-system time*, t_I, and the *attack efficiency*, *E*. Recall that *E* is the percentage of service queue positions seized by the attacker over the total number of positions seized during the attack execution, while t_I represents the time from when a request enters the server to the instant at which its corresponding answer is sent.

5.1. Experimental framework description

A full implementation of the attack, as described in [10], is employed, *i.e.*, the use of multiple attack threads implies that they share information about the connections to the server and once a thread has non-attended connections it allows others to attend them by sending corresponding short bursts of attack traffic.

The server is able to make separate use of the different defense techniques explained here. As described in Section 4, the only configuration parameters for *RST*, *RAI* and *RTQB* are $\Delta t_{max}^{RST}, \Delta t_{max}^{RAI}$ and Δt_{max}^{RTQB}, respectively, whereas only for *IRTQB* does an additional configuration parameter appear, *i.e.*, the similarity threshold, *ST*.

Legitimate users generate traffic requests following a Poisson distribution (exponential inter-arrival times). The time interval between legitimate users' requests is denoted by λ.

A wide range of different server scenarios are tested. These are designed by modifying the values for the mean

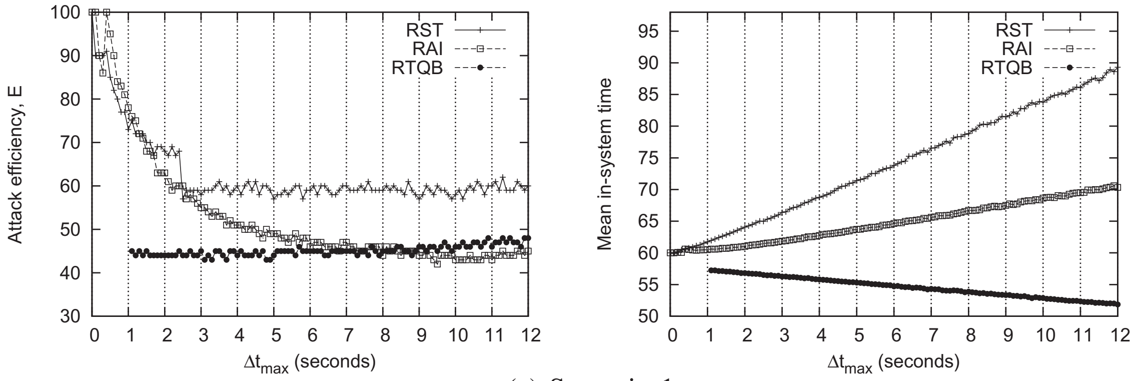
Table 1
Summary of configuration values for the attack and the server parameters in Scenario 1 (*S1*), Scenario 2 (*S2*) and Scenario 3 (*S3*).

Parameter	Value
Duration of attack burst, *B*	0.4 s
Time between attack packets in a burst	0.2 s
Mean service time, $\overline{T_s}$	12 s
Variance of server, $var[T_s]$	0 (*S1*), 0.2 (*S2, S3*)
Interval between legitimate users' requests, λ	3 s (*S1, S2*), 0.95 s (*S3*)
Number of server threads	1(*S1, S2*), 4 (*S3*)
Number of positions in service queue, *N*	4(*S1, S2*), 8 (*S3*)
Number of attack threads	= *N*
Attack duration	50,000 s
Round trip time, *RTT*	1 s
Similarity metric, *ST*	32

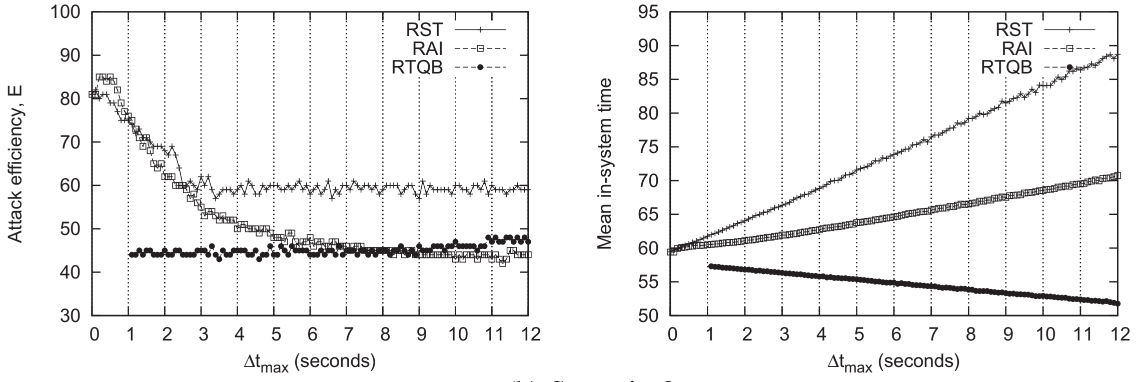
service time for attack and legitimate users' requests, $\overline{T_s}$, server variance, $var[T_s]$, round trip time, *RTT*, number of threads in the server, size of the service queue, *N*, and traffic rate of legitimate users, λ. For the purpose of showing our results here, we have defined three different scenarios:

- *Scenario 1, S1:* the server is monothreaded (a single execution thread is running in the server), and the variance of the service time for attack requests, $var[T_s]$, is 0. Here, a deterministic behavior of the server allows a clear observation of the effects of a given defense technique.
- *Scenario 2, S2:* here we explore the influence of introducing variance into the behavior of the server. Thus, we use the same configuration as in Scenario 1, but modifying the server variance, $var[T_s]$.
- *Scenario 3, S3:* the aim of this scenario is to check how a multithread operation in the server affects the performance of a given defense technique. We extend Scenario 2 by using a multithreaded version of the server.

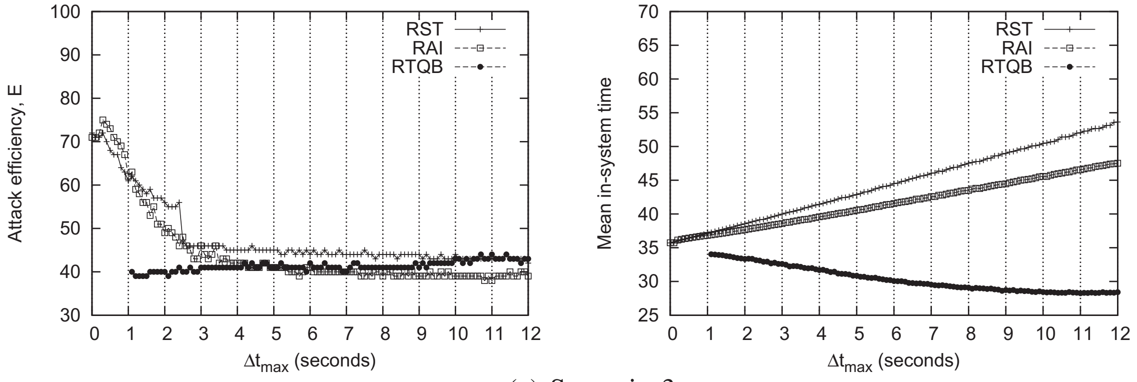
In the results described in the following, the configuration values used for both the attack and the server



(a) Scenario 1



(b) Scenario 2



(c) Scenario 3

Fig. 7. *Attack efficiency* and mean *in-system time* obtained during the execution of the attack with *RST*, *RAI* and *RTQB*.

parameters are shown in Table 1. Note that, for these results, the values for the parameters have been chosen according to (*i*) the attack design rules, *e.g.*, the number of attack threads depends on the size of the service queue, and (*ii*) the evaluation scenario considered (parameters $var[T_s]$ and *number of server threads*). Furthermore, for this set of configuration values, the traffic rate of legitimate users has been set up in such a way that the number of legitimate requests arriving at the server is similar to the number of attack requests. Finally, the round trip time for all the legitimate users and attackers is the same in these experiments. Although this is far from reality, we explore how different *RTT* values affect the efficiency of the defense techniques in Section 5.3.

5.2. Techniques evaluation

The results obtained for these scenarios, when the configuration parameters Δt^{RST}_{max}, Δt^{RAI}_{max} and Δt^{RTQB}_{max} are varied, are shown in Fig. 7 (in the following figures we generically denote all these parameters as Δt_{max}). We discuss now these results independently for every defense technique and later, in Section 7, the overall defense approach is discussed in a comparison of all the results achieved.

5.2.1. Results for RST

The results obtained for *RST* in Scenario 1 – Fig. 7a – show that *E* decreases from 100% to an asymptotic value in Δt^{RST}_{max} around 60%, which is reached for values $\Delta t^{RST}_{max} > 2.4$ *s*. This is the value at which the best performance of *RST* is achieved, given by the condition in Expression (5), *i.e.*, $\overline{\Delta t}_{RST} > \frac{B}{2} + RTT$. The value of 60% is given by the fact that, in this scenario, the number of legitimate requests is slightly lower than the number of attack requests (see Fig. 8).

In Scenario 2 – Fig. 7b, the evolution of *E* is similar. The only difference resides in the fact that for low values of Δt_{max}, *E* is lower than in Scenario 1, due to the existence of variance in the server, $var[T_s] = 0.2$. In Scenario 3 – Fig. 7c, – *E* also undergoes the same evolution. In this case, the asymptotic value is lower due to a higher value in the number of legitimate users' requests (see Fig. 8).

Regarding the results for the *in-system time*, Fig. 7 shows that the mean value increasese with Δt^{RST}_{max}. The increment follows the equation given by Expression (8). Note that, for Scenario 3, the existence of four threads in the server considerably reduces the value of the *in-system time*. However, even in this case there is an increment in t_l given by the application of *RST*.

5.2.2. Results for RAI

The results in Fig. 7 show that the efficiency of the attack decreases as the parameter Δt^{RAI}_{max} increases and the *RAI* defense is active. This is coherent with the behavior of *RAI*, given that the application of the defense enables an interval of $\Delta t_{RAI} - \frac{B}{2}$ seconds for legitimate users to introduce requests in the queue, before the expected arrival of a short attack burst.

Regarding the impact on the server behavior, it can be seen that the *in-system time* also rises, according to Expression (9), although the increment is lower than in the *RST* case.

5.2.3. Results for RTQB

The results for *RTQB* in Fig. 7 are shown for $\Delta t^{RTQB}_{max} > RTT$. Here, the minimum value is obtained for $\Delta t^{RTQB}_{max} = RTT$, whereas higher values in the parameter result in a slight increment of the attack efficiency. The reason for this is that this defense really enables an interval of time for legitimate users to seize positions in the queue delimited by the expiration of Δt_{RTQB} and the arrival of the next short attack burst. As Δt^{RTQB}_{max} becomes higher, this interval is shortened, thus decreasing the chances of legitimate users and, consequently, rising the attack efficiency.

The observed *in-system time* decreases when *RTQB* is applied. This is due to the fact that, as the queue is blocked during Δt_{RTQB}, the request in the service module is being processed during this time. Obviously, Δt_{RTQB} is not computed in the *in-system time* for the next request that arrives at the queue.

In summary, it can be deduced from these results that a practical *RTQB* implementation should be configured with $\Delta t^{RTQB}_{max} = RTT$. As previously indicated, note that *RTT* in these experiments takes the same value for all the attack-

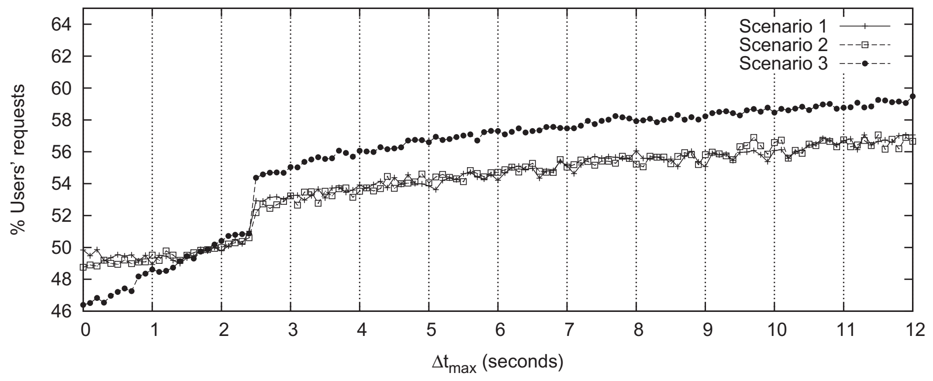


Fig. 8. Percentage of users' requests during the execution of the attack with *RST* defense for the three scenarios defined.

ers. In Section 5.3 we discuss how to deal with a realistic situation with heterogeneous *RTT* values.

5.2.4. Results for IRTQB

Let us recall that *IRTQB* is very similar to *RTQB*. However, it incorporates a mechanism for distinguishing legitimate users from attackers, based on two assumptions: (*i*) attack requests have a temporal similarity, *i.e.*, they arrive in short bursts, and (*ii*) attack requests also present a spatial similarity, *i.e.*, every short attack burst is sent from a limited number of different IP addresses (limitation in IP spoofing).

It is expected that the efficiency of *IRTQB* directly depends on the accuracy of the detection algorithm. As indicated in Section 4, we are not specifically interested in the evaluation of specific algorithms for the detection of the attacker, but only in assessing how to incorporate these algorithms in a defense strategy, and how this affects the server behavior. For this reason, in the evaluation of *IRTQB* we implement the simple algorithm described in Section 4.4, *i.e.*, a spatial similarity metric based only on the comparison of the source IP addresses of incoming requests.

We evaluate the attack efficiency with *IRTQB* for two different cases: (*i*) the number of different legitimate users

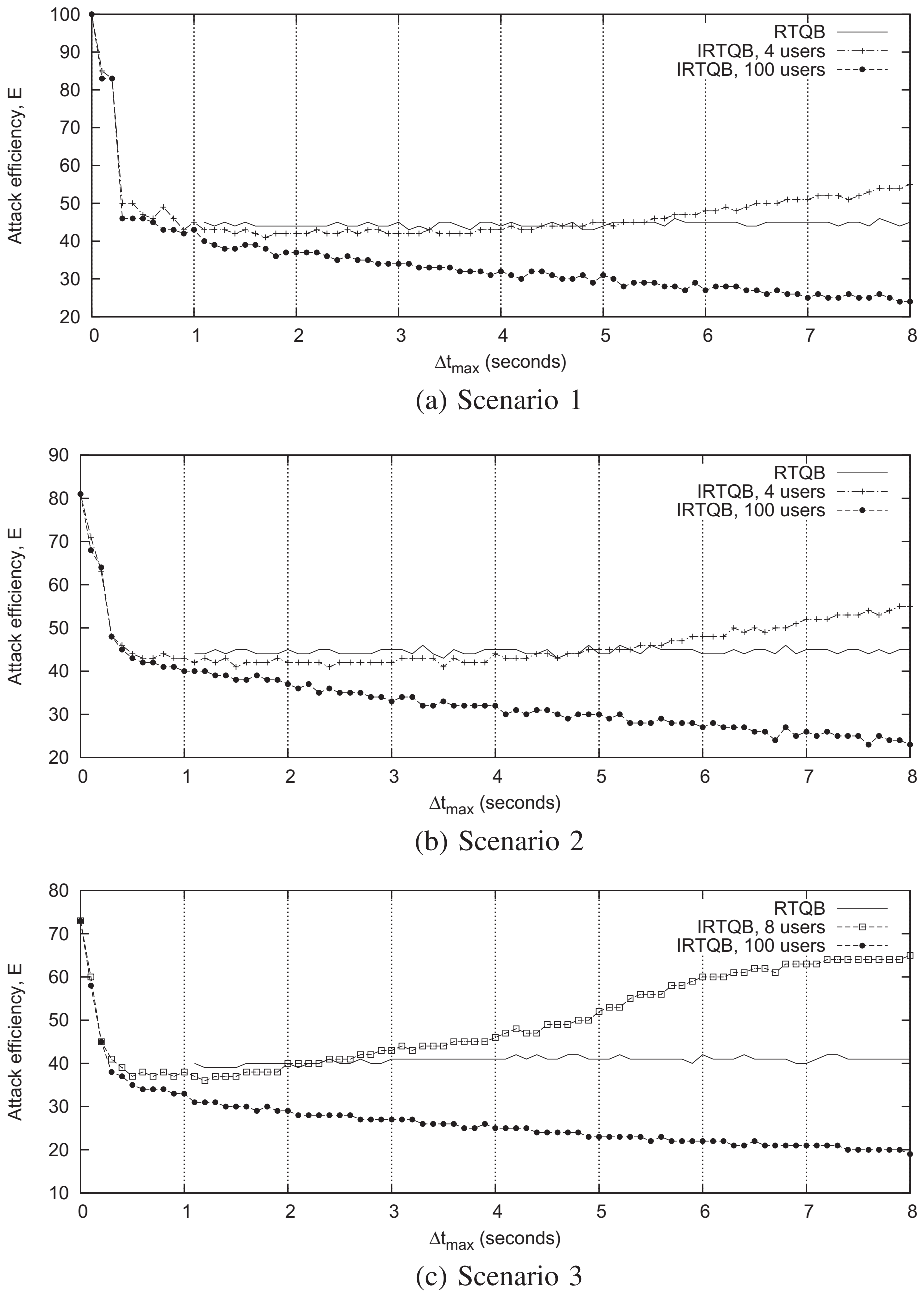


Fig. 9. Comparison of the *attack efficiency* for *IRTQB* and *RTQB* when different number of legitimate users are considered.

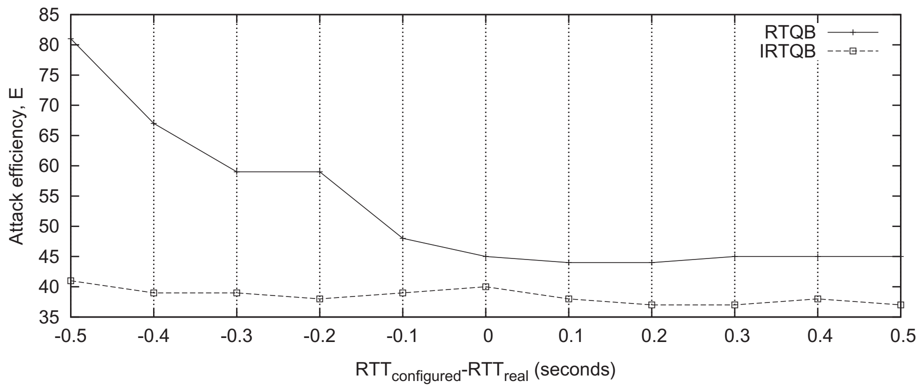


Fig. 10. Error resiliency analysis for the *RTT* estimation in both *RTQB* and *IRTQB*.

is similar to the number of IP addresses from which the attack is being executed, and (*ii*) the number of different legitimate users is higher than the number of attack locations. In the experiments, for the first case we choose 4 users – Scenarios 1, 2 – and 8 users – Scenario 3. In the second case, we choose 100 users for all scenarios. In both cases, the traffic rate for legitimate users is the same. Results for this experimentation are shown in Fig. 9.

As expected, *IRTQB* works better as the number of legitimate users is higher. Note that the algorithm discards spatially similar requests in a time interval $\left[-\frac{B}{2}, \Delta t_{max}^{IRTQB}\right]$. If the number of legitimate users is low for a given traffic rate, the probability that at least two legitimate users' requests are similar is higher. For this reason, a better performance is obtained for 100 users in Fig. 9.

When the Δt_{max}^{IRTQB} value is increased, the probability that two similar legitimate users' requests arrive at the server during the interval is higher. If the number of legitimate users is similar to the number of attack locations, the attack efficiency rises, as legitimate users are treated in a similar way to attackers. However, if the number of users is higher than the number of attackers, this is not true and, consequently, *IRTQB* is more effective (series "*IRTQB, 100 users*" in Fig. 9).

A configuration value for Δt_{max}^{IRTQB} which makes *IRTQB* independent of the percentage of legitimate users is a good choice. Here, we suggest the use of $\Delta t_{max} = RTT$. In this case, *IRTQB* would not be degraded by the effect of a low percentage of legitimate users, and the performance is also slightly better than for *RTQB*.

5.3. Parameter estimation

As suggested in Section 4, the server should properly configure either *RTQB* or *IRTQB* with the configuration value $\Delta t_{max} = RTT$. Let us suppose a value is chosen for Δt_{max} denoted as $RTT_{configured}$. We are interested in evaluating how a deviation between $RTT_{configured}$ and the real *RTT* value for a given request, RTT_{real}, affect the effectiveness of the attack. Fig. 10 shows the simulation results for the attack efficiency, considering Scenario 2, for different values of $RTT_{configured} - RTT_{real}$. These results show that *RTQB* is less resilient to negative deviations, while *IRTQB* remains stable for deviations in both directions. The following conclusion is drawn: it is better to configure Δt_{max} with a positive deviation.

Let RTT_i be the round trip time between the server and a generic client *i* which is sending requests to it. As a previous step in the configuration, the server should estimate, for every incoming request, the corresponding round trip time RTT_i, *e.g.*, for TCP connections this could be done by examining the timestamps for packets in the three way handshake procedure. Then, the server maintains a list containing the different RTT_i values. As Δt_{max} should take a value that will affect all the requests, the results from Fig. 10 lead us to choose a configuration value for Δt_{max} as

$$\Delta t_{max} = max[RTT_i]. \tag{12}$$

Recall that *RTQB* and *IRTQB* are mechanisms based on controlling the DoS periods in order to situate them around the *answer instants*, thus penalizing the attacker. It could happen that the defense technique itself causes even more DoS than does the attacker. This might happen if Δt_{max} takes a large value, *e.g.*, due to the existence of large RTT_i values according to Expression (12). In this case, the incoming traffic rate to the server becomes lower than the output rate, thus reducing the occupation of the service queue. Then, by monitoring this occupation, it could be deduced that this situation has been reached. At this moment, the considered maximum RTT_i value is discarded from the list of existent RTT_i and Δt_{max} is configured again as in Expression (12).

6. Real environment validation

In a previous work [11,10] in which the LoRDAS attack is well described and evaluated, we contributed several thorough experiments made with real environment implementations of LoRDAS attack. These were intended to illustrate two aspects: (*i*) A real implementation of the LoRDAS attack is feasible, and (*ii*) the results obtained from the used NS-2 simulation environment are coherent with those obtained in real environment implementations.

Here, we are not as interested in demonstrating the feasibility of a real implementation of the attack, as in evaluating if the defense techniques proposed in this paper are implementable and if they work as expected.

As shown in the results given in Section 5, both *RTQB* and *IRTQB* are the techniques which yield the best results. We argue that a valid *RTQB* implementation is enough to demonstrate also the feasibility of the implementation of *IRTQB*, as the latter only implies the additional inclusion of an algorithm to discern the requests to be discarded during the blocking time. Thus, we have chosen to make a real *RTQB* implementation, which is described now.

RTQB has been implemented on a Linux kernel 2.6.18, where the TCP/IP stack has been properly modified. We have focused on attacks to applications that use TCP sockets. In the Linux 2.6 kernel, the connections established by peers to the server are queued up in a *backlog queue*, which is associated with a listening socket defined by the application. This queue is located in kernel space, and the application is able to extract these connections from it by means of the `accept` system call [25]. Thus, our implementation consists of a kernel module that is able to block the entrance to the listening socket backlog queue during the time intervals around the answer instants (shutdown of TCP connections) specified by *RTQB*. The code within this module is activated from the kernel through the insertion of appropriately located kernel hooks [26].

We have also implemented a simple monoprocess server, which sequentially serves requests in a configurable fixed service time. This server emulates the behavior of a persistent HTTP timeout in an Apache server, at the same time that its simplicity allows us to easily obtain measures. Legitimate clients are also implemented as processes that sends requests following a Poisson distribution. Finally, we use our implementation of the attack presented in [10].

We have evaluated both the *attack efficiency*, *E*, and the *in-system time*, t_I, obtained in this test bed for different values of the *RTQB* configuration parameter Δt_{max}. The values selected for the rest of the attack and the server configuration parameters are those already chosen for the evaluation in the Scenario 2 of the simulated environment (Table 1). Furthermore, the estimated mean *RTT* approximates 1 ms in our test bed.

The results obtained are shown in Fig. 11. Here, the attack efficiency and the in-system time are represented when *RTQB* is not activated ($\Delta t_{max} = 0$) and for a set of different Δt_{max} values. We first note that the activation of RTQB make the attack efficiency decrease more than 37%. We can also see that the evolution of the attack efficiency is similar to that obtained from simulations (Fig. 7b). The oscillations obtained in the efficiency (around 41%) are really small, and they are due to changes in the network and server variability from one experiment to other. Also the values for the in-system time correspond with those obtained with the simulation environment (Fig. 7b).

These real environment experiments have allowed us to validate the results previously obtained from the simulated environment, as both present the same behavior.

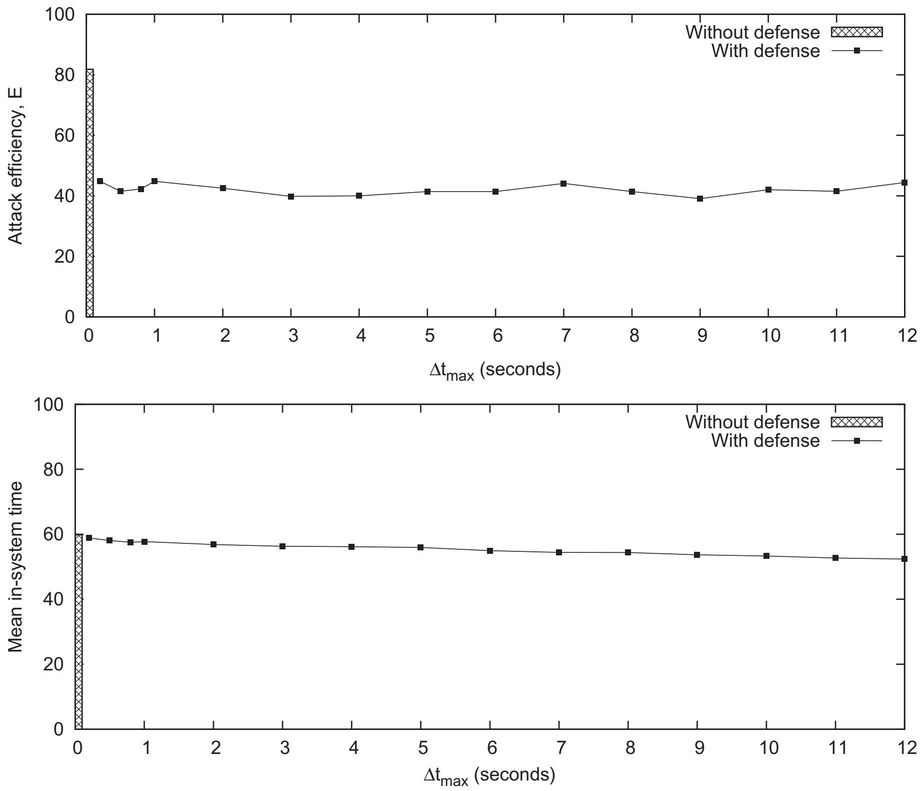


Fig. 11. Evolution of the *attack efficiency* and the mean *in-system time* with Δt_{max}, obtained during the execution of the attack to a real server with/without RTQB defense.

Moreover, we now dare to say that no important difficulties are expected in the implementation of the suggested defenses.

7. Discussion of the results

After the description and experimental evaluation of different defense techniques, we now aim at discussing these results and considering relevant aspects when adopting a final solution for the defense against LoRDAS attacks.

The results in Fig. 7 demonstrate that *RST*,although a simple strategy, is not the best solution in every case, either comparing the *attack efficiency* or the *in-system time* (impact on the server behavior). When low values for Δt_{max} are considered, *RTQB* and *IRTQB* are the techniques that produce the best results. However, for high Δt_{max} values, *RAI* could even work better than the former ones. The main problem with *RAI* is the impact generated in the server behavior, which increases linearly with Δt_{max}. For this reason, we could initially discard a solution based on *RAI*.

A decision between the remaining solutions, *i.e.*, *RTQB* and *IRTQB,* is not straightforward and may depend on the system to be secured against LoRDAS attacks. The following aspects should be considered:

- *Ease of implementation:* the implementation should be as easy as possible, in order not to overload the network stack with additional tasks that slow down the process. *RTQB* implementation is simple, as it only needs to keep track of *RTT* values for incoming requests. On the contrary, the implementation of *IRTQB* involves not only additional memory resources for saving the timestamps and the source of the incoming requests, but also additional processing for every incoming request in order to calculate the similarity with any other previous petition. Thus, considering this aspect, *RTQB* is a better solution.
- *Effectiveness of the technique:* as shown in Fig. 9, for the suggested configuration values in both *RTQB* and *IRTQB*, *i.e.*, $\Delta t_{max} = RTT$, *IRTQB* works slightly better than *RTQB*, especially when the percentage of legitimate users is high. With this criterion, *IRTQB* is a better solution for the defense.
- *Error resiliency:* deviations in the estimation of parameters like *RTT* may appear. As shown in Fig. 10, *IRTQB* is more resilient than *RTQB* to deviations due to the fact that the attack efficiency obtained maintains a stable value when deviations occur. Again, *IRTQB* is preferable under this criterion.

In summary, we recommend *IRTQB* provided that enough processing resources are available in the server. However, note that *IRTQB* depends on the detection algorithm used. A good choice for this algorithm may enhance the advantages of *IRTQB* at the expense of resource consumption in the server.

A final consideration should be made. Note that, during the description and evaluation of the different techniques, an overflow state in the server has been assumed, *i.e.*, the service queue is considered to be full of requests. A final question ought to be answered here: when should the defense technique be activated? In this respect, we suggest activating the defense strategies not only in the overflow state, but probabilistically for every answer in the server, with a probability that depends on the service queue occupation. We plan to explore different strategies in this line as a future work.

8. Conclusions and future work

In this paper we have explored different mechanisms for defending an application server against LoRDAS attacks. The approaches considered are based on modifying the way in which the server operates with the queues in which incoming requests are stored. Instead of presenting a final solution, we have described different alternatives, discussing the pros and cons of each.

Both the attack and the server with the suggested defenses techniques have been implemented in a simulated environment. An experimental framework designed to evaluate the effects of the defenses has been contributed and conclusions have been extracted from these experiments. Furthermore, the feasibility of an implementation of the suggested defense strategies has been evaluated.

From the alternatives evaluated, the best techniques are based on blocking the entry of requests in the service queue of the server once an answer is sent (*RTQB*) or discarding those requests which are selected by a suggested detection algorithm (*IRTQB*). These are able to reduce the efficiency of the attack by up to half, while no impact on the amount of time spent by the requests in the system is generated. The choice of the most suitable technique depends on the processing resources of the server to be protected, as the alternative *IRTQB* is more expensive in this sense but also more efficient.

As future work, we plan to follow three lines of research. First, we plan to work on the development of detection algorithms to differentiate attack requests from legitimate users' requests. The main ideas underlying this kind of algorithms have been discussed here, but only a simple algorithm has been tested. Far from considering only the source IP addresses, we plan to incorporate a more complex set of features to define the spatial similarity between requests. Second, in this paper we have explored the application of the suggested techniques when the server is in an overflow state. We will now work on testing how to progressively apply these defense techniques as the server becomes congested, and not only upon overflow conditions. Third, we plan to evaluate the performance of these techniques in a production environment in order to assess the amount of resources needed for their execution.

Acknowledgments

This work was partially supported by the Spanish National Research Programme of the MEC, under project TEC2008-06663-C03-02 (70% FEDER funds).

We also specially thank Pedro García Teodoro for his helpful comments.

References

[1] J. Mirkovic, P. Reiher, A taxonomy of DDoS attack and DDoS defense mechanisms, SIGCOMM Comput. Commun. Rev. 34 (2) (2004) 39–53.

[2] J. Mirkovic, S. Dietrich, D. Dittrich, P. Reiher, Internet Denial of Service. Attack and Defense Mechanisms, Prentice Hall, 2004, ISBN: 0-13-147573-8.

[3] A. Shevtekar, J. Stille, N. Ansari, On the impacts of low rate DoS attacks on VoIP traffic, Security and Communication Networks 1 (1) (2008) 45–56. doi: http://dx.doi.org/10.1002/sec.7.

[4] A. Kuzmanovic, E. Knightly, Low-rate TCP-targeted denial of service attacks (the shrew vs. the mice and elephants), in: Proceedings of the ACM SIGCOMM'03, 2003, pp. 75–86.

[5] M. Guirguis, A. Bestavros, I. Matta, Exploiting the transients of adaptation for RoQ attacks on internet resources, in: ICNP '04: Proceedings of the 12th IEEE International Conference on Network Protocols, IEEE Computer Society, Washington, DC, USA, 2004, pp. 184–195.

[6] M. Guirguis, A. Bestavros, I. Matta, Y. Zhang, Reduction of quality (RoQ) attacks on internet end-systems, in: INFOCOM 2005, 24th IEEE International Conference on Computer Communications, 2005, pp. 1362–1372.

[7] M. Guirguis, A. Bestavros, I. Matta, Y. Zhang, Reduction of quality (RoQ) attacks on dynamic load balancers: vulnerability assessment and design tradeoffs, in: INFOCOM 2007, 26th IEEE International Conference on Computer Communications, 2007, pp. 857–865.

[8] W. Ren, D. Yeung, H. Jin, M. Yang, Pulsing RoQ DDoS attack and defense scheme in mobile ad hoc networks, Int. J. Network Security 4 (2) (2007) 227–234.

[9] M. Guirguis, A. Bestavros, I. Matta, Y. Zhang, Adversarial exploits of end-systems adaptation dynamics, J. Parallel Distrib. Comput. 67 (3) (2007) 318–335. doi: http://dx.doi.org/10.1016/j.jpdc.2006.10.005.

[10] G. Maciá-Fernández, J.E. Díaz-Verdejo, P. García-Teodoro, Evaluation of a low-rate DoS attack against application servers, Comput. Security 27 (7) (2009) 335–354.

[11] G. Maciá-Fernández, J.E. Díaz-Verdejo, P. García-Teodoro, Evaluation of a low-rate dos attack against iterative servers, Comput. Networks 51 (4) (2007) 1013–1030. doi: http://dx.doi.org/10.1016/j.comnet.2006.07.002.

[12] H. Sun, J. Lui, D. Yau, Defending against low-rate TCP attacks: dynamic detection and protection, in: Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP04), 2004, pp. 196–205.

[13] E. Keogh, Exact indexing of dynamic time warping, in: Proceedings of the 28th VLDB Conference, China, 2002.

[14] H. Sun, J.C.S. Lui, D.K.Y. Yau, Distributed mechanism in detecting and defending against the low-rate TCP attack, Comput. Netw. 50 (13) (2006) 2312–2330. doi: http://dx.doi.org/10.1016/j.comnet.2005.09.016.

[15] W. Wei, Y. Dong, D. Lu, G. Jin, H. Lao, A novel mechanism to defend against low-rate denial-of-service attacks, Lecture Notes Comput. Sci. 3975 (2006) 261–271.

[16] A. Shevtekar, K. Anantharam, N. Ansari, Low rate TCP denial-of-service attack detection at edge routers, IEEE Commun. Lett. 9 (2005) 363–365.

[17] G. Yang, M. Gerla, M.Y. Sanadidi, Defense against low-rate TCP-targeted denial-of-service attacks, in: Proceedings of the IEEE Symposium on Computers and Communications (ISCC'04), Alexandria, Egypt, 2004, pp. 345–350.

[18] S. Sarat A., B.C. Pierce, D.N. Turner, On the effect of router buffer sizes on low-rate denial of service attacks, in: International Conference on Computer Communications and Networks, 2005 (ICCCN'05), 2005, pp. 281–286.

[19] Y. Chen, K. Hwang, Spectral analysis of TCP flows for defense against reduction-of-quality attacks, in: Proceedings of the IEEE International Conference on Communications (ICC'07), 2007, pp. 1203–1210.

[20] A. Shevtekar, N. Ansari, A router-based technique to mitigate reduction of quality (RoQ) attacks, Comput. Networks 52 (5) (2008) 957–970. doi: http://dx.doi.org/10.1016/j.comnet.2007.11.015.

[21] K. Argyraki, D.R. Cheriton, Scalable network-layer defense against internet bandwidth-flooding attacks, IEEE/ACM Trans. Networks 17 (4) (2009) 1284–1297. doi: http://dx.doi.org/10.1109/TNET.2008.2007431.

[22] M. Srivatsa, A. Iyengar, J. Yin, L. Liu, Mitigating application-level denial of service attacks on web servers: a client-transparent approach, ACM Trans. Web 2 (3) (2008) 1–49. doi: http://doi.acm.org/10.1145/1377488.1377489.

[23] S. Ranjan, R. Swaminathan, M. Uysal, A. Nucci, E. Knightly, DDoS-shield: DDoS-resilient scheduling to counter application layer attacks, IEEE/ACM Trans. Networks 17 (1) (2009) 26–39. doi: http://dx.doi.org/10.1109/TNET.2008.926503.

[24] K. Fall, K. Varadhan, The NS manual, <http://www.isi.edu/nsnam/ns/>, 2009. URL <http://www.isi.edu/nsnam/ns/>.

[25] W.R. Stevens, B. Fenner, A.M. Rudoff, Unix Network Programming, the Sockets Networking API, vol. 1, third ed., Addison-Wesley Professional, 2003, ISBN: 0-13-141155-1.

[26] C. Wright, C. Cowan, S. Smalley, J. Morris, G. Kroah-Hartman, Linux security modules: general security support for the linux kernel, in: Proceedings of the 11th USENIX Security Symposium, August, 2002, pp. 17–31.

Gabriel Maciá-Fernández is Assistant Professor in the Department of Signal Theory, Telematics and Communications of the University of Granada (Spain). He received a MS in Telecommunications Engineering from the University of Seville, Spain, and the Ph.D. in Telecommunications Engineering from the university of Granada. In the period 1999–2005 he worked as a specialist consultant at 'Vodafone España'. His research interests are focused on computer and network security, with special focus on intrusion detection, reliable protocol design, network information leakage and denial of service.

Rafael A. Rodríguez-Gómez is a Ph.D. student in the Department of Signal Theory, Telematics and Communications of the University of Granada. He received his MSc degree in Telecommunications from the University of Granada (Spain) in 2008. His research interests include defense against DoS attacks, security in P2P networks and stochastic modeling applied in the security field.

Jesús E. Díaz-Verdejo is Associate Professor in the Department of Signal Theory, Telematics and Communications of the University of Granada (Spain). He received his B.Sc. in Physics (Electronics speciality) from the University of Granada in 1989 and held a Ph.D. Grant from Spanish Government. Since 1990 he is a lecturer at this University. In 1995 he obtained a Ph.D. degree in Physics. His initial research interest was related with speech technologies, especially automatic speech recognition. He is currently working in computer networks, mainly in computer and network security, although he has developed some work in telematics applications and e-learning systems.