An Adaptive Selective Defense for Application Layer DDoS Attacks

Yuri Gil Dantas[1], **Vivek Nigam**[2], **Iguatemi E. Fonseca**[2],

[1] Technische Universität Darmstadt – Darmstadt, Hessen – Germany

[2]Federal University of Paraíba (UFPB) – João Pessoa, PB – Brazil

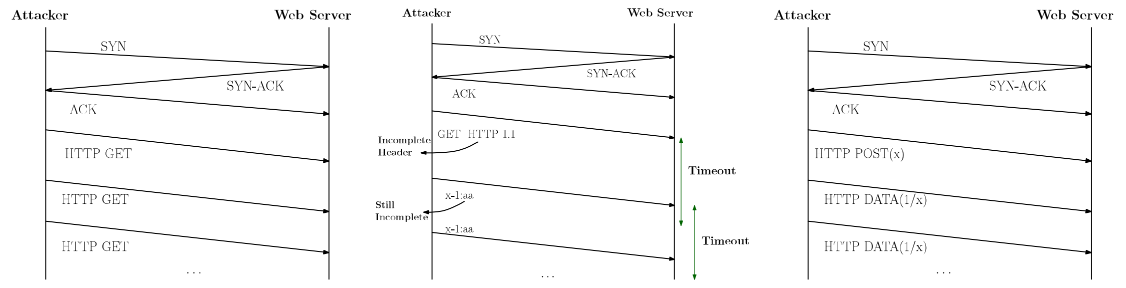
Abstract. *Recent Distributed Denial of Service (DDoS) attacks have been carried out over the application layer where an attacker can target a particular application of the server while leaving the others applications still available, thus generating less traffic and being harder to detected. This work proposes a novel defense called SeVen which uses Selective Strategies to mitigate such attacks. We validate SeVen by: (1) Simulation: The entire defense mechanism was formalized in Maude and simulated using the statistical model checker (PVeStA). 2) Experiments: Analysis of efficiency SeVen, implemented in C++, in a real experiment on the network showing that SeVen is effective in mitigating the HTTP POST and Slowloris attacks yielding high levels of availability.*

1. INTRODUCTION

Denial of Service Attacks (DDoS) have always been a major concern to network administrators. Traditionally, DDoS attacks are carried out on the transport layer by sending a number of packets to a server much greater than the server's processing capabilities making it unavailable to legitimate users. Although still dangerous, such attacks can be identified by using traffic analysis tools and mitigated by existing defenses [Zargar et al. 2013], such as the Adaptive Selective Defense [Khanna et al. 2008, Khanna et al. 2012].

In the recent years, a new generation of sophisticated attacks has emerged that exploit application layer protocols, such as HTTP and SIP protocols. Application Layer DDoS attacks (ADDoS) can target a single application in the server, *e.g.*, a web-server, instead of the whole server machine. The attack does not need to generate a huge amount of traffic, thus rendering existing defenses ineffective [Zargar et al. 2013]. Moreover, since the traffic generated by the attack is similar to legitimate traffic, it is hard to distinguish when a request is part of an attack or it is a legitimate request. These attacks are becoming increasingly preferred by attackers. In 2015, DDoS exploiting the HTTP protocol represented more than 30% of the total DDoS attacks just behind the traditional SYN-ACK DDoS with above 50% of all DDoS attacks (`https://tinyurl.com/q9p7qfz`).

Contributions: We formalize three different ADDoS attacks in the the computational tool Maude [Clavel et al. 2007], namely HTTP GET flooding, HTTP POST and Slowloris attacks (described in Section 2). We propose (Section 3) a novel defense against ADDoS attacks, called SeVen, based on ASV [Khanna et al. 2008]. As ASV was designed for mitigating transport layer DDoS attacks, it assumes that communication is a simple client-server stateless sync-ack interaction. This is, however, not enough for mitigating ADDoS attacks, as the protocols used by these attacks, such as HTTP, have a notion of state. SeVen thus extends ASV by incorporating into the defense a notion of state needed for mitigating ADDoS attacks, such as the HTTP POST and Slowloris attacks; We formalize SeVen in Maude [Clavel et al. 2007] and validate (Section 4) our defense by simulation using the



(a) Sequence of messages in the HTTP GET flooding Attack. (b) Sequence of messages in the Slowloris Attack. (c) Sequence of messages in the HTTP POST attack.

Figure 1. Behavior of three Application Layer DDoS Attacks.

statistical model checker tool PVeStA [AlTurki and Meseguer 2011]. We implement a tool for SeVen in C++. We carry out experiments on the network by realizing attacks on a Apache web-server using two different scenarios, i.e., with and without running SeVen. In each experiment, we measure the Success Rate of clients and their Time of Response (TTS). For all attacks, we observe great improvements on the Success Rate and TTS. Finally we briefly discuss the outcome of this work (Section 5).

2. ATTACK DESCRIPTIONS

HTTP GET Flooding Attack Flooding attacks [Zargar et al. 2013, Shankesi et al. 2009] are very similar to transport layer DDoS attacks in that the attacker and his zombies send a great number of packages. However, instead of consuming the resources of a whole server as in network layer DDoS, the target is an application running on the server, such as a web-server. When the attack is carried out, the application is overwhelmed and is no longer available to honest clients. The attack pattern is as shown in Figure 1(a), where the attacker establishes a connection by completing the SYN-ACK handshake with the target web-server. At this time, the attacker sends a great number of packages in order to overload such a web-server.

Slowloris Attack The Slowloris attack [slowloris 2013] exploits the HTTP GET method, used to retrieve web content. The attack follows by using the sequence of messages shown in Figure 1(b). The attacker completes the SYN-ACK handshake with the target web-server thus establishing a connection with the web-server machine exactly as a legitimate client would do. Then the attacker sends an incomplete header with a GET request. At this moment the web-server allocates one of its workers to handle this incoming request. However, since the request header is not yet complete, the web-server waits, as specified by the HTTP protocol version 1.1 [RFC 1999], until a new piece of the request header is received. If such a piece is not received until a given timeout elapses, the web-server rejects this request. The attacker can easily infer, beforehand, this timeout by measuring the time until a request with an incomplete header is rejected. Thus, once the timeout is almost reached, the attacker sends another piece of the header, which can be anything actually as long as it does not complete the header of the request. The tool implemented by Anonymous sends "x-1:aa". This causes the web-server to reset the timeout count and wait once again for another piece of the header.

HTTP POST Attack The HTTP POST attack [r-u-dead yet 2013] exploits the POST method from the HTTP protocol. It works on web-pages that have some type of form to be filled. Figure 1(c) depicts the sequence of messages used in the HTTP POST attack.

The attacker establishes a connection by completing the SYN-ACK handshake with the web-server's machine. At this point, the attacker sends a (complete) HTTP POST request informing that it will send a large number, x, of bytes to the web-server corresponding to the contents of some form. The web-server, then, allocates one of its workers to serve this request. However, instead of sending big chunks of this data to the web-server as a legitimate client would do, the attacker sends very small pieces, typically a single byte per subsequent message. Moreover, it sends each piece in large intervals (typically 10s per byte). In the meantime the allocated worker waits until it receives all x bytes and cannot serve any other (legitimate) requests. By sending a number of POST requests greater than the number of workers in the web-server, all workers are busy waiting for attacker requests to be completed and therefore the web-server is not able to serve legitimate clients.

Slowloris attack and HTTP POST attacks do not generate high traffic load, nor high CPU and memory consumption. Thus, it is easy for administrators (or automated tool) monitoring these parameters to believe that their application is not under attack.

3. SEVEN

Our novel defense Selective Verification in Application Layer (SeVen as a reference to the number of the Application Layer in the OSI model) is based on ASV [Khanna et al. 2008]. Instead of analysing the traffic pattern of a given attack, SeVen assumes that whenever under attack the number of requests from the attacker is greater than that of honest users. An application using SeVen does not immediately process incoming messages, but waits for a period of time, t_S, called a *round*. During a round, SeVen accumulates messages received in a buffer. Depending on the size of this buffer, it may reject some requests.

More precisely, SeVen is composed of a natural number (PMod) and of two buffers, $\mathcal{P}, \mathcal{R}$, represented by lists:

$$\langle \mathcal{P}, \mathcal{R}, \texttt{PMod} \rangle.$$

The buffer $\mathcal{P}$ contains the requests *partially processed*, and $\mathcal{R}$ the requests *are expected to be received and be processed* by the application. The natural number PMod is a counter, also used in ASV, that is used to modify the probability distributions of when a request should be kept as it will become clear below. At the beginning of a round $\texttt{PMod} = 0$. The buffers $\mathcal{P}$ and $\mathcal{R}$ contain requests of the form $\langle \texttt{ip} : \texttt{socket}, \texttt{data} \rangle$, where $\texttt{ip} : \texttt{socket}$ is the requesting agent's identifier represented by its IP and the socket it is using and data is the data in the request, for example, the contents of a form. The following invariants hold:

- We assume that two different requests in $\mathcal{P}$ (respectively, in $\mathcal{R}$) have necessarily different identifiers;
- $\langle \texttt{ip} : \texttt{socket}, \texttt{data}_2 \rangle \in \mathcal{R}$ iff $\langle \texttt{ip} : \texttt{socket}, \texttt{data}_1 \rangle \in \mathcal{P}$, where $\texttt{data}_1$ and $\texttt{data}_2$ are pieces of data;
- At the beginning of a round, for all $\langle \texttt{ip} : \texttt{socket}, \texttt{data} \rangle \in \mathcal{R}$, $\texttt{data} = 0$, where 0 denotes the empty data.

The first invariant specifies that requests in a buffer corresponds to different agents, while the second invariant specifies that all agents that are sending data during a *round* are being processed by $\mathcal{P}$. The third invariant specifies that at the beginning of a round, the application has not received any further data to be processed. It also assumes an upper-bound, k, on the number of elements in $\mathcal{P}$ and in $\mathcal{R}$. Intuitively, this bound specifies the number of requests an application can process at any given time. For instance in a web-sever, this bound is the number of sockets that it has to process requests.

During a round, SeVen does not reject any incoming requests until k requests has arrived. However, under the assumption that the application can only handle k requests at a time, if the buffer $\mathcal{R}$ has already k requests and yet another request, r_I, arrives, then SeVen needs to make a choice of which requests to keep in its buffers and which to drop:

- It may decide to not keep the incoming request, r_I, which means that the requests currently in the buffers are not affected;
- or it may decide to keep the incoming request, which means that one request in the buffers should be replaced by the new request.

These choices are governed by probability distributions. At the end of a round, SeVen processes the request that remain in $\mathcal{R}$.

3.1. Defense by Example

Instead of describing the algorithm in its complete detail, for which we refer to [Dantas et al. 2014, Dantas 2015], we illustrate its behavior by means of an example. Assume that the upper-bound $k = 4$. Assume that initially there are three requests to be processed, where for simplicity we use natural numbers for data parameter of requests:

$$\begin{aligned}\mathcal{P}_0 &= [\langle \mathtt{ip}_1 : \mathtt{socket}_1, 10\rangle, \langle \mathtt{ip}_2 : \mathtt{socket}_2, 5\rangle, \langle \mathtt{ip}_3 : \mathtt{socket}_3, 2\rangle] \\ \mathcal{R}_0 &= [\langle \mathtt{ip}_1 : \mathtt{socket}_1, 0\rangle, \langle \mathtt{ip}_2 : \mathtt{socket}_2, 0\rangle, \langle \mathtt{ip}_3 : \mathtt{socket}_3, 0\rangle]\end{aligned}$$

This state specifies that the application has received partial data from three requets, $\mathtt{ip}_1 : \mathtt{socket}_1, \mathtt{ip}_2 : \mathtt{socket}_2, \mathtt{ip}_3 : \mathtt{socket}_3$, and it is waiting for more data to be received. It waits for the time, t_S, of a round to collect new incoming requests. Say that it receives the request: $\langle \mathtt{ip}_1 : \mathtt{socket}_1, 5\rangle$ from $\mathtt{ip}_1 : \mathtt{socket}_1$ with 5 pieces of data. The application updates its $\mathcal{R}$ buffer to the following:

$$\begin{aligned}\mathcal{P}_0 &= [\langle \mathtt{ip}_1 : \mathtt{socket}_1, 10\rangle, \langle \mathtt{ip}_2 : \mathtt{socket}_2, 5\rangle, \langle \mathtt{ip}_3 : \mathtt{socket}_3, 2\rangle] \\ \mathcal{R}_1 &= [\langle \mathtt{ip}_1 : \mathtt{socket}_1, 5\rangle, \langle \mathtt{ip}_2 : \mathtt{socket}_2, 0\rangle, \langle \mathtt{ip}_3 : \mathtt{socket}_3, 0\rangle]\end{aligned}$$

Notice that it does not immediately process this new request. It waits for the round to finish. Say now that a new request arrives: $\langle \mathtt{ip}_4 : \mathtt{socket}_4, 7\rangle$ with a fresh identifer $\mathtt{ip}_4 : \mathtt{socket}_4$. Since the number of elements in $\mathcal{P}$ is less than $k = 4$, it can safely add this request to the buffers as follows, where @ is the concatenation operator:

$$\mathcal{P}_1 = \mathcal{P}_0 @ [\langle \mathtt{ip}_4 : \mathtt{socket}_4, 0\rangle] \qquad \mathcal{R}_2 = \mathcal{R}_1 @ [\langle \mathtt{ip}_4 : \mathtt{socket}_4, 7\rangle]$$

That is, it has received 7 pieces of data from $\mathtt{ip}_4 : \mathtt{socket}_4$ but not processed any piece of information.

Assume now that another request $req_\nu = \langle \mathtt{ip}_5 : \mathtt{socket}_5, 1\rangle$ has arrived. Since the number of elements in $\mathcal{P}$ is equal to $k = 4$, then it must decide whether to process req_ν or not. It first sets $\mathtt{PMod} := \mathtt{PMod} + 1$ and decides to keep req_ν with probability

$$\mathtt{Prob} = \frac{k}{k + \mathtt{PMod}}$$

Notice that the probability of accepting new incomming requests reduces progressively once the buffer contains k elements. For more about this choice we refer to [Khanna et al. 2008, Khanna et al. 2012, Dantas et al. 2014]. Say that after throwing the coin, it accepts to process the request req_ν. It should now decide which one of the requests in $\mathcal{P}$ and $\mathcal{R}$ it should drop, because it cannot process more than k requests by assumption. In this thesis, such a decision is done by an uniform probability, however SeVen can be instantiate by different probability distributions, as you can see in [Dantas 2015, Dantas et al. 2016, Lemos et al. 2016]. Say that it chooses to drop the request with identification $\texttt{ip}_2 : \texttt{socket}_2$, sending an appropriate message to the requesting agent. The resulting buffers are as follows where $\setminus$ is the operator that removes an element from a list:

$$\mathcal{P}_2 = (\mathcal{P}_1 \setminus \{\langle \texttt{ip}_2 : \texttt{socket}_2, 5\rangle\})@[\langle \texttt{ip}_5 : \texttt{socket}_5, 0\rangle]$$
$$\mathcal{R}_3 = (\mathcal{R}_2 \setminus \{\langle \texttt{ip}_2 : \texttt{socket}_2, 0\rangle\})@[\langle \texttt{ip}_5 : \texttt{socket}_5, 1\rangle]$$

Consider now the end of this round. The application processes the requests that survived in $\mathcal{R}$, *e.g.*, the 5 pieces of data received for request identified $\texttt{ip}_1 : \texttt{socket}_1$ are processed, that is, they are added to the initial 10 pieces. The resulting buffer $\mathcal{P}$ is:

$$\left[\begin{array}{c} \mathcal{P}_3 = \langle \texttt{ip}_1 : \texttt{socket}_1, 15\rangle, \langle \texttt{ip}_3 : \texttt{socket}_3, 2\rangle, \\ \langle \texttt{ip}_4 : \texttt{socket}_4, 7\rangle, \langle \texttt{ip}_5 : \texttt{socket}_5, 1\rangle \end{array}\right]$$

The data component of the requests in $\mathcal{R}$ are all set to 0, as well as PMod.

Finally, the application checks which requests have been completed. Say that this is the case for the request identified $\texttt{ip}_1 : \texttt{socket}_1$, then it sends the appropriate web application' data to agent ip_1.

Intuition for the Effectiveness of Selective Strategies The strategy that we just described above is not tailored to mitigate any particular attack. We are simply selecting requests by using traffic rate which incoming requests to select and by random with uniform probability which requests to drop. It might seem puzzling that such a simple strategy works, but we have two rational assumptions for that: 1) the goal of a Low-Rate ADDoS attack is to occupy for long periods of time the workers of the web-server. 2) once the application is at its maximum capacity, it is likely that it is under attack and it is likely that there is larger number of attackers consuming the web-server's resource than legitimate clients. Thus although the requests to be dropped are selected at random, there is a greater chance of selecting a request from an attacker.

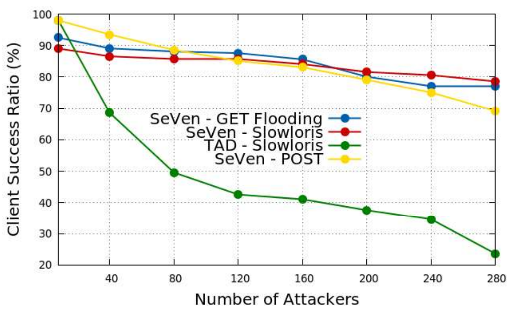
4. SIMULATION AND EXPERIMENTAL RESULTS

This section describes our main simulation and experimental results. [Dantas 2015] contains further details of the Maude formalization and C++ implementation of SeVen.

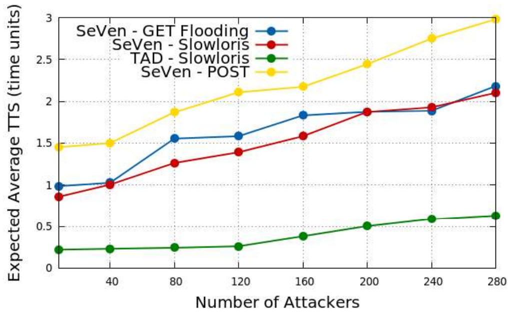
4.1. Simulation Results

In our simulations, we compare our results with a defense similar to the ones in the literature based on Traffic Analysis. At a glance, this defense keeps track of the request that it received, in particular, the number of pieces of data and the id. For example, if two consecutive Slowloris requests, i.e., "x-1:aa", are received by the application with the same id and the same number of pieces of data, the defense believes that it is an attack and simply blocks subsequent requests from this id. We call this simple defense as Traffic Analysis Defense (TAD). As shown in [de Almeida 2013], TAD is capable of mitigating Slowloris attacks generated by a small number of attackers. However, once there is a great number of attackers, this simple defense mechanism does not have a good performance any longer, as predicted by [Kumar and Selvakumar 2013]. For more details of such a defense, we refer to [de Almeida 2013, Dantas et al. 2014, Dantas 2015].

For our simulations, we used the following scenarios:



(a) Client Success Ratio.



(b) Average TTS

Figure 2. Simulations of Client Success Ratio and Average TTS where $k = 12$.

1. SeVen – GET FLOOD: we simulated an HTTP GET flooding attack when the target application used our defense SeVen;
2. SeVen – Slowloris: we simulated an Slowloris attack when the target application used our defense SeVen;
3. SeVen – POST: we simulated an HTTP POST attack when the target application used our defense SeVen;
4. TAD – Slowloris: we simulated an HTTP Slowloris attack when the target application used the defense TAD.

Figure 2 contains the results of the simulations we carried out using different number of attacker that we measure how many (of the 200) clients were successful in receiving an acknowledgment from the target application stating that their request has been completed and how much time in average that it takes for a successful client to receive an acknowledgment that his request was successfully processed by the application. Figure 2(a) depicts the evolution of Client Success Ratio when we increase the number of attackers. For all three attacks, the performance of SeVen is similar. When there are 280 attackers, as opposed to 200 clients, the application running SeVen still can maintain a high level of availability, namely superior to 70%. On the other hand, TAD has similar or even superior results when there is a small number of attackers, but then it falls drastically with the increase of attackers. This is in conformance with the evaluation in [Kumar and Selvakumar 2013] that filtering defenses do not perform well when there is a great number of attackers.

Figure 2(b) depicts the average TTS when we vary the number of attackers. All scenarios involving SeVen have a higher TTS than the scenario using TAD. This is expected, as SeVen only answers request when the round of t_S elapses, while TAD answers immediately. Moreover, as expected, there is an increase on the average TTS. In all scenarios involving SeVen the TTS doubles when we increase the number of attackers from 8 to 280, which seems reasonable. The same happens to the TTS when using TAD. Notice, however, that when using TAD the number of clients that actually receive an acknowledgment reduces considerably with the number of attackers, as depicted in Figure 2(a) and the TTS is only computed using the Successful Clients. So despite TAD having a smaller TTS, the number of clients that have their request completely processed is much less.

4.2. Experimental Results

We carried out a number of experiments on the network to check the robustness of SeVen against Slowloris and HTTP POST attacks. In order to have a realistic DDoS attack, we set both target application and SeVen in Vitória-ES/BR and carried out the attack using

Table 1. Result Summary

	Without SeVen		With SeVen	
	Success Rate	TTS	Success Rate	TTS
No Attack	100.0%	0.01s	100.0%	0.03s
POST	0.0%	∞	97.25%	0.05s
Slowloris	0.0%	∞	94.47%	0.06s

distributed machines from João Pessoa-PB/BR. Moreover, we used a benchmark tool, named Siege [Siege 2014], to generate legitimate client requests also from João Pessoa.

We used an Apache web-server 2.4.7 running in a Ubuntu machine, which was configured with 200 sockets, i.e., the maximum number of connections that the web-server can process. The timeout for the web-server was set to 40 seconds. This means that if a socket remains silent for longer than 40 seconds, then the request arrived at that socket is dropped. We generated (using Siege) 50 clients each 5 seconds, which means in average a rate of 10 clients per second. The goal is to keep 1/4 of the buffer occupied by legitimate clients. For generating the Slowloris and HTTP POST attacks, we used Slowloris [slowloris 2013] and Switchblade [SwitchBlade 2015] tools respectively. We generated 250 attackers every 35 seconds, i.e., a rate of 7.14 attackers per second, which is smaller that our client rate but enough for denying service to a small/medium size web-server. It supports our claim that ADDoS do not need to generate a large amount of traffic to deny a target service. Finally, we set SeVen as a proxy and the round time of SeVen was set to 10 ms.

Table 1 summarizes the results on the Success Rate and TTS for when the web-server is not under an attack, and for the Slowloris and HTTP POST attacks for the scenarios when using SeVen and without SeVen. We observe that the web-server maintains great levels of availability when using SeVen for both attacks. On the other hand, when SeVen is not running the web-server is not able to respond any requests from legitimate clients. In addition, we also measured the slight overhead added by SeVen working as a proxy, which in our experiments was 0.02 seconds.

Our experiments demonstrate that SeVen is indeed effective for mitigating Slowloris and HTTP POST attacks as already showed in our simulations using formal methods. This indicate that formal methods is a suitable tool for specifying defense for mitigating Distributed Denial of Services attacks allowing to increase our confidence on the proposed defense before actual implementation.

5. Conclusions and Future Work

The outcomes of the Thesis We consider the outcome of the whole project a success. At the time this paper was written, we have three published papers [Dantas et al. 2014, Dantas et al. 2016, Lemos et al. 2016], two concluded bachelor theses and two ongoing master theses on the topic. Due to such a success, we also have a solid ongoing project funded by RNP (www.rnp.br) in which we are currently deploying SeVen to protect the web-services of different educational institutions. In addition, we are also investigating how SeVen can be incorporated into existing services such as those used by Fone@RNP.

We propose a novel defense for Application Layer DDoS attacks (ADDoS), called

SeVen. We demonstrated by simulations and experiments that SeVen can be used to mitigate Slowloris and HTTP POST attacks. We already have carried out experiments over the network for the GET Flooding attack with great level of availability, however in order to stick with the thesis results we did not mention it in this paper. We have tested it with other web-servers, e.g., nginx. We are implementing the alternative selective strategies, e.g., using different probability distributions. In order to protect multiple servers, we are incorporating load balancing strategies into SeVen. We are also investigating whether SeVen can be used to mitigate second order DoS attacks [Olivo et al. 2015].

References

[AlTurki and Meseguer 2011] AlTurki, M. and Meseguer, J. (2011). Pvesta: A parallel statistical model checking and quantitative analysis tool. In *CALCO*, pages 386–392.

[Clavel et al. 2007] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. (2007). *All About Maude: A High-Performance Logical Framework*.

[Dantas 2015] Dantas, Y. G. (2015). Estratégias para tratamento de ataques de negação de serviço na camada de aplicação em redes ip – http://tede.biblioteca.ufpb.br/bitstream/tede/7841/2/arquivototal.pdf. Master Thesis in Portuguese.

[Dantas et al. 2016] Dantas, Y. G., Lemos, M. O. O., Fonseca, I., and Nigam, V. (2016). Formal specification and verification of a selective defense for tdos attacks. In *11th International Workshop on Rewriting Logic and its Applications (WRLA)*, LNCS.

[Dantas et al. 2014] Dantas, Y. G., Nigam, V., and Fonseca, I. E. (2014). A selective defense for application layer DDoS attacks. In *IEEE Joint Intelligence and Security Informatics Conference, JISIC 2014*, pages 75–82. IEEE.

[de Almeida 2013] de Almeida, L. C. (2013). Ferramenta computacional para identificação e bloqueio de ataques de negação de serviço em aplicações web. Master Thesis in Portuguese.

[Khanna et al. 2008] Khanna, S., Venkatesh, S. S., Fatemieh, O., Khan, F., and Gunter, C. A. (2008). Adaptive selectiveverification. In *INFOCOM*, pages 529–537.

[Khanna et al. 2012] Khanna, S., Venkatesh, S. S., Fatemieh, O., Khan, F., and Gunter, C. A. (2012). Adaptive selective verification: An efficient adaptive countermeasure to thwart dos attacks. *IEEE/ACM Trans. Netw.*, 20(3):715–728.

[Kumar and Selvakumar 2013] Kumar, P. A. R. and Selvakumar, S. (2013). Detection of distributed denial of service attacks using an ensemble of adaptive and hybrid neuro-fuzzy systems. *Computer Communications*, 36(3):303 – 319.

[Lemos et al. 2016] Lemos, M. O. O., Dantas, Y. G., Nigam, V., and Sampaio, G. (2016). A selective defense for mitigating coordinated call attacks. In (SBRC).

[Olivo et al. 2015] Olivo, O., Dillig, I., and Lin, C. (2015). Detecting and exploiting second order denial-of-service vulnerabilities in web applications. In CCS, 2015.

[r-u-dead yet 2013] r-u-dead yet (2013). https://code.google.com/p/r-u-dead-yet/.

[RFC 1999] RFC, H. (1999). https://tools.ietf.org/html/rfc2616.

[Shankesi et al. 2009] Shankesi, R., AlTurki, M., Sasse, R., Gunter, C. A., and Meseguer, J. (2009). Model-checking DoS amplification for VoIP session initiation. In *ESORICS*, pages 390–405.

[Siege 2014] Siege (2014). https://www.joedog.org/siege-home/.

[slowloris 2013] slowloris (2013). http://ha.ckers.org/slowloris/.

[SwitchBlade 2015] SwitchBlade (2015). http://www.proactiverisk.com/switchblade/.

[Zargar et al. 2013] Zargar, S. T., Joshi, J., and Tipper, D. (2013). A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks. *IEEE Communications Surveys and Tutorials*, 15(4):2046–2069.