A Selective Defense for Application Layer DDoS Attacks

Yuri G. Dantas Vivek Nigam Iguatemi E. Fonseca,
Federal University of Paraíba, João Pessoa, Paraíba, Brazil
Emails: {ygd,vivek,iguatemi}@ci.ufpb.br

***Abstract*—Distributed Denial of Service (DDoS) attacks remain among the most dangerous and noticeable attacks on the Internet. Differently from previous attacks, many recent DDoS attacks have not been carried out over the network layer, but over the application layer. The main difference is that in the latter, an attacker can target a particular application of the server, while leaving the remaining applications still available, thus generating less traffic and being harder to detect. Such attacks are possible by exploiting application layer protocols used by the target application. This paper proposes a novel defense for Application Layer DDoS attacks (ADDoS) based on the Adaptive Selective Verification (ASV) defense used for mitigating Network Layer DDoS attacks. We formalize our defense mechanism in the computational system Maude and demonstrate by using the statistical model checker PVeStA that it can be used to prevent ADDoS. In particular, we show that even in the presence of a great number of attackers, an application running our defense still has high levels of availability. Moreover, we compare our results to a defense based on traffic monitoring proposed in the literature and show that our defense is more robust and also leads to less traffic.**

I. Introduction

Distributed Denial of Service attacks (DDoS) have been a serious concern to network administrators since the origins of the Internet. Many DDoS attacks can be easily deployed and cause great damage by blocking the availability of services. An attack is normally realized by recruiting zombies through the use of worms. These recruits then send a large number of packages to a server in a coordinated fashion so to overload the server's processing capacity making it unavailable to honest clients.

In the past years, a group calling themselves "Anonymous" has carried out a number of flooding attacks on organizations such as Mastercard.com, PayPal, Visa.com and PostFinance [1]. This attack affected 9 major U.S. banks, namely Bank of America, Citigroup, Wells Fargo, U.S. Bancorp, PNC, Capital One, Fifth Third Bank, BB&T, and HSBC. They have still been suffering attacks from a a foreign hacktivist group called "Izz ad-Din al-Qassam Cyber Fighters" [14], affecting the availability of several online banking sites. Last year, an attack involving the group Spamhaus and the company Cyberbunker carried out a DDoS attack which generated around 300 Gbps of useless traffic [8], one of the biggest attack until then.

However, with the introduction of recent defense mechanisms, attacks carried out by simply downloading a tool and launching an attack over the Network Layer against a server are not longer a major threat [24]. For example, the defense Adaptive Selective Verification (ASV) [12], [13] has been successfully used for mitigating many DDoS attacks. Effective attacks have become much more involved. Attacks over the network layer, which are carried out by simply sending a large amount of packages, are now being replaced by attacks over the application layer which take into consideration the vulnerabilities of protocols used by applications such as HTTP. Examples of such Application Layer DDoS attacks (ADDoS) include VoIP amplification flooding attack [21], [24], HTTP GET flooding attack [22], [24], HTTP PRAGMA attack [9], [22], HTTP POST attack [19] and the recent NTP attack [18].

Application Layer DDoS (ADDoS) are particularly dangerous as they can be used to attack a particular application of the server, such as a web-server, while leaving the remaining applications still available. This means that the number of attackers and the traffic needed to carry out an attack is much less than traditional Network Layer DDoS Attacks and therefore harder to detect.

The contribution of this paper is three-fold:

1) We formalize three different ADDoS attacks in the the computational tool Maude [7], namely HTTP GET flooding, HTTP PRAGMA and the HTTP POST attacks;
2) We propose a novel defense against ADDoS attacks, called SeVen, based on ASV [12]. As ASV was designed for mitigating Network 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 PRAGMA attacks;
3) We formalize SeVen in Maude [7] and validate our defense by simulation using the statistical model checker tool PVeStA [4]. We compare our results with a defense similar to the ones in the literature based on Traffic Analysis. We test our defense against the three attacks described above in item 1. Our simulations show that SeVen is both more robust and also leads to better traffic patterns than the defense based on Traffic Analysis (TAD), but that our defense leads to a smooth overhead on the Total Time of Response (TTS).

Since ADDoS attacks are relatively new, there are not many studies on how to mitigate them. Indeed, while there is a huge amount of real traffic data of Network Layer DDoS attacks

978-1-4799-6364-5/14 $31.00 © 2014 IEEE
DOI 10.1109/JISIC.2014.21

to work with, such as the databases CAIDA, KDD Cup e UNINA [15], the same is not the case for ADDoS attacks. Thus providing meaningful statistical simulations for ADDoS attacks using formal method tools is key for understanding ways for mitigating such attacks.

For instance, to the best of our knowledge, there are no defenses in the literature for mitigating HTTP POST attacks. This attack is particularly tough as it can quickly consume all resources of a target application without requiring many attackers. One can easily verify this by using the tool r-u-dead-yet available at [19]. Our simulations show that with the use of SeVen, an application can still be available to up to 70% of clients even in the presence of a great number of attackers carrying out such an HTTP POST attack.

We start in Section II by describing the three attacks above that we formalized in Maude. Then in Section III, we introduce the defense SeVen and another defense based on Traffic Analysis similar to the ones in the literature. Section IV describes the simulation scenarios that we carried out, and discusses our results. Finally in Section V we conclude by discussing related work and pointing out future work directions. Finally, we point out that the implementation used to carry out our simulations is available for download at [2].

II. Application Layer DDoS Attacks

We describe two types of Application Layer DDoS attacks namely the Flooding and the Slowloris attacks [24]. In particular we formalize one Flooding attack and two Slowloris attacks[1] in Maude. We show in Section IV that our defense, SeVen, described in Section III, can be used to keep an application available despite the presence of a number of attackers.

A. Flooding Attacks

Flooding attacks [21], [24] are very similar to Network 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.

We formalize the HTTP GET attack. Its attack pattern is as shown in Figure 1. As one can observe by the attack, the target application receives a large number of messages (HTTP GET). First the attacker sends a HTTP GET request to the target application to probe whether it is available. If the attacker receives an acknowledgment from the target application, then the attacker simply sends periodically new HTTP GET requests without waiting for further acknowledgment messages, as specified in the automata depicted in Figure 1.

B. Slowloris Attacks

While Flooding attacks are similar to Network Layer DDoS attacks, Slowloris attacks exploit the protocols used by the target application and are able to make an application unavailable by using much less traffic and fewer attackers. We discuss below two such attacks, namely the HTTP PRAGMA and HTTP POST attacks. Both of these attacks can be carried out by tools that can be easily be found on the Internet [19].

[1]Slowloris is a tool that can be used to carry out Application Layer DDoS attacks.

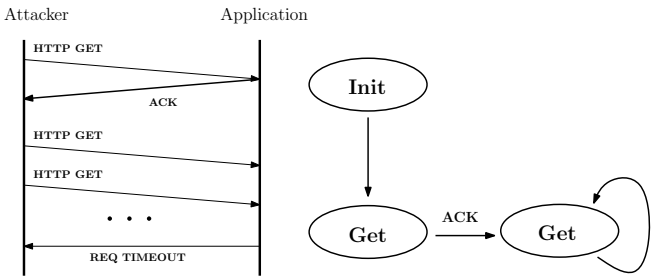


Fig. 1. Sequence of messages used to realize an HTTP GET attack and the finite state machine of the attacker when carrying out this attack.

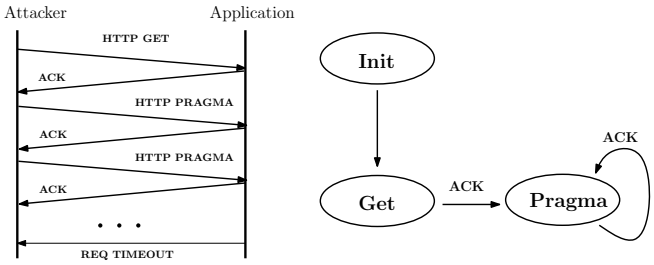


Fig. 2. Sequence of messages used to realize an HTTP PRAGMA attack and the finite state machine of the attacker when carrying out this attack.

a) HTTP PRAGMA Attack: This attack exploits an HTTP field called PRAGMA, which is a header field intended for use in HTTP protocol requests. It is used by the browser to tell the application and any intermediate caches that it wants a fresh version of a previously requested resource. In practice, when a PRAGMA message from a client A is received by the application, the application resets the timeout of the connection with A, thus allowing A's connection to remain for a longer time in the application's memory. This field does no longer have a purpose in HTTP version 1.1, but for compatibility reasons it is still available.

The HTTP PRAGMA attack is as depicted in Figure 2. In particular, the attacker sends a GET message in order for its request to be allocated in the application's memory, and then near to the timeout of the connection, he sends a PRAGMA which resets the corresponding timeout, allowing him to continue consuming resources of the application. As shown in [9], [24], a single attacker with few resources is able to realize an attack to an application in such a way.

The finite state machine shown in Figure 2 specifies the states of the attacker when realizing an HTTP PRAGMA attack. First he sends a GET message to the target application. He then waits for the acknowledgment from the target application for his GET message. Once this acknowledgment is received, he periodically sends a new PRAGMA and waits for

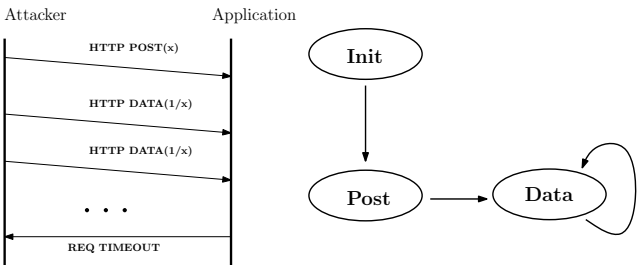


Fig. 3. Sequence of messages used to realize an HTTP POST attack and the finite state machine of the attacker when carrying out this attack.

an acknowledgment from the target application, as specified by the automata in Figure 2 .

b) HTTP POST Attack: This attack is a bit more ingenious exploiting POST requests of the HTTP protocol. A POST request is sent whenever a client completes a form. Its function is to inform the application, *e.g.*, a web-server, the size of the data entered into the form. Once the application receives a POST request, it waits for the contents of the form, which may be transmitted using one or more subsequent HTTP DATA messages.

A typical sequence of messages used in an attack is depicted in Figure 3. The attacker sends to the application a POST request with the size, x, of the data to be transmitted. The attacker typically uses a big number. The application then waits until either a timeout occurs or he receives all pieces of the entry. However, instead of the attacker sending big pieces of the entry, as honest clients would behave, he proceeds by sending to the application a small piece of data per message, thus occupying the application's resources for a longer time. In Figure 3, the attacker sends one byte at a time until either a timeout is reached or the attacker sends x bytes.

The finite state machine that governs the attacker is also shown in Figure 3. In particular, the attacker does not wait for any acknowledgment of the target application, but simply sends a POST request and then periodically sends DATA requests.

III. Defenses

This section contains two types of defenses, one is our novel defense, SeVen, based on the Adaptive Selective Verification (ASV) [12], [13] and the second defense, called TAD, is based on Traffic Analysis, similar to many defenses in the literature [6], [15]. In Section IV, we use TAD as a reference defense in order to see how our proposal compares with the state-of-the-art.

A. Selective Verification in Application Layer (SeVen)

Our novel defense Selective Verification in Application Layer (SeVen[2]) is based on ASV [12]. While ASV was designed for mitigating Network Layer DDoS attacks assuming a stateless sync-ack communication between client and servers, SeVen is designed to mitigate Application Layer DDoS attacks and assumes state-dependent protocols. This is a key difference as many ADDoS attacks rely on protocols that are not stateless, *e.g.*, the Slowloris attacks described above.

As with ASV, 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 an internal buffer. 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 $\mathcal{P}$ denotes the request *partially processed*, and $\mathcal{R}$ the requests *received and that are to 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.

For simplicity, we assume that each element of $\mathcal{R}$ and $\mathcal{P}$ is a tuple of the form $\langle id, N, T \rangle$, where id is a unique identifier of the request being processed and N and T are natural numbers. We assume that two different requests in a buffer have necessarily different identifiers. That is, it cannot be the case that there are two occurrences of requests with the same identifier. The number T denotes the total number of pieces of data to be processed by a request, while the number N has different denotations depending on the buffer: in $\mathcal{P}$, N denotes the number of pieces of data already processed, while in $\mathcal{R}$ the number of pieces of data that have been received and are to be processed. In a real implementation, requests would contain other information, *e.g.*, the identifier would be composed by the IP address of the requesting agent and socket number, etc. We assume that these are retrievable from the id of requests.

The two buffers $\mathcal{R}$ and $\mathcal{P}$ reflect the requests that the application is handling, so for each request being processed in $\mathcal{P}$ there is a corresponding one in $\mathcal{R}$ and vice-versa, *i.e.*,

$$\langle id, N, T \rangle \in \mathcal{P} \text{ if and only if } \langle id, M, T \rangle \in \mathcal{R}$$

for any values N, M, T and identifier id, where $\in$ is the list membership function.

For example, consider the following buffers:

$$\mathcal{P}_1 = [\langle id_1, 30, 100 \rangle, \langle id_2, 5, 10 \rangle, \langle id_3, 15, 100 \rangle]$$
$$\mathcal{R}_1 = [\langle id_1, 2, 100 \rangle, \langle id_2, 0, 10 \rangle, \langle id_3, 85, 100 \rangle]$$

They specify that the application has received and processed, respectively, $30, 5, 15$ pieces of data for the requests identified with id_1, id_2, id_3, while it has only received, but not yet processed, respectively, $2, 0, 85$ more pieces of data for the requests identified with id_1, id_2, id_3.

Finally, we also assume 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.

The bound k is key for specifying the behavior of SeVen. During a round, SeVen does not reject, *i.e.*, drop, any requests

[2]as the number of the Application Layer in the OSI model.

until the number k of requests have arrived, since we are assuming that the application can handle k requests. However, if the buffer $\mathcal{R}$ has already k requests and yet another request 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, which means that the requests currently in the buffer are not affected;
- or it may decide to keep the incoming request, which means that one request in the buffer should be replaced by the new request.

These choices will be governed by probability distributions as detailed in the following section. At the end of the round, SeVen processes all the request that remain in the buffer.

Intuitively, once the buffer $\mathcal{P}$ of the application is full, the requests in $\mathcal{R}$ and the new incoming requests are competing for some time of the application's processing power. As we assume an upper-bound on the number of requests it can process, reflected by the upper-bound k on the size of $\mathcal{P}$, some packages should be lost. This type of defense works because whenever the application is under attack, the attacker sends many more messages and/or attempts to use longer the application's processing power, and therefore the attacker's request has a higher probability of being dropped allowing more time for processing the requests of honest clients.

B. SeVen Defense Algorithm

At the beginning of a round, all values in $\mathcal{R}$ are of the form $\langle id_i, 0, T\rangle$, that is the value associated to a request is 0, reflecting the fact that at the beginning of a round no requests have been received and therefore there is no content to be processed. Moreover, PMod is also zero.

During a round of t_S seconds, the application behaves as follows, where $\setminus$ is the operation that removes an element from a list, and @ is the operation that appends two lists:

1) While the buffer $\mathcal{P}$ is not full, *i.e.*, the length of $\mathcal{P}$ is less than k, the application keeps accumulating request as follows: Suppose that an incoming request is of the form $\langle id, N, T\rangle$:
   a) If there is a request $\langle id, M, T\rangle$ in $\mathcal{P}$ with the same identifier id requesting and the same total of T pieces of information to be processed, it means that the request received is a continuation of a request that has been partially processed by the application. We then set $\mathcal{R} := (\mathcal{R} \setminus \langle id, M, T\rangle)@[\langle id, N+M, T\rangle]$, that is, it updates the request to be processed;
   b) Otherwise, the request received is new, and we set $\mathcal{R} := \mathcal{R}@[\langle id, N, T\rangle]$ and $\mathcal{P} := \mathcal{P}@[\langle id, 0, T\rangle]$. Notice that this request has not been processed, which is specified by the the value 0 used for this request in $\mathcal{P}$.
2) If the buffer $\mathcal{P}$ is full, *i.e.*, the number of elements in $\mathcal{P}$ is k, and a new incoming request $r = \langle id, N, T\rangle$ arrives:
   a) Set $\texttt{PMod} := \texttt{PMod} + 1$;
   b) If $\langle id, M, T\rangle \in \mathcal{R}$, then the application updates it, *i.e.*, $\mathcal{R} := (\mathcal{R} \setminus \langle id, M, T\rangle)@[\langle id, N + M, T\rangle]$;
   c) Otherwise, the request r is a new request. This means that the application will not be able to handle all requests in $\mathcal{R}$ and also the incoming request r. Thus, it should decide to drop some request, either the request r or some request in $\mathcal{R}$. It acts as follows:
      i) It decides whether to keep the request r or ignore. It generates a random number $rand$. If $rand \leq \texttt{Prob}$, where Prob is given below, then it decides to keep the request r, otherwise the application drops r:
      $$\texttt{Prob} = \frac{k}{k + \texttt{PMod}}$$
      Notice the use of PMod in the denominator. It has the effect of decreasing the probability that new requests are accepted.
      ii) If it decides to drop r, a message is send to the agent id informing that the agent's request was not processed and the buffers $\mathcal{P}, \mathcal{R}$ are left unchanged;
      iii) If it decides to keep r, then it chooses according to a uniform probability distribution (U), which request in $\mathcal{P}$ and $\mathcal{R}$ it is going to stop processing.
      Say that it chooses the request with identification id_d, and let $\langle id_d, M, T\rangle$ in $\mathcal{R}$ and $\langle id_d, N, T\rangle$ in $\mathcal{P}$ be the corresponding request for id_d, then we set the buffers as follows: $\mathcal{R} := (\mathcal{R} \setminus \langle id_d, M, T\rangle)@[\langle id, N, T\rangle]$ and $\mathcal{P} := (\mathcal{P} \setminus \langle id_d, N, T\rangle)@[\langle id, 0, T\rangle]$. Moreover, the application sends a message to id_d stating that his request has been dropped.
3) Once a round is over, that is, t_S seconds have elapsed, the application processes the request (that survived) in $\mathcal{R}$. In particular, it performs the following steps:
   a) Set $\texttt{PMod} := 0$;
   b) For each $\langle id, N, T\rangle \in \mathcal{R}$, such that $N > 0$ we do the following: Let $\langle id, M, T\rangle \in \mathcal{P}$ be the corresponding request in $\mathcal{P}$, then we set $\mathcal{R} := (\mathcal{R} \setminus \langle id, N, T\rangle)@[\langle id, 0, T\rangle]$ and $\mathcal{P} := (\mathcal{P} \setminus \langle id, M, T\rangle)@[\langle id, M + N, T\rangle]$.
   c) For each $\langle id, M, T\rangle \in \mathcal{P}$ such that $M \geq T$ and with corresponding request $\langle id, N, T\rangle \in \mathcal{R}$, we set $\mathcal{R} := \mathcal{R} \setminus \langle id, N, T\rangle$ and $\mathcal{P} := \mathcal{P} \setminus \langle id, M, T\rangle$, specifying that this request has been completed and no longer needs to be using the application's resources. Moreover, the application sends an acknowledgment message to each one the agents id for which requests have been completed.

C. Example

Consider the buffers $\mathcal{P}_1$ and $\mathcal{R}_1$:

$$\mathcal{P}_1 = [\langle id_1, 30, 100\rangle, \langle id_2, 5, 10\rangle, \langle id_3, 15, 100\rangle]$$
$$\mathcal{R}_1 = [\langle id_1, 2, 100\rangle, \langle id_2, 0, 10\rangle, \langle id_3, 85, 100\rangle]$$

Assume that PMod is zero, that the upper-bound of the application is $k = 4$ and moreover that a round of t_S seconds just started. If the application receives the request $\langle id_4, 2, 10\rangle$, then the buffers evolve to the following, where corresponding requests are added to both buffers (Step 1b):

$$\mathcal{P}_2 = [\langle id_1, 30, 100\rangle, \langle id_2, 5, 10\rangle, \langle id_3, 15, 100\rangle, \langle id_4, 0, 10\rangle]$$
$$\mathcal{R}_2 = [\langle id_1, 2, 100\rangle, \langle id_2, 0, 10\rangle, \langle id_3, 85, 100\rangle, \langle id_4, 2, 10\rangle]$$

Notice, that the application has reached his upper-bound, *i.e.*, the buffer is full. Now consider that the new request $\langle id_5, 4, 10\rangle$ arrives. Since the buffer is full, the application will "throw a coin" generating a random number and comparing it with `Prob` (Step 2(c)i) to decide whether it keeps the new request or not. Say that it decides that the request $\langle id_5, 4, 10\rangle$ should be kept. In this case, the application "throws a coin" according to probability distribution `U` (Step 2(c)iii). Say that it chooses that the request with identification number id_2 should be dropped, then the buffers evolve to:

$$\mathcal{P}_3 = [\langle id_1, 30, 100\rangle, \langle id_3, 15, 100\rangle, \langle id_4, 0, 10\rangle, \langle id_5, 0, 10\rangle]$$
$$\mathcal{R}_3 = [\langle id_1, 2, 100\rangle, \langle id_3, 85, 100\rangle, \langle id_4, 2, 10\rangle, \langle id_5, 4, 10\rangle]$$

and a corresponding message is sent to the agent id_2 informing him that his request has been dropped.

Assume now that the round is over, that is, t_S seconds have elapsed. The application processes the requests in the buffer $\mathcal{R}$ (Step 3b), obtaining:

$$\mathcal{P}_3 = [\langle id_1, 32, 100\rangle, \langle id_3, 100, 100\rangle, \langle id_4, 2, 10\rangle, \langle id_5, 4, 10\rangle]$$
$$\mathcal{R}_3 = [\langle id_1, 0, 100\rangle, \langle id_3, 0, 100\rangle, \langle id_4, 0, 10\rangle, \langle id_5, 0, 10\rangle]$$

Finally, since the request id_3 has been completed, *i.e.*, the application received 100 of the 100 pieces of information, the application removes it from both buffers (Step 3c), resulting in:

$$\mathcal{P}_3 = [\langle id_1, 32, 100\rangle, \langle id_4, 2, 10\rangle, \langle id_5, 4, 10\rangle]$$
$$\mathcal{R}_3 = [\langle id_1, 0, 100\rangle, \langle id_4, 0, 10\rangle, \langle id_5, 0, 10\rangle]$$

It also sends an acknowledgment message to the agent id_3 stating that its request has been completed.

Remarks: Notice that instead of two buffers, we could also have used a single one, which would keep track the processed and received pieces of information. However, we chose to use two lists as it is closer to what actually happens in practice and more importantly it allows one to specialize this defense to the use of other specific protocols, such as the protocols used in VoIP. In fact, we can add much more information to these the request stored in the buffers and create more elaborate defenses. For example, we could use different upper bounds for $\mathcal{P}$ and $\mathcal{R}$, or include more concrete information, such as the location of the request or the time that requests of an identifier are in the buffers. Some of these data may also be used to build parameters for new probability distributions for deciding to keep a request and to decide which request in the buffer to drop, *e.g.*, requests that have been longer in the buffer should have higher probability of being dropped. All these options are left to future work.

Comparison to ASV Although SeVen was based on the defense ASV, there are some important differences. In particular, ASV was designed to mitigate Network Layer DDoS attacks, while SeVen is designed for mitigating Application Layer DDoS attacks. Thus, the assumption used in ASV is that the client-server communication is a stateless sync-ack interaction. The behavior of SeVen on the other hand is not stateless, but depends on the number of pieces of data already processed. For instance, an application using SeVen sends an acknowledgment only when the total payload is received. This is key for defending against the HTTP POST attack described in Section II. Indeed, formalizing a defense with effects and state that can be used against ADDoS attacks was left as future work in [10].

D. A Defense based on Traffic Analysis

Many of the tools available for carrying out ADDoS, such as Slowloris and r-u-dead-yet [19], use a regular traffic pattern of attack. For instance, when carrying out an HTTP PRAGMA attack, Slowloris uses the same size of packages for each one of the PRAGMA messages and sends PRAGMA messages with a fixed time periodicity. This has lead to defenses for attacks carried out by such tools based on these regularities. There is a number of techniques used to detect these patterns, for instance, machine learning technique [15], Markov-Chains [23], or hard-wired into the defense [9]. We formalized a simplified version of the latter, which has been used for HTTP PRAGMA attacks only. Our main purpose is to have some comparison with the literature, as up to the best of our knowledge, no defense against Application Layer DDoS attacks has yet been formalized and no statistical results are available; only experimental results on the network using a small number of attackers [9] are available.

In our formalization, this defense works as follows: it keeps track of the request that it received, in particular, the number of pieces of data and the id. If two consecutive PRAGMA requests 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 of this id. We call this simple defense as Traffic Analysis Defense (TAD).

As shown in [9], TAD is capable of mitigating HTTP PRAGMA attacks generated by a small number of attackers using the tool Slowloris. This is confirmed by our simulation results described in Section IV. However, once there is a great number of attackers, this simple defense mechanism does not have a good performance any longer, as predicted by [15].

Finally, we notice that up to the best of our knowledge, we do not know of any defense mechanism designed to mitigate HTTP POST attacks. Thus in our simulation, we only use SeVen as defense. Nevertheless, our simulations using SeVen demonstrate that it is a powerful defense against HTTP POST attacks. We leave the experimentation in real network as future work.

IV. Formal Model and Simulation Results

We now describe our simulation results and the assumptions used. For our formalization in Maude, we follow [10], [11] and use an Actor Model, where attackers, clients, and applications are actors sending and receiving messages. Moreover, in order to use PVeStA, the use of the probabilistic distribution should be the only source of non-determinism. That is, all rewrite rules should be deterministic once a sample has been decided on. Therefore, as in [10], [11], we use a scheduler,

that maintains a queue of messages, which are going to be transmitted and processed by agents and finally, clients and attackers are generated by a generator actor.

Our formalization is parametric in the following values:

- Network Timeout – t_C: This parameter models the time that Network Layer sends a Connection Timeout to the Application Layer. This means that if the the application does not receive any message from an agent in t_C seconds, then the agent's connection is terminated. When this happens, the agent is also removed from both $\mathcal{P}$ and $\mathcal{R}$ buffers the application defense. In our simulations, we use 0.4s.
- SeVen Round Time – t_S: This is the time that SeVen waits accumulating requests, as described in Section III. In our simulations, we use 0.4s.
- Size of Buffer – k: This is the upper-bound on the size of $\mathcal{P}$, denoting the processing capacity of the application.
- Number of clients ($countClient$) and attackers: One can also configure the number of clients and attackers in our simulations. In all our simulations, we fixed the number of clients to $countClient = 200$ clients;
- Total time of the simulation - $total$: This is the total time of the simulation using PVeStA. We used in our simulations $total$ equal to 40s, similar to the time used in [10];
- Delay of the Network: We also assumed a delay of 0.1s of message in the network;
- Degree of confidence for the simulation: Our simulations were carried out with a degree of confidence of 99% (see [3], [20] for more details on probabilistic model checking).

In our simulation, we used different quality measures are specified by expression of the QuaTEx quantititative, probabilistic temporal logic defined in [3]. We perform statistical model checking of our defense in the sense of [20]: once a QuaTEx formula and desired degree of confidence are specified, a sufficiently large number of Monte Carlo simulations are carried out allowing for the verification of the QuaTEx formula. In this paper, the Monte Carlo simulations are carried out by the computational tool Maude [7] and the statistical model checking is carried out by the PVeStA

The QuaTEx formulas, *i.e.*, the quality measures, that we use in our simulations are defined below. The operator $\bigcirc$ is a temporal modality that specifies the advancement of the global time to the time of the next event (see [3] for more details).

- Client Success Ratio – This measures how many (of the 200) clients were successful in receiving an acknowledgment from the target application stating that their request has been completed. This measure is specified by the following QuaTex expression, where $countSuccessful$ is a counter starting at zero and is incremented whenever a client receives an acknowledgment that his request was successfully processed by the application:

$$successRatio(total) = \text{ if } time > total \text{ then } \frac{countSuccessful}{countClient} \text{ else } \bigcirc successRatio(total)$$

- Network Overhead – This measures how many requests both from clients and attackers were transmitted in the network. This measure is specified by the following QuaTex expression, where $countRequest$ is counter starting at zero and is incremented whenever a client or an attacker sends a request:

$$requests(total) = \text{ if } time > total \text{ then } countRequests \text{ else } \bigcirc requests(total)$$

- Average TTS – This measures 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. $sumTTS$ is the sum of the TTS of successful clients:

$$avgTTS(total) = \text{ if } time > total \text{ then } \frac{sumTTS}{countSuccessful} \text{ else } \bigcirc avgTTS(total)$$

Finally, the behavior of an attacker was specified in Maude, according to the corresponding attack, namely, by using the finite state machine shown in Figures 1, 2 and 3. We do not distinguish attackers from their Zombies. We assume that both behave similarly and we call them all attackers. In our simulations, we are assuming that attackers are created with a rate of 120 attackers per second. The same is true for client generation. We consider several scenarios, detailed below, with different total number of attackers varying from a total of 8 attackers to 280 attackers.

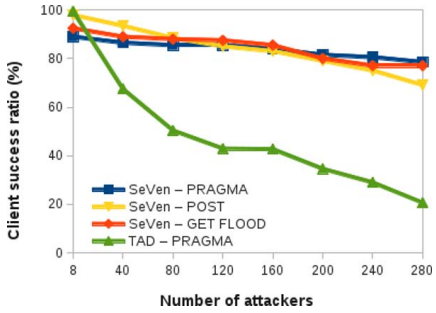
A. Simulation Results

For our simulations, we used the following scenarios:

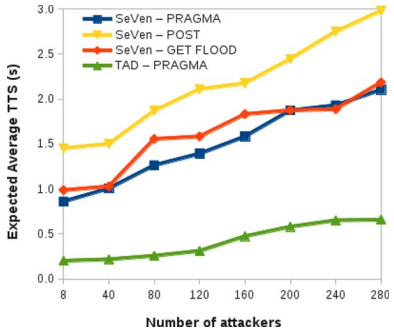
1) SeVen– GET FLOOD: we simulated an HTTP GET flooding attack when the target application used our defense SeVen;
2) SeVen– PRAGMA: we simulated an HTTP PRAGMA attack when the target application used our defense SeVen;
3) SeVen– POST: we simulated an HTTP POST attack when the target application used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

Figure 4 contains all the results of the simulations we carried out using different number of attacker and upper-bound on the size of buffers, and with the measures described above. Figure 4(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 [15] that filtering defenses do not perform well when there is a great number of attackers.

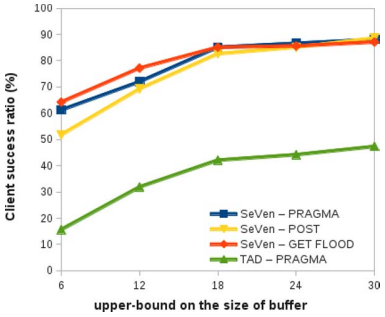
Figure 4(b) depicts the average TTS when we vary the number of attackers. All scenarios involving SeVen have a higher TTS than the scenario using TAD. This is expected, as SeVen only answers request when the round of t_S elapses, while TAD answers immediately. Moreover, as expected, there is an increase on the average TTS. In all scenarios involving



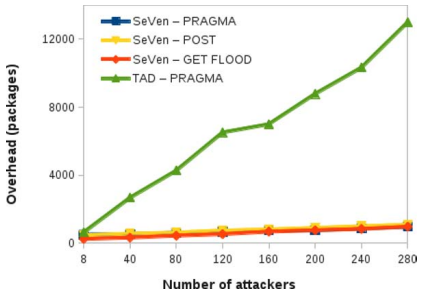
(a) Client Success Ratio versus Number of Attackers, where $k = 12$.



(b) Average TTS, where $k = 12$.



(c) Client Success Ratio versus the Upper-Bound on the Size of the Application Buffer, assuming 280 attackers.



(d) Overhead of Network Traffic versus the Number of Attackers, where $k = 12$.

Fig. 4. Simulations of Client Success Ratio and Average TTS when varying the number of attackers.

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 4(a) and the TTS is only computed using the Successful Clients. So despite TAD having a smaller TTS, the number of clients that actually have their request completely processed is much less.

Figure 4(c) depicts how the Client Success Ratio increases when we increase the upper-bound on the size of the buffer. In all simulations, we assumed the existence of 280 attackers. In the scenarios using SeVen the availability of the application increases quite rapidly from around 50% to 80% when we increase the upper-bound from 6 to 18 and then it does not increase that fast any longer. This suggests that there is an optimal size of buffer that should be enough to mitigate such ADDoS attacks. On the other hand, TAD also increases its availability when we increase the buffer of the target application, but it remains still far from the levels achieved by using SeVen, around 40%.

Finally, Figure 4(d) depicts the evolution of the Network Overhead in our simulations, that is, how many overall requests are sent to the target application. While the Network Overhead increases modestly when using our SeVen scenario, it increases quite rapidly when using TAD. We believe that this has to do with the fact that the use of rounds forces the attacker to wait longer for a message from the application and consequently wait longer to send another request. Take for instance, the HTTP PRAGMA attack. An attacker only sends a new PRAGMA request when it has received an ACK from the previous PRAGMA, because the attacker has to be sure that his connection was not terminated, by for instance the server due to timeout (see Figure 2). The same does not happen when using TAD. As the application immediately sends an ACK in response to a PRAGMA request from the attacker, the attacker can decide to send the next following PRAGMA whenever he wants.

In summary, our simulation results show that SeVen seems to be a good defense against ADDoS flooding and Slowloris attacks as it keeps the target application with high levels of availability despite a great number of attackers.

V. Conclusions and Related Work

This paper introduced a novel defense for Application Layer DDoS attacks (ADDoS), called SeVen, which is based on the defense ASV [12] for Network Layer DDoS attacks. We then demonstrated that SeVen can be used to mitigate a number of attacks, including Flooding and Slowloris attacks, such as HTTP POST attack. Up to the best of our knowledge, this is the first defense for the HTTP POST attack.

There have been other defenses against ADDoS. Most of

them use the traffic pattern of the attack into consideration, such as the round trip of messages, location, and IP, to infer when to filter some messages. There are models based on machine learning techniques, such as Neuro-Fuzzy [15], or models using Markov-Chains [23], or hard-wired into the defense [9]. None of these, however, have been formally verified, but validated using real experiments on the network using a small number of attackers. In [15], the authors mention that filtering defenses seem to work only when there is a small number of attackers. This seems to be supported by our simulations using a defense inspired by the defense in [9]. Finally, none of them have been used to mitigate the HTTP POST attack.

The formalization of DDoS attacks and their defenses has been subject of other papers. For example, Meadows proposed a cost based model in [17], while others use branching temporal logics [16]. This paper takes the approach used in [5], [10], [11], where one formalizes the system in Maude and uses the Statistical Model Checker PVeStA to carry out analyses. However, until now, their formalization relied on the assumption that the Client-Server communication consists of a stateless request-reply interaction, typical of Network Layer DDoS. Our defense and formalization are not stateless, as the behavior of the server will depend on the number of bytes processed. This indeed important to mitigate HTTP POST attacks. This direction was left as future work in [10].

For future work, we are working on extending our model to include more quantitative measures, such as processing times, so to be able to formally verify defenses for asymmetric attacks, such as Amplification Attacks. This was subject of the paper [21], which studied ways to formalize the impact of attackers when carrying such attacks.

We also are currently working on validating our defense by using real experiments over the network. We expect to learn more lessons when carrying out these experiments which will help us refine our defense. For instance, understand better which probabilities for dropping and swapping requests are more suitable for which applications.

ACKNOWLEDGMENT

We thank Leandro C. Almeida for fruitful discussions. We also thank Jonas Eckhardt and Tobias Mühlbauer for helping us with the PVeStA tool and lending us their code to take a look and work with. This work was partially supported by the CNPq and Capes.

REFERENCES

[1] Operation payback cripples mastercard site in revenge for wikileaks ban http://www.theguardian.com/media/2010/dec/08/operation-payback-mastercard-website-wikileaks. 2010.
[2] Seven https://github.com/ygdantas/SeVen.git. 2013.
[3] Gul Agha, José Meseguer, and Koushik Sen. Pmaude: Rewrite-based specification language for probabilistic object systems. *Electron. Notes Theor. Comput. Sci.*, 153(2):213–239, May 2006.
[4] Musab AlTurki and José Meseguer. Pvesta: A parallel statistical model checking and quantitative analysis tool. In *CALCO*, pages 386–392, 2011.
[5] Musab AlTurki, José Meseguer, and Carl A. Gunter. Probabilistic modeling and analysis of dos protection for the asv protocol. *Electr. Notes Theor. Comput. Sci.*, 234:3–18, 2009.
[6] Hakem Beitollahi and Geert Deconinck. Analyzing well-known countermeasures against distributed denial of service attacks. *Computer Communications*, 35(11):1312 – 1332, 2012.
[7] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Mart´-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer.
[8] L. Dave. Global internet slow after "biggest attack in history http://www.bbc.co.uk/news/technology-21954636. 2013.
[9] Leandro C. de Almeida. Ferramenta computacional para identificao e bloqueio de ataques de negao de servio em aplicaes web. Master Thesis in Portuguese, 2013.
[10] Jonas Eckhardt, Tobias Mühlbauer, Musab AlTurki, José Meseguer, and Martin Wirsing. Stable availability under denial of service attacks through formal patterns. In *FASE*, pages 78–93, 2012.
[11] Jonas Eckhardt, Tobias Mühlbauer, José Meseguer, and Martin Wirsing. Statistical model checking for composite actor systems. In *WADT*, pages 143–160, 2012.
[12] Sanjeev Khanna, Santosh S. Venkatesh, Omid Fatemieh, Fariba Khan, and Carl A. Gunter. Adaptive selectiveverification. In *INFOCOM*, pages 529–537, 2008.
[13] Sanjeev Khanna, Santosh S. Venkatesh, Omid Fatemieh, Fariba Khan, and Carl A. Gunter. Adaptive selective verification: An efficient adaptive countermeasure to thwart dos attacks. *IEEE/ACM Trans. Netw.*, 20(3):715–728, 2012.
[14] T. Kitten. Ddos: Lessons from phase 2 attacks http://www.bankinfosecurity.com/ddos-attacks-lessons-from-phase-2-a-5420/op-1. 2013.
[15] P. Arun Raj Kumar and S. Selvakumar. Detection of distributed denial of service attacks using an ensemble of adaptive and hybrid neuro-fuzzy systems. *Computer Communications*, 36(3):303 – 319, 2013.
[16] Ajay Mahimkar and Vitaly Shmatikov. Game-based analysis of denial-of-service prevention protocols. In *CSFW*, pages 287–301, 2005.
[17] Catherine Meadows. A formal framework and evaluation method for network denial of service. In *CSFW*, pages 4–13, 1999.
[18] Matthew Prince. Technical details behind a 400Gbps NTP amplification DDoS attack, http://blog.cloudflare.com/technical-details-behind-a-400gbps-ntp-amplification-ddos-attack. 2013.
[19] r-u-dead yet. https://code.google.com/p/r-u-dead-yet/. 2013.
[20] Koushik Sen, Mahesh Viswanathan, and Gul Agha. On statistical model checking of stochastic systems. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 266–280. Springer, 2005.
[21] Ravinder Shankesi, Musab AlTurki, Ralf Sasse, Carl A. Gunter, and José Meseguer. Model-checking DoS amplification for VoIP session initiation. In *ESORICS*, pages 390–405, 2009.
[22] slowloris. http://ha.ckers.org/slowloris/. 2013.
[23] Yi Xie and Shun-Zheng Yu. Monitoring the application-layer ddos attacks for popular websites. *IEEE/ACM Trans. Netw.*, 17(1):15–25, 2009.
[24] Saman Taghavi Zargar, James Joshi, and David Tipper. A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks. *IEEE Communications Surveys and Tutorials*, 15(4):2046–2069, 2013.