Journal of King Saud University – Computer and Information Sciences (2013) 25, 7–24

King Saud University

Journal of King Saud University – Computer and Information Sciences

www.ksu.edu.sa
www.sciencedirect.com

ORIGINAL ARTICLE

Analysis of accounting models for the detection of duplicate requests in web services

S. Venkatesan [a,*], M.S. Saleem Basha [b], C. Chellappan [c], Anurika Vaish [a], P. Dhavachelvan [b]

[a] *Indian Institute of Information Technology, Allahabad, India*
[b] *Department of Computer Science, Pondicherry University, Pondicherry, India*
[c] *Department of Computer Science and Engineering, Anna University, Chennai, India*

Received 31 December 2011; revised 5 April 2012; accepted 15 May 2012
Available online 24 May 2012

KEYWORDS

Request history;
DoS attack prevention;
Critical infrastructure protection;
Web service security

Abstract The Denial of Service (DoS) attack is the major issue in the web service environment, especially in critical infrastructures like government websites. It is the easiest one for the attackers where they continuously generate the duplicate request with less effort to mitigate the availability of server resources to others. To detect and prevent this type of duplicate request attacks, accounting the client history (i.e., client request detail) is very important. This paper proposes a cookie based accounting model, which will record each and every client request in the cookie and the hash value of the cookie in the server database to detect the client's misbehavior like modifying the cookie information or resending (replay) the prior request cookie with the current request. Also this paper has analyzed all the accounting models including the proposed accounting model with respect to qualitative and quantitative results to prove the proposed model efficiency. The proposed model achieves more than 56% efficiency compared to the next efficient existing model.

© 2012 King Saud University. Production and hosting by Elsevier B.V. All rights reserved.

1. Introduction

A denial of Service is an attack generated by the attackers simply by sending the huge number of spam or duplicate requests to the server to make it unavailable to others or legitimate clients. The Computer Emergency Response Team (CERT-DoS 2001) classifies denial of service attacks into three broad categories. The first category aimed at the consumption of resources such as network bandwidth or CPU. The second category is the physical destruction or alteration of network components. The third category is the destruction or alternation of configuration management. Amongst the three types, the first type of attack is easy for the attackers to do without taking much effort. This attack will be either through the protocol exploitation (TCP SYN flood: sending only the TCP SYN request continuously without responding to the TCP/ACK, UDP flooding: sending large number of UDP packets to the servers port, Smurf attack: broadcasting the ping request with the source address of the victim and the reply of the ping request will reach the victim from all machines, Ping flood: sending massive number of ping requests) or through the massive number of application layer level duplicates (spam) requests. This paper is intended to concentrate on the application layer level duplicate requests.

* Corresponding author.
E-mail address: venkalt_s@yahoo.co.in (S. Venkatesan).
Peer review under responsibility of King Saud University.

1319-1578 © 2012 King Saud University. Production and hosting by Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.jksuci.2012.05.003

In application layer level spam requests, attackers may frequently send the requests (may be same page or data requests or different requests) to the server as a legitimate host but with the intention to reduce the availability of resources. Attackers may be the human or compromised machines (Zombies or Botnet). A botnet comprises thousands or compromised machines to disrupt the availability of critical resources, which will become a major issue in the web services (Alonso et al., 2004). To overcome the issue of the botnet, the concept of Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA) (Kandula et al., 2005) is used, such as the challenge response method. There are various CAPTCHA models available where the server will request the client to respond to the challenge. For example, a client has to send the textual format of the words given in the image. In web services, we cannot use this concept for all the web pages and for every request because it will increase the computational cost and will also hurt the clients. Also, some of the CAPTCHA's are recognized by the zombies or bots (Gaddam, 2008). In case the human is performing this application layer level attack, then it is not possible to detect.

The denial of service may also be handled through the various difficult level puzzles (Aura et al., 2000; Feng and Kaiser, 2010; Khor and Nakao, 2011), but it is very difficult to have that in the web services because we cannot have the puzzles for every page. It will hurt the legitimate clients and it will increase the computational cost for generating the puzzle for every request and verify it for every response.

Alfantookh (2006) developed the DoS intelligent detection system using neural networks. However to detect the attack in the web service environment the client history is required. In order to detect and prevent the duplicate request attack in the web service environment, some efficient model is needed, which requires the client request (client history) details. Hence, we need the proper client history accounting model. With this motive, this paper proposes an accounting model for web services. The first focus of this paper is to efficiently account the request history and servicing the client with less computational time. The second focus of this paper is to prevent the replay and the modification attack on the request history.

To keep the client history, a cookie is used and the credentials of the cookie will be stored in the server log. We all know that the cookie is a client side tool and has low security. However, this proposed model will increase the security level of the cookie through integrity verification. The request history of the client will be analyzed based on different criteria provided by Aura et al. (2000), Yuan and Mills (2005), and Ranjan et al. (2006) to detect the duplicate or spam requests. The request history consists of the total count of the pages visited, the count of every web page access, inter-arrival time of the request, date and time of very first request, etc.

The cookie will get updated for every request and is transferred to the client along with the response. It is obvious that the client has the open opportunity to make any modification on the cookie (request history) because it is available to client. To overcome this issue, the hash value of the cookie is stored on the server side and it is verified for every request. In case the client makes any modification on the cookie or resends the cookie of the second or third request, the client can be easily captured by the server with the help of the hash value (any tiny modification on the data will give the avalanche affect on the hash value) stored in the server log.

The experimental results given in this paper has been taken with 250 legitimate user access to prove that this proposed model is more efficient, with respect to space and computational complexity, than the existing models.

The remaining of this paper is organized as follows: Section 2 gives a brief description of the related works. Section 3 describes the proposed accounting model for detecting and preventing the spam requests. Section 4 describes the properties of the proposed model. Section 5 describes the analysis of all the accounting models and the experimental analysis of the proposed model in comparison with the existing models, including the inconsistency removal. Section 6 concludes the paper with some directions on future enhancements.

2. Related work

The Business Process Execution Language (BPEL) engine executes the specified business processes with web services by consuming reasonable memory and processing time, but when the massive number of spam requests come, then consumes more memory and CPU time (Jensen et al., 2007, 2008) and it causes a denial of service. Also, the comparison chart given by Jensen et al. (2008) shows that the prevention of this type of attack is very important compared to other DoS attacks like TCP SYN flood. Suriadi et al. (2010) also analyzed the application layer level denial of service attack impacts on web services (Web Service Definition Language (WSDL) flooding: sending massive number of WSDL requests where every WSDL request itself having a very complicated security process to secure itself from the public; heavy cryptographic process: adding multiple signature block and multiple heavy encryption blocks; and deep XML DoS attack: embedding the excessively nested XML in the message part). Suriadi et al. (2010) also shows that a lot of memory and CPU cycles are consumed for these kinds of attacks. Hence, the prevention of these application layer level DoS attacks is very much important to save the memory and CPU cycle, and therefore, protect the critical infrastructures. This is our main motivation, namely develop an accounting model, which will be necessary to detect and prevent the duplicate or spam requests attack.

To prevent the spam request DoS attack, the following different approaches for accounting the request history are given by Jensen and Schwenk (2009).

(i) **Monitoring approach**: Every request to the server must be notified to the monitor for global trace back. The serious problem with this model is performance reduction by monitoring all the requests. Also, it does not have the concept of request history, which is important to detect the attack.

(ii) **Server side logging approach**: Every request to the servers is logged in the local files. In the event of a request, the server may look up the appropriate request history in its log files to analyze it and contacts the service to fulfill the request of the customer if it is a legitimate request, otherwise the request would be dropped. The issue with this approach is storing all information in the server log. It will raise the size of the log files and the complexity of searching for a particular entry.

(iii) **Request history approach**: In order to overcome the issue of the maximum log file, a more viable approach is the service invocation history, that is, to place the complete history into the request message itself. Every service that processes a request takes the history from the incoming request message, appends its own service address, and puts the resulting history block into each outgoing request message that belongs to the same request context. This way, each request message can be examined to exactly identify all services that are previously processed. In the event of an attack, the service at the observation point can directly investigate the attack request messages in order to determine the attack at the entry point.
The drawback with this model is that there is no guarantee that the client will send the next request with the very first preceding request history. There is a possibility to send the second or third preceding request history with the current request. For example, consider that the client has sent its second request and got the response with the history up to the second request. After that, the client sends its third request with the history received for the first response. In this situation, the server cannot detect the attack either at the entry point or at the processing point.

(iv) **Extended request history approach**: In the above approach, the attacker can have the chance to insert the fake service invocation history aiming to cheat the observation point. In order to prevent this kind of fake insertion, the concept of a security token is introduced. That is, every incoming request issues a token (consists of message id and timestamp) which is encrypted with the symmetric key and embedded with the request history. For the forthcoming requests (consists of the previous history), they will analyze the request history after verifying the tokens, which will be decrypted using the symmetric key available with the observation point. Even though a security token is available, the possibility is there to add the fake history or modify the requests history. To overcome this issue, they have added the digital signature scheme to protect the integrity. The request history will be verified based on the digital signature and the token validity. Every request history is valid for a period of time which is available in the token. The request history comes after the timeout period elapsed.
The issue in this model is with the overhead related to computational time of the symmetric cryptographic operation, verifying the digital signature. Also, it has the vulnerability of the replay attack; that is, within the time period the attacker has the opportunity to perform the replay.

(v) **Context-based access control**: In this approach, the access is restricted to the persons related to their process. For example, create customer billing account is only allowed to the create customer. The problems with this model are: First, setting up the customer access control policy in the initial stage is difficult in a web service; Second, it is not applicable for the inter company service invocations. Third, the customer account persons (attackers) can perform the flooding here, which cannot be detected.

Apart from the above request history accounting models, some other models are also available to maintain confidentiality and integrity of the session cookies, which may also be helpful to detect the spam request. The main motive of all the models is to secure the cookie (session) information, not to have the proper request history accounting and preventing the replay attack. All research works discussed in this section have the perspective of accounting the history and avoiding the replay and modification attacks. Before going into that, one thing we need to emphasize that all the models are using the cookie to maintain the session histories but Sit and Fu (2001) identified the two important issues of using the cookie. The first one is that most of the websites rely on cookie expiration to automatically terminate a login session. But cookies can be modified by users to extend expiration times. The second issue is that most of the websites are using plaintext cookies without SSL protection. This may cause the eavesdropping of the cookie which will lead to leakage of the user movements. Hence, if we maintain the client request history in the cookie, then it is possible for the attackers to modify, eavesdrop or replay the request history. However, most of the following existing models had solved the above issues of using cookie through encryption and verifying the integrity.

Park and Sandhu (2000) used IP-based cookie, a hashed password-based cookie, and signature-based cookie to secure the cookie. However, this mechanism is not efficient in preventing the replay attack (Wu et al., 2010).

Fu et al. (2001) proposed the model for securing the cookie using the keyed non-malleable Message Authentication Code (MAC). Keyed non-malleable MAC is used to hash the cookie information along with time stamp and the data. This method provides the confidentiality with the keyed Hash based MACs (HMAC) and also maintains the protection of replay attack through time stamp mechanism. For a period of time, the HMAC key will be updated or new one will be generated to avoid the replay protection, but the problem is within the time period, a user can easily perform the replay attack.

Xu et al. (2002) proposed a session protection model by adopting the One-Time Pad (OTP) for credit card session protection. The credit card security is very important in the e-commerce service, which is also discussed by the Panigrahi et al. (2009) along with the solution to prevent the credit card fraud, but the cookie information needs to be protected (i.e., the cookie information related to credit card flow from the client should not be replayed). In the OTP-based model, for every new cookie or for each update of an old one, a new random key will be created and the cookie information is encrypted and the older keys are removed. Whenever the server creates the cookie or updates the cookie, it also needs to create a new random key to encrypt the information and also the random key will be stored by replacing the older one in the database with additional details like session time and the index string to identify the database. Whenever the server receives the cookie, it will decrypt the cookie using the key stored in database and allow the client request for process if the cookie is valid. At the end, again, the updated cookie will be encrypted with a new key and embedded within the response.

In this model, the client cannot change the cookie or perform replay attack because it is encrypted with one time key. Nevertheless, this solution has a major drawback, that is, generation of random numbers (OTP: where key length should be

greater than or equal to the message length) for every request. It is costly to generate OTP for every request.

Yue and Wang (2009) discussed the overload (DoS) protection in e-commerce websites. Here, the concept of two hash-tables is introduced, one of which is going to store the IP address of the premium customers and the second is to store the network ID prefixes of all the customers. Under the overload model condition, any customer coming with the IP address, which is available in the IP hash table, is allowed to access the service with full probability, otherwise they will get the low probabilities to access the services. This low probability IP will be checked with the randomized function whether to allow the request or not. Even though it seems good, it is having some issues like

- It is applicable only for e-commerce sites which have registered customers.
- No new users are allowed to access the sites during overload.
- It needs more space to maintain the data of all customers, IP and network ID always (we cannot expect or restrict the client to access our services from the dedicated IP only).

Pujolle et al. (2009) developed a security architecture by adopting the proxy in the protection scenario. The architecture is performing the encryption and hashing operations to secure the cookie in the proxy server, instead of doing that at the web server end, to avoid the unwanted bottlenecks of the server. The issue with this approach is the same as in the Xu model (2002), but at the proxy end, that is, random number generation for the one time pad (OTP). To mitigate the cost, if they are using the same key for the whole session, then it will be vulnerable to the replay attack.

Wu et al. (2010) had proposed the model for securing cookies based on MAC address encrypted key ring. In this model, the client needs to send the initial request to the server along with its public key. On the server side, the cookie will be generated for the client and it will be put into the digital envelope along with the server public key, timestamp and the server's signature of the cookie. In the next step, the server encrypts the digital envelope using the public key of client and forwards it to the client with the response.

Whenever the client gets the digital envelope, the client will decrypt the envelope using its private key. At first, the client will verify whether the signature is correct or not using the server public key. Second, the client will check whether the timestamp is legal or not. At last, the client will send the envelope consisting of the server key, cookie, timestamp, and signature with the next request to the server. After receiving the envelope, the server will verify the signature and timestamp after decrypting the envelope using its private key. If the signature and timestamp match then the cookie will be accepted, otherwise it will be rejected. This model uses the database to store the client information (like user's preferences & other information, whenever the user surfs the website).

The drawback is that the cookie is not stored in the database to verify the integrity of the cookie. This model shows the cookie and the database are updated for every request, but there is no discussion that the cookie is stored in the database. Hence, the model is analyzed in two ways:

First, in case the cookie is stored in the database: The cookie from the clients is not verified against the cookie available in the database by the server to check for a replay attack.

Second, in case the cookie is not stored in the database: Nothing is with the server to identify the replay of the cookie by the client.

This model is applicable for the prevention of man-in-the-middle attack not for the prevention of the replay attack. Also, it requires more cryptographic operations, which will consume more memory and CPU cycles.

Casado et al. (2006) introduced the concept of flow-cookie which will be handled by the third party cookie box to prevent the DoS attack. The request from the client to the server and the response from the server to the client will pass through the third party cookie box. The function of the cookie box is to identify the malicious request and block it. For this purpose, it is using the cookie, which is called as flow cookie here. After establishing the TCP connection using the flow cookie (TCP timestamp), whenever the server sends response to the client request the new flow cookie will be generated by the cookie box and it will be verified when it is echoed by the client. The flow cookie will be generated using the secret (Sr) known only to the cookie box and counter (Cr) value which will be incremented for every n seconds, source and destination port (*srcport* and *dstport*) and IP address (*srcip* and *dstip*). All these items will be hashed using a UMAC-32 or HMAC algorithm and hold the cookie as given below.

$$\text{cookie} = \text{MAC}(Sr \| Cr \| srcip \| srcport \| dstip \| dstport)$$

The maximum lifetime of the cookie is two times the increment period of Cr. The flow cookie along with the next request or acknowledgement coming after the given timeout period will be rejected and considered as malicious. Hence, the replay of the cookie, creation of a fake cookie, and modifying the cookie is not possible. However, the replay of flow cookie is possible within $2^{*}n$ seconds.

The next major drawback of this model is setting up the expiry period. In the web service, we cannot expect the next request from the client in n seconds (in case the value of the n is small) because the page may need a lot of time to fill the form or the client may be interrupted. To solve this issue, the solution given in the model is that the web server can issue an HTTP keep-alive by updating the client flow-cookies for every15 s.

Even though the cookie size is small and it will not consume significant bandwidth according to the claim of the authors, it will disturb the client machine devices like firewall, Intrusion Detection System (IDS) for frequent verification of the frequent flow cookie. Also, if all the web servers use the same concept, then the significant amount of the bandwidth will be consumed for the cookie renewal. It will also consume the execution time of the third party cookie box or web server for creating a cookie for every 15 s to massive number of clients. Also this kind of renewal of a cookie for every 15 s may be considered as the Trojan Horse by the IDS or some other detection systems because of the frequent packet transmission.

Similarly, Oppermann (2006) has proposed the cookie based model, which was also discussed by Eddy (2006) along with the other TCP SYN flood prevention models. In this model, the cookie is generated with the three following parameters.

(a) Truncated hash value of secret bit, SYN sequence number, connection Information structure like IP and Port address and index to current secret bit pool.
(b) Secret index.
(c) Encoded Maximum segment size.

To validate the cookie, the new hash value will be generated using the connection information, SYN sequence number and secret index available in the cookie along with the secret bit in the pool indexed by the index bit of the cookie. If the newly generated truncated hash value and truncated hash value in the cookie match, then the packet is accepted and will be allowed for further activities; otherwise it will be dropped. Hence, the fake cookie generation and modification is not possible. However, replay is possible if the same model applied in the web service environment because the secret bit is always the same for the same index. To overcome this issue, it is possible to have the secret bits only for a period of time then afterwards it will move onto the next location, or it will replaced with another secret value so that the replay cookie after the time elapsed will not get the equivalent hash value available in the cookie, and in this case the attack will be identified. Similar to the above third party cookie box approach (Casado et al., 2006), the expiry time based replay detection will hurt the client, because to detect the replay attack we need to set the expiry time (in case the expiry time is small, then the genuine client is also unable to send the next request in the short time due to interruption). If we set long time expiry period like five minutes, then it will give a way to the replay attack. Hence, detecting the replay of cookie using expiry period cannot give better results.

Eid and Aida (2010) proposed the model to protect the server from the flooding attack through the access nodes. When the client send the request to the server, after establishing the TCP connection, then the request will be diverted to the access nodes. Further all the requests should go via the access nodes which will do the rate limiting and access control functionalities.

The main disadvantage of this model is having a lot of access nodes and all the transmissions are redirected to the access nodes, which will consume more bandwidth. Also, there is no solution to mitigate the DoS attack. It is simply like having a proxy which will get affected by the attackers, not the real web servers. However, the request of the legitimate client will also pass through the access nodes. If the access nodes are in a bottleneck situation, then the service to the legitimate will not be delivered. Also, they have suggested to have the massive number of access nodes to solve this bottleneck problem but the trust is another problem coming out of this.

Hang and Hu (2009) proposed a flooding prevention model; their main aim is to mitigate the collision attack in the MAC. A collision attack means identifying the new message which will produce hash output exactly similar to the original message hash output. In order to mitigate this attack, the hash output bit needs to be increased. Hence, in this model, instead of having the less number of bits (24 bits) for hash algorithm output, they increased the number of bits to 32 bits. According to the cryptanalysis, the $2^{n/2}$ (n-number of hash output bits) brute force is required to get the collision. In case the output size is 24 bits, then the combination required is 2^{12} but in case of the 31 bits, it will be 2^{15}. In this way, we can mitigate the collision through which the duplicate request will be detected without any false negative.

Also this model is setting up the expiry time for the response, like the previous models, to avoid the fake or modified or replay of cookies. It uses a secret key for generating the hash value for every time period. If the response time is expired, then the secret key will be changed. This model is very well applicable for the TCP SYN flood attack, but it is not applicable for the web service because of the problem that we discussed for the previous approach (Casado et al., 2006; Oppermann, 2006), that is, the client is unable to send the response in the given timeout period. Hence, it will give the false positive. Not only that, there may be the possibility for the replay attack within the timeout period when it comes to the web service.

Alhabeeb et al. (2010a,b) proposed a model with client authentication (CA) and authenticated client communication (ACC) components to prevent flooding and malicious packet attack. According to the model, the client should first register and get approval from the CA component to contact the ACC component for service. The packets from the client should come with encrypted tag consisting of (i) ID of the first equipment which the client connected to; (ii) ID of the network service provider which owned this equipment; (iii) the IP address of the client; and (iv) Time of issuing the tag. All this information in the tag is to prevent the flooding attack in an effective manner. However, it is not suitable to avoid the replay attack.

Alhabeeb et al. (2011a,b) added a random number concept along with the four above elements in the tag. For every request, the client will be issued the random number which will come from the random number table. The random number from the table will be chosen according to the time the tag generated. The particular random number used for all tags created during the given interval. The interval provided is very small, which will allow at most two tags to be generated. Hence, this model can prevent the replay attack with the help of the random number. However, the problem with this model is holding and generating the huge number of random numbers with good length of bits to complicate the brute force attack and with expiry time (time interval). R. Baskaran et al. (2012) developed the History based Accounting and Reacting DoS Attack (HARA) model which is like the proposed model of this paper. Even though the HARA model is having few special features similar to the proposed model, it cannot efficiently handle the cookie for accounting and to detect the attack because of the improper way of handling the client request cookie. Also it does not address many issues like recovering the model from network failure, etc.

3. Proposed cookie based accounting model

To help in detecting and preventing the spam requests, we proposed the cookie based accounting model which will consume less computational time and memory space. Fig. 1 shows the detailed architecture of the proposed model for accounting the client history and detecting the attack. According to this model, every client should send the request and get the response via the entry and exit points, respectively. When the client request enters the server, the client request history integrity will be verified through the hash value verifier and the request history will be analyzed through the history analyzer (both actions are applicable only starting from the second request from the client but not for the very first request).

If the request is legitimate, the request of the client will be processed by the client request processor (may be BPEL engine: required information will be retrieved from the server database, if needed) and the respective response is prepared and sent to the request history generator. The request history generator, in turn, will generate the request history in case of the first request, otherwise it will update the request history based on the request. In addition to that, it will generate the hash value and store it in the hash value database for preced-

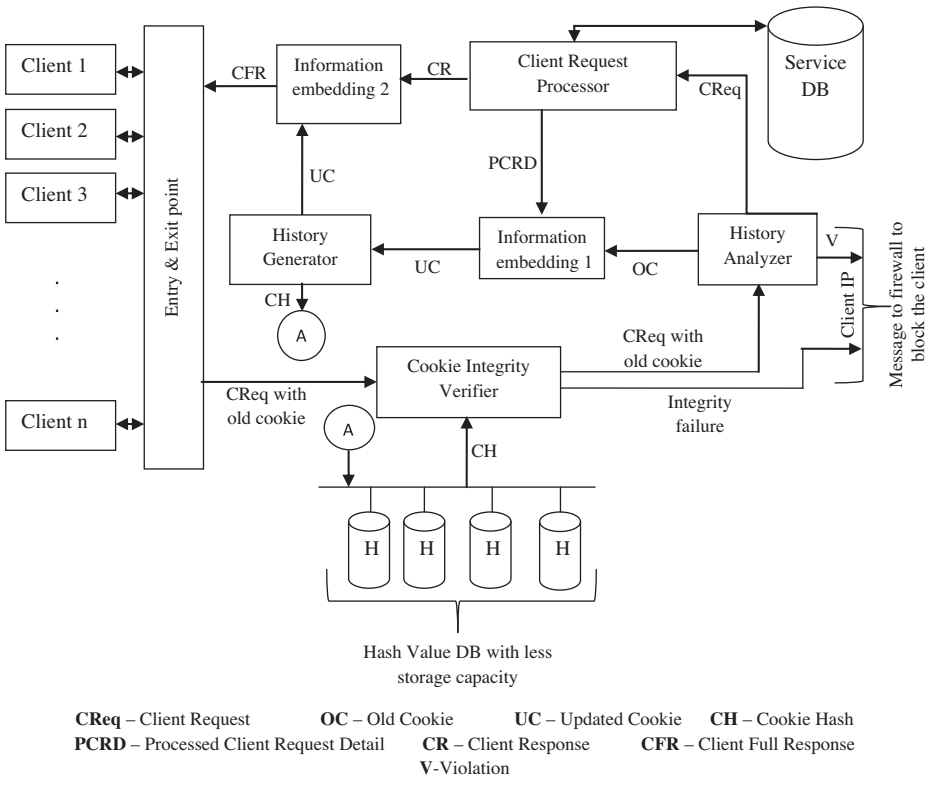

Figure 1 Cookie based architecture for accounting and processing the client history.

ing history verification. At last, it will forward the response to the client along with the cookie (Information Embedding 2). The process of the proposed model for the second and its consecutive requests are defined in Algorithm 1.

This proposed model needs two prior processes, that is, before processing the actual client requests, and one post process, that is, after processing the client requests. The prior processes are hash value verification and the request history analyzing. The post process is the history generator (and updater). The modules of the proposed model are working as follows.

- **Hash function verification**: Whenever the client request comes to the server along with the cookie (request history), this module will verify the integrity of the request history by generating the hash value for the request history in the cookie and compare it with the available hash value in the database (H) equivalent to the client IP address or identity (ID). If the hash values match, then the request will be allowed for further activities; otherwise the request will be dropped.
- **Request Analyzer**: Analyzing the request is based on the environment. In case of a payment site, the client sends the same request for a number of times, then that request should be dropped and the respective client will be informed to wait for some time before coming again. The decision depends on the administrator of the respective web service environment.

In case of a result publication server, some client may want to see the results for different registration numbers. They will start to send the request with different numbers. In case of an attacker, they will send the n number of requests for the same registration number. If n is greater than m (m is the threshold limit decided by the administrator of the server) then it will be blocked through the firewall but this scenario is very basic and old. Now, the botnet attackers are applying their intelligence to perform the advanced kind of attacks, it is very complex to detect.

Hence, the client request needs to be analyzed in different formats to detect the spam request coming like the repetition of the request, similar request inter-arrival time for all requests and pseudo random request inter-arrival time. The following four conditions can also be used to detect and prevent the spam request efficiently. The parameters for the conditions are: α is the request inter-arrival time; r is the request webpage; k is the constant value for a session; n is the maximum threshold limit of the single page access.

Algorithm 1 Request History Handling

Step 1: Arrival of second and its consecutive request with cookie.
Step 2: Retrieve the hash value (HVD) of the respective IP from the database using the indexing information (ID) available in the cookie.
Step 3: Generate the hash value (HVG) for the cookie received from client.
Step 4: Compare both HVG and HVD.
Step 5: If both hash matches then client request will be diverted to history analyzer; otherwise client request will be rejected.
Step 6: If history analyzer detects any spam then it will reject the request and block the client; otherwise transfer the request to BPEL engine for process (this step is not ellaborated in the paper because the focus is on efficient accounting of the request history).
Step 7: After client request processor (BPEL) completes the process, update the cookie with relevant information (Information embedding 1).
Step 8: Generate the hash value for the updated cookie and store it in the database by replacing the existing (previous request) hash value.
Step 9: Embed the updated cookie with the client response (Information embedding 2) and send it to client.

(i) $\alpha_5 = \alpha_4 = \alpha_3 = \alpha_2 = \alpha_1$; all the request inter-arrival time of the client is equal (given only for 5 request).
(ii) $\alpha_5 - \alpha_4 = \alpha_4 - \alpha_3 = \alpha_3 - \alpha_2 = \alpha_2 - \alpha_1$; the request inter-arrival time is increasing in same difference.
(iii) $\alpha_5 - \alpha_4 + k = \alpha_4 - \alpha_3 + k = \alpha_3 - \alpha_2 + k = \alpha_2 - \alpha_1 + k$; the request inter-arrival time is increasing in same difference.
(iv) $r_i > n$, (n value is based on the service); the specific page access requests exceeds the threshold limit; this is very important because the attacker may concentrate on the webpage which will consume more computational complexity.

Apart from the above simple conditions, the analyzing conditions given by Das et al. (2011), Yuan and Mills (2005), and Ranjan et al. (2006) will also be used to detect spam requests efficiently. The decisions with the help of the request inter-arrival time will give the false negative and false positive because the inter-arrival time may vary based on the network performance between the client and the server. To mitigate this, it is recommended to consider every request initiation time for taking the decision. However, this paper does not elaborate on the analysis part because main focus of the paper is on accounting.

- **History generator and updater**: After the BPEL engine processes the client request, this module will generate the cookie, which contains the request date, time, request page, unique identity (ID) and other information based on the requirements. Next, it will generate the hash value for the request history in the cookie and store it in the database along with the ID and IP of the client. For the second and its consecutive requests, the cookie will be updated based on the client request and the hash value will be generated for this updated cookie, and then the new hash value will be stored in the database in place of the old hash value.
- **Information fusion 1**: is to update the old cookie based on the new client request. The client request processor will process the request and give the information that it processed to the client. Based on the information, the old cookie (OC) will be updated. For example, the webpage processed for the client (PCRD) will be the input along with the old cookie, then cookie will get updated by incrementing the counter of that particular webpage and the total counter of the client access on the website.
- **Information fusion 2**: is to combine the server response and the cookie before delivering the response to the client.

4. Properties of the proposed model

Our proposed model has the following properties compared to the existing accounting models:

- **Less overhead time for searching**: The server stores the hash value of every client in the database. Hence it needs more time to search, store or update the hash value in the huge capacity database. To mitigate the time complexity in the proposed model, multiple databases are used with less storage, and also the location (database number or name which is database indexing) of the hash value is included with the request history. Whenever the client request comes to the server, the hash verifier will look for the database name or number in the cookie and it will get the hash value from that respective database instead of searching in all the databases. Also, the proposed model needs less fields (IP, Hash value and unique ID) in the database when compared to the existing models.
- **Replacement of the log details**: In order to avoid the usage of huge memory capacity database to store all the hash values, the low capacity database will be used and updated automatically that is the older log details are replaced with the new client information when database is full. It will increase the speed of the SQL query. The older information is transferred to the auditing log databases, which is the offline process used for future auditing purposes (Srivatsa et al., 2007; Jansen, 2008).
- **Replay attack prevention**: No client can send the cookie of the second or third preceding response with the current request. It can be easily identified with the help of the updated hash value available with the server for every preceding request. Only the request with the very first preceding response cookie is allowed. Hence, replay attack is not possible in the proposed model. Also, the client is not able to modify the request history (i.e., they can modify the request history but that will be easily identified with the help of the hash value available in the server log).
- **Integrating this approach in Web Service**: We all know that nowadays the web service environment uses three tiers in the server side (Presentation tier, Component tier and Back office tier). This model can be implemented in the component tier (which is responsible for processing the client request), that is, before doing the client request process, we can have the Hash verifier and History analyzer, and after the completion of client request process, we can have the History generator as given in the proposed model.
- **Cookie format**: The format of the cookie is {Unique Identity, Database Number, Page 1 Access: Number of Times, Page2 Access: Number of Times, ...}. The adversary (third party) can capture the plain cookie and may see the information of the client page access. This raises the privacy issue for the client. To overcome this issue, the server can send the cookie information over the SSL protocol channel, which will give secure transmission (to avoid eavesdropping by attackers); otherwise the request history maintained in the cookie should be in different format which is understandable only by the server not by others. Hence, the eavesdropper cannot interpret the request history.

5. Analysis of accounting models

5.1. Reliability analysis

In the related works, we have discussed the accounting models used to detect and prevent the spam request. This subsection analyzed all the models reliability efficiency with respect to the replay attack. The modification attack is not possible in all the models because the request histories are encrypted or hashed. To analyze the models, a few acronyms are used in this section:*C*-Client, *S*-Server, *R*-Request, *Re*-Response and "||" – concatenation.

5.1.1. Server Side Logging Model (Jensen and Schwenk, 2009)

- $C \text{—} R_1 \text{——} > S$ {Client History (client IP, Time and other information) will be stored in Database}
- $C < \text{——} Re_1 \text{—} S$
- $C \text{—} R_2 \text{——} > S$ {Update Client History in the Database}

- C <——Re_2—S
- C —R_3—— > S {Update Client History in the Database}
- C <——Re_3—S

In this scenario, the client history is always with the server. Hence, there is no possibility of a modification attack, a fake insertion or a replay attack.

5.1.2. Fu et al. (2001) Keyed Hash (HMAC) Model

Normal scenario

C —R_1– –> S
{1: $cookie_1$ = Client History (client IP, Time and other information), 2: Key K_1 creation, 3: Generate hash value $H_1 = MACK_1$ ($cookie_1$) and 4: store the key K_1 and client IP or ID in Database}|
C <——Re_1|| $cookie_1$||H_1—S
C —R_2|| $cookie_1$||H_1– –> S
{0: the hash value will be generated for the received $cookie_1$ using key K_1 equivalent to the client in the database and verified with the hash value H_1 received from the client. If they match then, 1: $cookie_2$ = Update $cookie_1$ based on client new information, and 2: Generate hash value $H_2 = MACK_1$($cookie_2$). If they do not match, the request will be dropped}
C <——Re_2|| $cookie_2$||H_2—S
C —R_3||$cookie_2$||H_2– –> S
{0: the hash value will be generated for the received $cookie_2$ using key K_1 equivalent to the client in the database and verified with the hash value H_2 received from the client. If they match then, 1: $cookie_3$ = Update $cookie_2$ based on client new information, and 2: Generate hash value $H_3 = MACK_1$($cookie_3$). If they do not match, the request will be dropped}
C <——Re_3| $cookie_3$|H_3—S

Attack scenario

C —R_1– –> S
{1: $cookie_1$ = Client History (client IP, Time and other information), 2: Key K_1 creation, 3: Generate hash value $H_1 = MACK_1$ ($Cookie_1$) and 4: store the key K_1 and the client IP or ID in Database}
C <——Re_1|| $cookie_1$||H_1—S
C —R_2|| $cookie_1$||H_1– –> S
{0: the hash value will be generated for the received $cookie_1$ using key K_1 equivalent to the client in the database and verified with the hash value H_1 received from the client. If they match then, 1: $cookie_2$ = Update $cookie_1$ based on client new information, and 2: Generate hash value $H_2 = MACK_1$($cookie_2$). If they do not match, the request will be dropped}
C <——Re_2|| $cookie_2$||H_2—S
C —R_3||$cookie_2$||H_2– –> S
{0: the hash value will be generated for the received $cookie_2$ using key K_1 equivalent to the client in the database and verified with the hash value H_2 received from the client. If they match then, 1: $cookie_3$ = Update $cookie_2$ based on client new information, and 2: Generate hash value $H_3 = MACK_1$($cookie_3$).If they do not match, the request will be dropped}
C <——Re_3|| $cookie_3$||H_3—S
C—R_4||***cookie$_1$***|| H_3– –> S: ***Replay Attack***
{0: the hash value will be generated for the received $cookie_1$ using key k_1 equivalent to the client in the database and verified with the hash value H_1 received from the client. If they match (here they will match) then, 1: $cookie_2$ = Update $cookie_1$ based on client new information, 2: Generate new Key K_2, 3: Generate hash value $H_2 = MACK_2$($Cookie_2$)and 4: replace the old key K_1 with new key K_2 in Database.}
C <——Re_4|| $cookie_2$||H_2—S: **Replay attack is successful**

In the above attack scenario cookie is replayed and it is not identified by the server because the key remain same for a period of time. The key will get changed after a period of time. If the key get changed then the replay attack will not be successful. Hence the client can perform the attack within the timeout period.

5.1.3. Xu et al. (2002) OTP Model

Normal scenario

C —R_1– –> S
{1: $cookie_1$ = Client History (client IP, Time and other information), 2: OTP Key K_1 creation, 3: Encrypt the cookie with key K_1, $EC_1 = EK_1$ ($cookie_1$), 4: store the key K_1, key expiration time and the client IP or ID in database and 5: create the envelope consisting of database identity (D_{id}) and the encrypted cookie $E_1 = (EC_1||D_{id})$}
C <——Re_1||E_1—S
C —R_2||E_1– –> S
{0: de-encapsulate the E_1 and get the respective client key from the database using database identifier D_{ID} and decrypt the cookie EC_1 with key K_1 available in the database and verify the history. If satisfies then, 1: $cookie_2$ = Update $cookie_1$ with new client information, 2: new OTP Key K_2 creation, 3: Encrypt the cookie with K_2, $EC_2 = EK_2$ ($cookie_2$), 4: update the database with the new key K_2 and expiration and 5: create the envelope of database identity and the encrypted cookie $E_2 = (EC_2||D_{id})$}
C <——Re_2||E_2—S
C —R_3||E_2– –> S
{0: de-encapsulate the E_2 and get the respective client key from the database using database identifier D_{ID} and decrypt the cookie EC_2 with key K_2 available in the database and verify the history. If satisfies then, 1: $cookie_3$ = Update $cookie_2$ with new client information, 2: new OTP Key K_3 creation, 3: Encrypt the cookie with K_3, $EC_3 = EK_3$ ($cookie_3$), 4: update the database with the new key K_3 and expiration and 5: create the envelope of database identity and the encrypted cookie $E_3 = (EC_3||D_{id})$}
C <——Re_3||E_3—S

Attack scenario

C —R_1—— > S
{1: $cookie_1$ = Client History (client IP, Time and other information), 2: OTP Key K_1 creation, 3: Encrypt the cookie with key K_1, $EC_1 = EK_1$ ($cookie_1$), 4: store the key K_1, key expiration time and the client IP or ID in database and 5: create the envelope consisting of database identity and the encrypted cookie $E_1 = (EC_1||D_{id})$}
C <——Re_1||E_1—S
{0: de-encapsulate the E_1 and get the respective client key from the database using database identifier D_{ID} and decrypt the cookie EC_1 with key K_1 available in the database and verify the history. If satisfies then, 1: $cookie_2$ = Update $cookie_1$ with new client information, 2: new OTP Key K_2 creation, 3: Encrypt the cookie with K_2, $EC_2 = EK_2$ ($cookie_2$), 4: update the database with the new key K_2 and expiration and 5: create the envelope of database identity and the encrypted cookie $E_2 = (EC_2|D_{id})$}
C <——Re_2 ||E_2—S
C —R_3 ||E_1—— > S : ***Replay Attack***
{0: de-encapsulate the E_1 and get the respective client key from the database using database identifier D_{ID} and decrypt the cookie EC_1 with key K_2 available in the database and verify the history. In this situation, the encryption key is different and the decryption key is different. Hence the decrypted cookie will give meaningless data to the server, which will not be understandable. Hence, the client request will be dropped.}

Since the cookie is encrypted with one time key, the replay of the cookie is not possible.

5.1.4. Alhabeeb et al. (2011a,b) Random Number

Normal scenario

C —R_1—— > S
{1: $cookie_1$ = Client History (client IP, Time and other information), 2: Get random number R_1 from the random number table, 3: Encrypt the cookie, random number and token creation time T_1 with K_1, $EC_1 = EK_1(cookie_1||R_1||T_1)$ which will be called as token, 4: Store the token creation time T_1 with the random number R_1}
C <——$Re_1||EC_1$—S
C —R_2|| EC_1—— > S
{0: decrypt the EC_1 with the key K_1, get the random number R_1 and compare that with the random number taken from the random number table based on the token creation time T_1. If both match there is no change in the cookie, then 1: $cookie_2$ = Update $cookie_1$ with new client information, 2: Get the new random number R_2 from the random number table, 3: Encrypt the cookie, random number and token creation time T_2 with K_1, $EC_2 = EK_1(cookie_2||R_2||T_2)$, 4: Store the token creation time T_2 with the random number R_2}
C <——$Re_2||EC_2$—S
C —R_3|| EC_2—— > S
{0: decrypt the EC_2 with the key K_1, get the random number R_2 and compare that with the random number taken from the random number table based on the token creation time T_2. If both match then 1: $cookie_3$ = Update $cookie_2$ with new client information, 2:Get the new random number R_3 from the random number table, 3: Encrypt the cookie, random number and token creation time T_3 with K_1, $EC_3 = EK_1(cookie_3||R_3||T_3)$, 4: Store the token creation time T_3 with the random number R_3}
C <——$Re||EC_3$—S

Attack scenario

C —R_1—— > S
{1: $cookie_1$ = Client History (client IP, Time and other information), 2: Get random number R_1 from the random number table, 3: Encrypt the cookie, random number and token creation time T_1 with K_1, $EC_1 = EK_1(cookie_1||R_1||T_1)$ which will be called as token, 4: Store the token creation time T_1 with the random number R_1}
C <——$Re_1||EC_1$—S
C —R_2 ||EC_1—— > S
{0: decrypt the EC_1 with the key K_1, get the random number R_1 and compare that with the random number taken from the random number table based on the token creation time T_1. If both match then 1: $cookie_2$ = Update $cookie_1$ with new client information, 2: Get the new random number R_2 from the random number table, 3: Encrypt the cookie, random number and token creation time T_2 with K_1, $EC_2 = EK_1(cookie_2||R_2||T_2)$, 4: Store the token creation time T_2 with the random number R_2}
C <——$Re_2||EC_2$—S
C —$\boldsymbol{R_3||EC_1}$—— > S :***Replay Attack***
{0: decrypt the EC_1 with the key K_1, get the random number R_1 and compare that with the random number taken from the random number table based on the token creation time. In this situation, if the random number R_1 is allotted to any other token creation time then the replay attack will be identified. If it is not assigned to another token creation time and still with the old token creation time then the replay attack will be successful}

The model is fool proof against the replay in case the random number allotted to other token or random number removed from the random number table after a period of time; otherwise the model is vulnerable.

5.1.5. Jensen and Schwenk (2009) Extended history approach with digital signature

Normal scenario

C —R_1—— > S
{1: $cookie_1$ = Client History (client IP, Time and other information), 2: Generate unique token T consists of message ID and timestamp for the cookie 3: Create the digital signature for the token and cookie using the key generated for the client $DS_1 = EK_1$ (H ($cookie_1||T$)) and encrypt the token and cookie $EC_1 = EK_1$ ($cookie_1||T$), 4: Store the key in the database along with the client IP or ID}
C <——$Re_1||EC_1||DC_1$—S
C —$R_2||EC_1||DC_1$—— > S
{0: get the decryption key K_1 from the database using the client IP and decrypt the EC_1, get the token and cookie. Also decrypt and get the hash value from DC_1. 2. Verify the digital signature by creating the hash value for the decrypted token and cookie and compare with the hash value in the DC_1. If both match then 1: $cookie_2$ = Update $cookie_1$ with new client information and update token, 3: Encrypt the cookie and token T with K_1, $DS_2 = EK_1$ (H ($cookie_2||T$)) and encrypt the token and cookie $EC_2 = EK_1$ ($cookie_2||T$)}
C <——$Re_2||EC_2||DC_2$—S
C —$R_3||EC_2||DC_2$—— > S
{0: get the decryption key K_1 from the database using the client IP and decrypt the EC_2, get the token and cookie. Also decrypt and get the hash value from DC_2. 2. Verify the digital signature by creating the hash value for the decrypted token and cookie and compare with the hash value in the DC_2. If both match then 1: $cookie_3$ = Update $cookie_2$ with new client information and update token, 3: Encrypt the cookie and token T with K_1, $DS_3 = EK_1$ (H ($cookie_3||T$)) and encrypt the token and cookie $EC_3 = EK_1$ ($cookie_3||T$)}
C <—— $Re_3||EC_3||DC_3$—S

Attack Scenario

C —R_1—— > S
{1: $cookie_1$ = Client History (client IP, Time and other information), 2: Generate unique token T consists of message ID and timestamp for the cookie 3: Create the digital signature for the token and cookie using the key generated for the client $DS_1 = EK_1$ (H ($cookie_1||T$)) and encrypt the token and cookie $EC_1 = EK_1$ ($cookie_1||T$), 4: Store the key in the database along with the client IP or ID}
C <——$Re_1||EC_1||DC_1$—S
C —$R_2||EC_1||DC_1$—— > S
{0: get the decryption key K_1 from the database using the client IP and decrypt the EC_1, get the token and cookie. Also decrypt and get the hash value from DC_1. 2. Verify the digital signature by creating the hash value for the decrypted token and cookie and compare with the hash value in the DC_1. If both match then 1: $cookie_2$ = Update $cookie_1$ with new client information and update token, 3: Encrypt the cookie and token T with K_1, $DS_2 = EK_1$ (H ($cookie_2||T$)) and encrypt the token and cookie $EC_2 = EK_1$ ($cookie_2||T$)}
C <——$Re_2||EC_2||DC_2$—S
C —$R_3||EC_1||DC_1$—— > S: ***Replay attack (Assumption: the time period is not exceeded)***
{0: get the decryption key K_1 from the database using the client IP and decrypt the EC_1, get the token and cookie. Also decrypt and get the hash value from DC_1. 2. Verify the digital signature by creating the hash value for the decrypted token and cookie and compare with the hash value in the DC_1. If both match then 1: $cookie_2$ = Update $cookie_1$ with new client information and update token, 3: Encrypt the cookie and token T with K_1, $DS_2 = EK_1$ (H ($cookie_2||T$)) and encrypt the token and cookie $EC_2 = EK_1$ ($cookie_2||T$)}
$EC_2 = EK_1$ ($Cookie_2||T$)}
C <——$Re_3||EC_2||DS_2$—S: **Replay attack is successful**

This model is unable to prevent the replay attack if the attack comes within the time period; otherwise a replay attack is not possible.

5.1.6. Proposed Model

Normal scenario

C —R_1— > S
{1: $cookie_1$ = Client History (client IP, Time and other information), 2: Generate the hash value H_1 = Hash ($cookie_1$) and store in the database along with the client *IP* or *ID*}
C <——$Re_1||cookie_1$—S
C —$R_2||cookie_1$—— > S
{0: get the cookie and generate the hash value RH_1 = Hash ($cookie_1$) and fetch the hash value H_1 from the database. If both hash values H_1 and RH_1 match then there is no change in the cookie then 1: $cookie_2$ = Update $cookie_1$ with new client information, 2: Generate the hash value H_2 = Hash($cookie_2$) and replace the old hash value H_1 with new hash value H_2 in the database}
C <——$Re_2||cookie_2$—S
C —$R_3||cookie_2$—— > S
{0: get the cookie and generate the hash value RH_2 = Hash ($cookie_2$) and fetch the hash value H_2 from the database. If both hash values H_2 and RH_2 match then there is no change in the cookie then 1: $cookie_3$ = Update $cookie_2$ with new client information, 2: Generate the hash value H_3 = Hash($cookie_3$) and replace the old hash value H_2 with new hash value H_3 in the database}
C < ——$Re_3||cookie_3$—S

C —R_1-— > S
{1: $cookie_1$ = Client History (client IP, Time and other information), 2: Generate the hash value H_1 = Hash ($cookie_1$) and store in the database along with the client *IP* or *ID*}
C <——$Re1||cookie_1$—S
C —$R_2||cookie_1$—— > S
{0: get the cookie and generate the hash value RH_1 = Hash ($cookie_1$) and fetch the hash value H_1 from the database. If both hash values H_1 and RH_1 match then there is no change in the cookie then 1: $cookie_2$ = Update $cookie_1$ with new client information, 2: Generate the hash value H_2 = Hash($cookie_2$) and replace the old hash value H_1 with new hash value H_2 in the database}
C <——$Re_2||cookie_2$—S
C —$\boldsymbol{R_3||cookie_1}$—— > S : ***Replay Attack***
{0: get the cookie and generate the hash value RH_1 = Hash ($cookie_1$) and fetch the hash value H_2 from the database. Compare both the hash values. Here, they will not match so drop the client. The further action against the client is with the server administrator}

The replay of the cookie is not possible in the proposed model. Hence, the model is foolproof against the replay attack. The interpretation of the above analysis is that the proposed model, OTP model, Random number based model, and server side logging model are reliable with respect to the replay attack. The other models are vulnerable to the replay attack. Moreover, the models based on OTP and Random number are same. Hence, we consider only the OTP based model for the further analysis in the paper.

5.2. Involved parameter analysis

This subsection compares all the accounting models, including the proposed model, with respect to the parameters. The models like Hash table (Yue and Wang, 2009) and proxy (Pujolle et al., 2009) are not necessary to compare with. The reason is that the proxy model is doing the same thing as the OTP or HMAC model (Yue and Wang, 2009). The second model is the two hash table model which is not applicable for all the environments as discussed in Section 2.

Table 1 shows the comparison of all the accounting models with different usage parameters (Key usage, need of database storage, hash value generation, use of one-time key, use of cookie, need of encryption, identity of record storage in database, overhead and foolproof). Out of the five models, two models are not foolproof (the reason is described in Section 5.1). Also, these two models need additional processing time for hashing, encryption, key generation and space for storage of keys compared to the other models given in Table 1. Hence, it is not efficient to use these two models. The remaining three models are foolproof and, therefore, we considered these three models for experimental result analysis.

5.3. Experimental result analysis

It is obvious that all spam request detection accounting models require the additional computational time and memory than the normal scenario. The additional computational time required for log information retrieval, storage, encryption, decryption, hash value generation, and so on. Similarly, the proposed model also needs additional computational time for hash value generation, insertion, updating and retrieval of the hash value into and from the database.

The computational time complexity and the memory (space) complexity of the three models (server side logging, OTP based model and proposed model) are analyzed with the simple website created by us. It is developed with the PHP (Server side language), HTML, JavaScript (Client side language), MySQL (database) and Apache (Web server) with the server system configuration of 3.2 GHz processor and 2 GB RAM.

The website consists of six pages: (1) Home cum Login page, (2) Login Verification page, (3) Content Upload page, (4) Content Insertion Page, (5) Content reading instruction page and (6) Secure Content viewing page.

The information (request history) available in the cookie for the proposed model include: all webpage access count (overall page request count), individual page request count, and the initial request time. The database record consists of the hash value of the request history (cookie), IP address and the unique ID. The database table format of the proposed model is shown in Fig. 2.

The information available in the database for server side logging are webpage request total count, individual page request count, initial request time, IP address, and the unique ID. The database table format of the existing server side logging model is shown in Figs. 3 and 4 shows the database table format of the OTP model. The experiment was conducted with 250 legitimate users to prove the efficiency of the proposed model. Fig. 5 shows the extracted experimental database information of all the three models.

The main intention of the experiment is to show how much processing time and memory required by the three models to prove their efficiency. Also, the normal scenario is used for the comparison with other models, only with respect to time complexity not for memory because it does not need any additional memory. Normal scenario means there is no mechanism

Table 1 Comparison between the accounting models.

Accounting models	Key usage	Database storage	Hashing	One-time key	Cookie	Encryption	Location identity
Server side logging approach	No	Yes	No	No	No	No	No
Extended request history model with digital signature	Yes	Yes	Yes	No	Yes	Yes	No
Use of one-time Pad model	Yes	Yes	No	Yes	Yes	Yes	Yes
Use of HMAC model	Yes	Yes	Yes	Yes	Yes	No	No
Proposed model	No	Yes	Yes	No	Yes	No	Yes

Accounting Models	Overhead	Foolproof
Server side logging approach	Needs more storage and request processing time	Yes
Extended request history with digital signature	Needs more processing time for encryption and hashing. Also needs storage space for Key	No
Use of one-time pad model	More cost required to generate massive keys and needs storage space for Key	Yes
Use of HMAC model	More cost required to generate massive keys and needs storage space for Key	No
Proposed cookie based hashing	Needs more storage and request processing time	Yes

Table 2 Processing Time Difference between the existing and proposed cookie based model (time in microseconds).

Number of users	Average processing time for server side logging (A)	Average processing time of the proposed model (B)	Difference between (A) and (B)
10 Users	0.040555	0.038748	1.81E-03
20 Users	0.051719	0.043091	8.63E-03
50 Users	0.059521	0.048759	1.08E-02
80 Users	0.061105	0.046572	1.45E-02
100 Users	0.06957	0.054494	1.51E-02
250 Users	0.173929	0.136235	3.77E-02

ID	IP	Hash Value

Figure 2 Database format of proposed cookie based model.

ID	IP	Initial Request Time	One-Time Key

Figure 4 Database format of OTP model.

adopted to handle the request history to detect the duplicate requests (i.e., website without any protection mechanism).

The first experimental analysis is with the memory requirements of the models. Fig. 6 shows the memory requirements of the three models for the different set of users (from 10 to 250). The illustration of Fig. 6 clearly states that the existing server side logging model requires more memory when compared to the proposed model. The existing OTP model consumes less memory compared to the proposed model. However, the key size taken for implementation of the OTP model is 40 bits. If the key size goes high like 128 bits or 256 bits for strong encryption of the cookie (which is very much required to increase the iteration of the brute force attack) or the size of the cookie goes high (OTP key should be equal to or greater than the cookie size) then it requires more memory than the memory consumed by the proposed model. Hence, the proposed model will be efficient in terms of memory complexity in case the key size is high in OTP model; otherwise the proposed model needs more memory than the OTP model.

The second analysis is with the time complexity. In a normal web service, whenever client sends request for a page, the process execution engine will process it and send the response. To detect the spam request, the existing and proposed model needs some additional processing than the normal website process. This will increase the processing time complexity of the process execution engine. For example, the proposed model includes the hash value generation time, history analyzing time (simple analysis: that is how many times the singe page has been accessed. If it is greater than the threshold limit five

ID	IP	Total page access count	Initial Request Time	Page1 (P1) – access count	P2 – count	P3 – count	P4 – count	.....	Pn –count

Figure 3 Database format of existing server side logging model.

***OTP Model information in DB*:** "1";"127.0.0.1";"57";"3";"sssssssss"

OTP Model fields in DB: "ID";"IP";"Min";"Sec";"One-Time Key"

***Server side Logging Model information in DB*:** "1";"127.0.0.1";"1";"0";"0";"0";"0";"0";"1";"22";"44"

Server side Logging Model fields in DB: "ID";"IP";"P_1count";" P_2count ";" P_3count ";" P_4count ";" P_5count ";" P_6count ";"total page access count";"Min";"Sec"

***Proposed Model information in DB*:** "1";"172.0.0.1";"1f9e4d0aaf625b4173e1b858ad80a9f1"

Proposed Model fields in DB: "ID";"IP";"Hash Value"

Figure 5 Exported data from the database for all models.

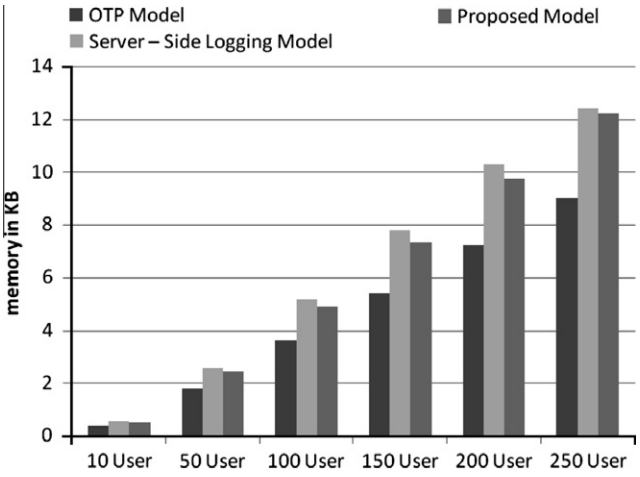

Figure 6 Experimental results with respect to memory.

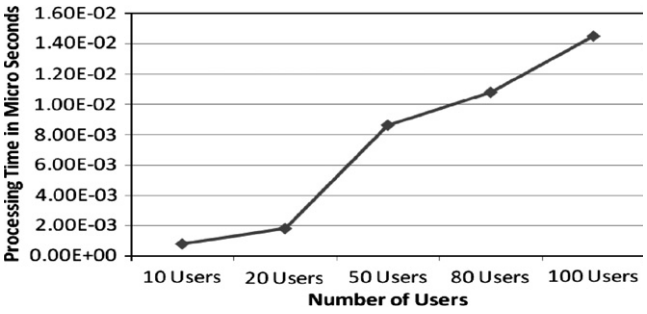

Figure 8 Processing time difference between the server side logging and proposed model.

then the client will be blocked), hash value mapping, storing, updating, retrieval time from the database and cookie update time, in addition to the normal request processing time. Similarly, the existing server side logging model includes the history information storing, updating, retrieval time from the database and history analyzing time (same as the proposed model), in addition to the request processing time. This is similar for the OTP model. The experimental results of all the accounting models and normal scenario for accessing the six pages of the website by the set of 10–250 users are shown in Fig. 7. The number of records in the database is equal to the number of users. The experimental output of every user includes the total six webpage access time.

For the OTP model, our implementation concept is different from the actual concept given by the developers. In the actual model, the cookie will be encrypted through one time key and then sent to the client with the response but we have implemented a similar approach, namely after storing the cookie in the database, the cookie is encrypted and sent to the client with the response.

In the next step of the actual implementation, whenever the client send the encrypted cookie embedded in the request, then the cookie will be decrypted and verified for violation. If there is no violation, then the request will be processed, new key will be generated, cookie will be updated, updated cookie will be encrypted with the new key, the new key will be stored in

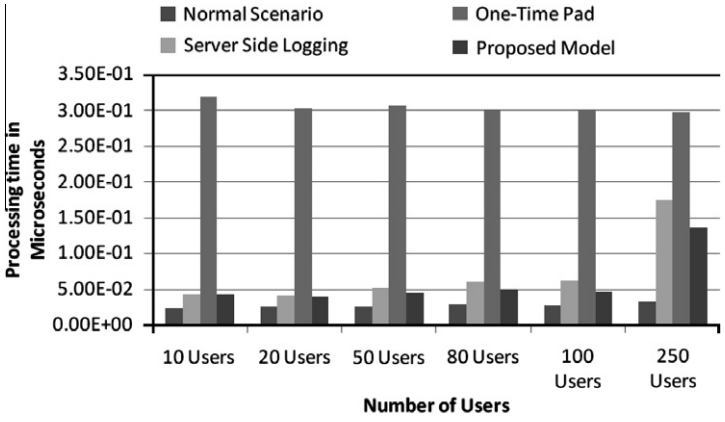

Figure 7 Average computational time comparison.

Table 3 Actual experimental results of 100 users

Average users	Processing time of OTP based model (xo)	Processing time of the server – side logging (xs)	Processing time of the proposed model (xp)
1st 10 users	**0.317655**	**0.038802**	**0.035963**
2nd 10 users	**0.320129**	**0.147793**	0.073363
3rd 10 users	0.284886	0.066969	**0.149392**
4th 10 users	0.298514	0.06302	**0.14318**
5th 10 users	0.286461	0.073168	0.056617
6th 10 users	0.286909	**0.049881**	0.041531
7th 10 users	**0.318381**	0.079164	0.046918
8th 10 users	0.303875	0.092625	0.037471
9th 10 users	**0.278412**	**0.042267**	0.038168
10th 10 users	0.296241	0.092382	**0.169679**

the database and the response along with the new encrypted cookie will be sent to the client.

In our implementation, whenever the client sends the encrypted cookie to the server, it will be decrypted with the key available in the database. Next, the history information will be retrieved from the database and compared with the decrypted request history information (cookie) to verify the integrity of request history. If they match, then the request will be processed, new key will be generated, cookie will be updated, updated cookie will be encrypted with the new key, the new key and the updated cookie will be stored in the database and the response along with the new encrypted cookie will be sent to the client. The modified implementation of this model causes more computational time consumption when compared to the other models. However, this model has the greater drawback which are the one time key for every request and the OTP key should be equal to or greater than the cookie size.

Even if we have implemented the OTP model as it was given by the developer, the processing time consumption would be more or equal to the proposed model, because the number of fields in the database is one more than the proposed model. The reason for implementing it in our way is to check its reliability. That is, just by seeing the decrypted value; we cannot determine the exact value. For example, 0 may change to 1. It will not be determined that this value is modified or not, because we do not have the original value to check the integrity. However, we may say one thing that the decrypted cookie (modified cookie) will not get the cookie format so that we can identify the integrity violation.

Fig. 7 shows that the proposed model always takes less time compared to the existing models for all users. Also, it is obvious that the normal scenario (without any protection mechanism) always takes less processing time compared to all the models because there is no additional computation. Table 2 and Fig. 8 clearly show the increase of time differences between the proposed model and existing server side logging model when the number of users increases. The reason for the increase in the processing time is the number of user entries in the database. Hence, if the number of users increases then our proposed model can save processing time compared to the existing models.

5.4. Inconsistency analysis

The results taken from the real time experiments cannot give the normalized results for every access. Few accesses may take more time/less time compared to other accesses because of the CPU load. The same thing happened in our above experimental results. In this situation, we took the previous request access time and used in that place. However, it is not good to give the comparison between the models by this approach. Hence, we used the standard deviation to identify and remove the access which is having the inconsistent processing time.

We have taken only the first hundred users access for the analysis. All the hundred users access are segregated into ten for all models as shown in Table 3. The reason for taking the set of 10 users instead of the individual user is to give the clear representation.

For the value given in Table 3 for server side logging model (xs), we have applied the Standard Deviation (SD) function to identify the upper limit and the lower limit of the access time to avoid the inconsistent output.

$$\text{mean of server side logging}\,(\mu) = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{\sum_{i=1}^{10} xs_i}{10} = 0.074607 \text{micro seconds}\,(ms)$$

$$\text{variance of server side logging}\,(\sigma^2) = \frac{\sum_{i=1}^{n} (x_i - \mu)^2}{n} = \frac{\sum_{i=1}^{10} (xs_i - 0.074607)^2}{10} = 0.00091547ms$$

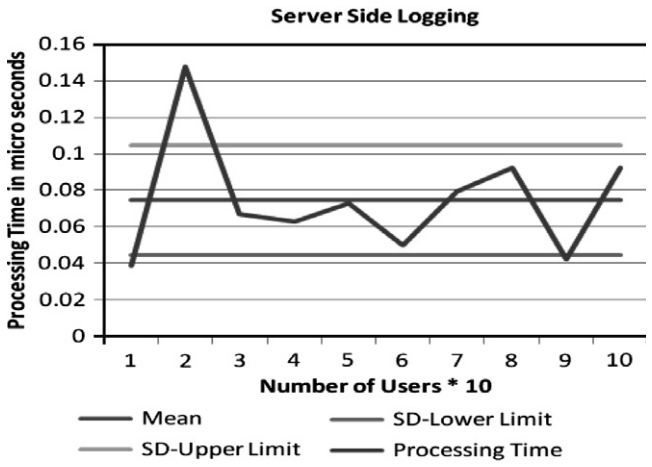

Figure 9 Inconsistency representation of server side logging model.

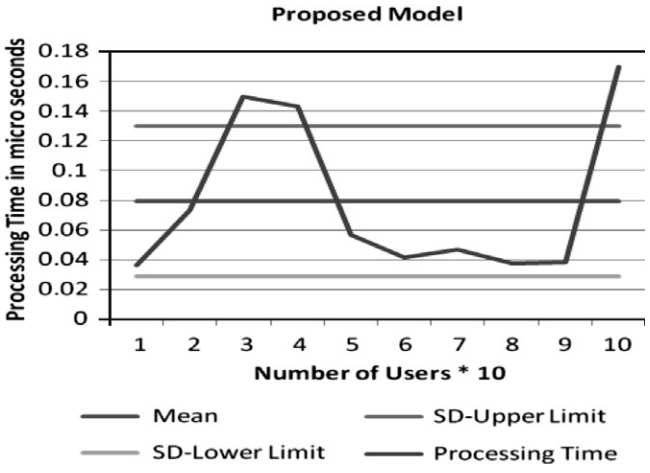

Figure 10 Inconsistency representation of proposed model.

$$Standard\,Deviation\,for\,server\,side\,logging(\sigma) = \sqrt{\sigma^2}$$
$$= \sqrt{0.000915147} = 0.030251\,ms$$

After finding the SD value, Fig. 9 has been generated to identify the inconsistent values. Fig. 9 shows that the inconsistent values are the 1st, 2nd and 9th set of users which is shown in bold in column three (xs) of Table 3.

The same SD function is used for the proposed model (xp) to identify the lower and upper limit of the access time as shown below.

$$mean\,of\,proposed\,model(\mu) = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{\sum_{i=1}^{10} xp_i}{10}$$
$$= 0.07928\,micro\,seconds\,(ms)$$

$$variance\,of\,proposed\,model(\sigma^2) = \frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}$$
$$= \frac{\sum_{i=1}^{10}(xp_i - 0.079228)^2}{10}$$
$$= 0.002550631ms$$

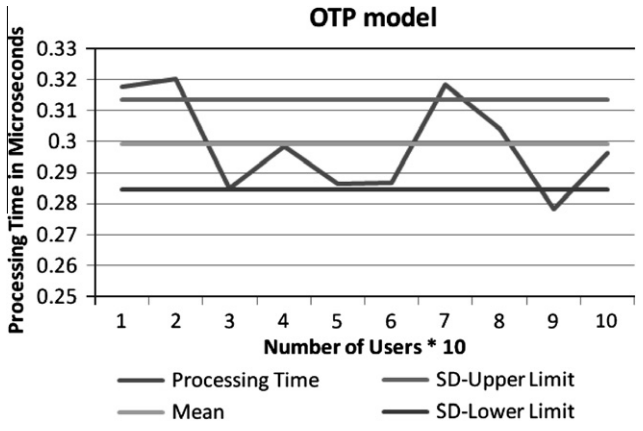

Figure 11 Inconsistency representation of OTP model.

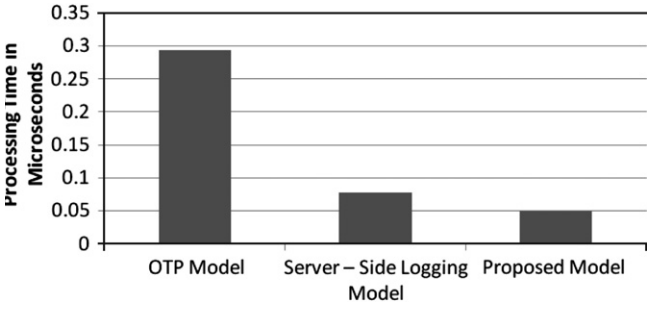

Figure 12 Server side logging model vs cookies based model.

$$Standard\,Derivation\,for\,proposed\,model(\sigma) = \sqrt{\sigma^2}$$
$$= \sqrt{0.002550631} = 0.050504\,ms$$

After finding the SD value, Fig. 10 has been generated to identify the inconsistent values. Fig. 10 shows, that the improper values are the 3rd, 4th and 10th set of users which is shown in bold in column four (xp) of Table 3.

Similarly, the SD function was used for the OTP model (xo) to identify the lower and upper limit of the access time.

$$mean\,of\,OTP\,based(\mu) = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{\sum_{i=1}^{10} xo_i}{10}$$
$$= 0.299146 micro\,seconds\,(ms)$$

$$variance\,of\,OTP\,based(\sigma^2) = \frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}$$
$$= \frac{\sum_{i=1}^{10}(xo_i - 0.299146)^2}{10}$$
$$= 0.000221796ms$$

$$Standard\ Derivation\ for\ OTP\ (\sigma) = \sqrt{\sigma^2} = \sqrt{0.000212796}$$
$$= 0.014588\,ms$$

After finding the SD value, Fig. 11 has been generated to identify the inconsistent values. Fig. 11 shows, that the improper values are the 1st, 2nd, 7th and 9th set of users which is shown in bold in column two (xo) of Table 3.

After obtaining these mathematical results, we have identified the inconsistent processing time caused by the CPU and it

Table 4 Consistency vs inconsistency.

	OTP model	Server side logging model	Proposed cookies based model
Average output of 100 users (inconsistency)	0.299146	0.074607	0.079228
Average output of 100 users after removing the inconsistent results (consistency)	0.292814	0.073887	0.047147

was removed. But the OTP has four improper values whereas the other two models have only three improper values. To bring the proper comparison, one more value needs to be taken from the other models. Hence we removed the value which is very near to the standard deviation upper and lower limits. The values in gray color in Table 3 are very near to the upper or lower limit.

Now, the four set of users (inconsistent set) in all the models are removed and compared with one another as shown in Fig. 12, which shows that the proposed model consumes less time when compared to the existing models.

Table 4 shows that the actual results taken from the experiment for average output of 100 users. It shows that the proposed model consumes more time when compared to the existing server side logging model. After removing the inconsistency results (through standard deviation) which was caused due to the CPU load, it shows that the proposed model takes less processing time than the existing server side logging model. Hence, the request processing time of the proposed model is more than the existing server side logging model means due to the inconsistency only, not because of the model. Hence, the proposed model is efficient to handle the multi-millions of users request history in the web service environment with respect to time complexity.

The above analysis among the models shows that the proposed model is efficient for accounting the request history of the clients to detect the spam request attack.

5.5. Network connection fault tolerant analysis

Earlier, Bernstein (1996) and Bona (1996) proposed the cookie based model to prevent the TCP SYN flood attack. When client sends a SYN request, the server calculates a one-way hash of the sender's sequence number, ports, the server's secret key, and a counter that changes every minute and sends that to the client along with the TCP/ACK response without allotting the space in the Transmission Control Block (TCB). Nothing is stored in the server side. When the client replies with an ACK packet, the server recalculates the same hash function with the help of the secret key which is denoted by the counter value. If the recalculated hash value and hash value that comes with the ACK do not match then the ACK packet will be dropped; otherwise the request will be accepted for further activities. The disadvantage of this model is, when the TCP/ACK packet is lost, the server is prevented from resending SYN + ACK packets, because there is no information regarding the client (Ricciulli et al., 1999). So, the cookie based approaches are not that much evolved in the TCP SYN flooding design.

If we bring the same protection model into the web service denial of service attack then this problem will be solved by resending the request. For example, the client sent a request to the server for a service and the server processed the request and responded to the client, but it was not received by the client due to the network problem. In this case, the client may resend the request to the server for the same service. According to the TCB, if the same request comes again then it will be considered as new, but in web services it will not be considered as new because the request will come with the old cookie. In case the client sent the first request and not received the response with the cookie, the client needs to resend the request and it will be considered as new client and have to pass through some puzzle test again, which is inconvenient to the customer. To avoid this issue, we may maintain the cookie information in the client side for the first request; after that it will be removed for the future requests. In case of the second and its consecutive request, even though the client did not receive the response, they may resend the earlier request and receive the equivalent response.

As per the proposed model, the resend or replay is not possible so it is very difficult to handle the above legitimate replay request situation. The following scenario clearly illustrate this false positive issue.

C —R_1– –> S
1. Create cookie: c_1
2. Create hash value $H_1 = H(c_1)$.
3. Store in database

C —$R_2||c_1$– –> S
1. Create hash value for the cookie received $RH_1 = H(c_1)$.
2. Retrieve equivalent hash value of the cookie from database, that is, the hash value H_1
3. Compare H_1 and RH_1. If there is no match drop the request; otherwise follow the remaining steps.
4. Update cookie: c_2 = update (c_1, R_2).
5. Create hash value $H_2 = H(c_2)$.
6. Store the new hash value in the database by replacing the old one.

C < –× --$Re_2||c_2$—S: **The response failed to reach the client**

C —$R_2||c_1$– –> S
1. Create hash value for the cookie received $RH_1 = H(c_1)$
2. Retrieve equivalent hash value of the cookie from database that is the hash value H_2.
3. Compare H_2 and RH_1. There is no match, so connection dropped.

To solve this problem, the only solution is having the last two credentials in the database (Xu et al., 2002). For example consider the following scenario where the genuine client is continuing the dialogue with the server.

C —R_1– –> S
1. Create cookie: c_1
2. Create hash value $H_1 = H(c_1)$
3. Store in database (that is in *1st* credential column)

C < ——$Re_1||c_1$–S
C —$R_2||c_1$– –> S
1. Create hash value for the cookie received $RH_1 = H(c_1)$
2. Retrieve equivalent hash value of the cookie from database, that is, the hash value H_1.
3. Compare H_1 and RH_1. If there is no match drop the request; otherwise follow the remaining steps
4. Update cookie: c_2 = update(c_1, R_2)
5. Create hash value $H_2 = H(c_2)$
6. Store in the another column (that is in the *2nd* credential column) of the database

C < –×–$Re_2||c_2$—S: The response failed to reach the client
C —$R_2||c_1$– –> S
1. Create hash value for the cookie received $RH_1 = H(c_1)$.
2. Retrieve equivalent hash value of the cookie from database, that is, the hash value H_2.
3. Compare H_2 and RH_1. There is no match, so follow the remaining step.
4. Retrieve second equivalent hash value of the cookie from database, that is, the hash value H_1.
5. Compare H_1 and RH_1. If there is no match drop the packet; otherwise follow the remaining steps.
6. Update cookie: c_2 = update(c_1, R_2)
7. Create hash value $H_2 = H(c_2)$
8. Replace the last entry hash value with the new cookie hash value in the database(that is in the 2nd credential column)

C < ——$Re_2||c_2$–S
C —$R_3||c_2$– –> S
1. Create hash value for the cookie received $RH_2 = H(c_2)$.
2. Retrieve equivalent hash value of the cookie from database that is the hash value H_2.
3. Compare H_2 and RH_2. If there is a match follow the remaining step.
4. Update cookie: c_3 = update(c_2, R_3)
5. Create hash value $H_3 = H(c_3)$
6. Replace the credential column with the new cookie hash value in the database(that is *1st* credential column)

The same with the view of the following basic set theory is the intersection function f with inputs set s_1 and s_2 provides the null output if there is no match with any of the two hash values and it will be dropped; otherwise the valid cookie output will be there.

$$f(s_1, s_2) = s_1 \cap s_2 \quad \text{where} \quad s_1 = \{H_1, H_2\} \quad \text{and} \quad s_2 = \{RH\}$$

$$f(s_1, s_2) = \{ = \emptyset, \text{packet failed to prove its identity} = H_1 \text{ or } H_2 \text{packet proved its identity}$$

Even though it will consume additional memory, it is vital to avoid network connection problems. With respect to the processing time, additional time is consumed only at the time of replay; otherwise there is no additional processing time. In view of verifying the cookie with the stored credential, the latest one should be verified first to avoid the additional processing time if there is no replay attack. In case the attacker starts to send the replay intentionally, it will affect the process power so we need to set the threshold limit to send the replay of very first preceding request not more than that. This network problem issue is not only applicable for the proposed model and the model proposed by Xu et al. (2002); it is applicable for all the models.

The next important thing to consider is setting the expiry time which is discussed in the Section 2 for the existing models. There, we claimed that the expiry time hurts the client and also gives false positives. If we will not give expiry time for the client to send a request or response then the web service will be vulnerable to the attack, even though we are proposing or creating the very effective security models. Another important consideration is on setting the timeout period; for example if we set the expiry period five minutes, then the replay attack is very well achieved and if the expiry time is 30 s then the client will suffer from the false positive. All the existing models are identifying the replay through the expiry time so that the expiry time is very crucial for all the existing models but for the proposed model, the replay attack, the fake cookie, or the modified cookie detection is not dependent on the expiry time. Hence, setting up a reasonable expiry time will not affect our model robustness and legitimate clients will be serviced properly.

5.6. Analysis of the cookie modification

In our model, to protect the cookie from replay attack, fake creation attack, and modification attack the hash code is used. As we know, the hash is vulnerable to the collision attack (two inputs may produce the same hash output), i.e., to get the new input producing the hash output equivalent to the original input we need $2^{n/2}$ (n is the number of output bits) brute force. In order to achieve the memory complexity, we suggested the minimum output bit hash algorithm but it is vulnerable to the collision attack. However, to perform the collision attack, the attacker has to achieve the next two things according to our model.

a. The newly created or modified cookie, which gives hash output similar to the original cookie, should have the IP address remaining constant in the cookie field because it is available in the server side to verify.
b. There is a server accepted format for the cookie in Fig. 4. If the cookie is not in the similar format then the request will not be allowed to enter the processing environment. Hence, the collided input should also have the similar cookie format.

Hence, it is complex for the attackers to bring out the cookie in the similar format and the particular part of the content constant in the $2^{n/2}$ brute force. Thus, it needs more brute force to achieve the collision.

6. Conclusion and future work

The application layer denial of service (spam or duplicate request) is a smart attack to minimize the availability of the resources without any risk. To help in detecting and preventing this application layer duplicate or spam request based denial of service attack, this paper proposed a cookie based hashing accounting model with special properties given in Section 4. After analyzing all the existing and proposed accounting models, the three (OTP model, server side logging and proposed cookie based hashing model) models are identified as fool

proof. Later, these three models' experimental results are analyzed with respect to memory (space) and time complexity. The analysis output has shown that the server side logging model consumes more time and more memory compared to the proposed model. The OTP models consumes less memory and more computational time when compare to the proposed model. In case, the size of the OTP key is increased then the memory complexity of the OTP model will also be more than the proposed model. Also, OTP incurs extra cost to generate a new key for every request. Hence, the proposed model is more than fifty-six percentage computationally efficient than the next efficient server side logging existing model to account the request history in the web service environment (especially for critical infrastructures like government websites) and help in detecting the spam request. The future work of this paper is to efficiently analyze the accounting model through History Analyzer to detect the spam requests.

References

Alfantookh, Abdulkader.A., 2006. DoS attacks intelligent detection using neural networks. Journal of King Saud University – Computer and Information Science 18, 27–45.

Alhabeeb, Mohammed, Le, Phu Dung, Srinivasan, Bala. (2011a). "Preventing denial of service attacks in government e-services using a new efficient packet filtering technique". In: Proceedings of Ninth IEEE International Symposium on Parallel and Distributed Processing with Applications, pp. 262–269..

Alhabeeb, Mohammed, Le, Phu Dung, Srinivasan, Bala. (2011b). "Evaluating the functionality of the token filtering technique in filtering denial of service packets using a new formal evaluation model." In: Proceedings of Ninth IEEE International Symposium on Parallel and Distributed Processing with Applications, pp. 318–323..

Alhabeeb, Mohammed A., Almuhaideb, Abdullah, Le, Phu Dung, 2010a. Holistic approach for critical system security: flooding prevention and malicious packet stopping. International Journal of Telecommunications 1 (1), 14–24.

Alhabeeb, Mohammed, Alsunbul, Saad, Almuhaideb, Abdullah, Le, Phu Dung, Srinivasan, Bala. (2010b), "A novel security approach for critical information systems: preventing flooding in the non-authenticated client area using a new service from local network service providers." In: Proceedings of 2010 11th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, pp. 49–54..

Alonso, Gustavo, Casati, Fabio, Konu, Harumi, Machiraju, Vijay, 2004. Web Services: Concepts, Architectures and Applications. Springer.

Aura, Tuomas, Nikander, Pekka, Leiwo, Jussipekka, 2000. DOS-resistant authentication with client puzzles. Lecture Notes in Computer Science 2133, 170–177.

Baskaran, R., Saleem Basha, M.S., Balasubramanian, A., 2012. Two Tier Security Framework for Service Oriented Systems: Intelligent Security Measures. LAP LAMBERT Academic Publishing, pp. 1–176..

Bernstein, D.J. (1996). "Syn floods – a solution" <http://www.op.net/~jaw/syn-fix.html>..

Bona, R. (1996). "TCP SYN attacks – a simple solution". <http://www.cctec.com/maillists/nanog/historical/9610/msg00155.Html>..

Casado, Martin, Cao, Pei, Akella, Aditya, Provos, Neils. (2006). "Flow-cookies: using bandwidth amplification to defend against DDoS flooding attacks" Stanford HPNG Technical Report..

Das, D., Sharma, U., Bhattacharyya, D. K. (2011). "Detection of HTTP flooding attacks in multiple scenarios." In: Proceedings of the 2011 International Conference on Communication, Computing & Security (ICCCS'11), pp. 517–522..

Denial of service attack, White paper by Software Engineering Institute, Carnegie Mellon University, (2001). <http://www.cert.org/tech_tips/denial_of_service.html> [Latest access on 18 August 2011]..

Eddy, Wesley M., 2006. Defenses against TCP SYN flooding attacks. The Internet Protocol Journal 9 (4), 2–16.

Eid, Mohamad Samir A., Aida, Hitoshi, (2010). "Securely hiding the real servers from DDoS floods."In: Proceedings of 10th Annual International Symposium on Applications and the Internet, pp. 165–168..

Feng, Wu-chang, Kaiser, Ed, (2010). "kaPoW Webmail: effective disincentives against spam." In: Proceedings of CEAS 2010 – Seventh Annual Collaboration, Electronic messaging, Antiabuse and Spam Conference, pp. 1–9..

Fu, Kevin, Sit, Emil, Smith, Kendra, Feamster, Nick, (2001). "Dos and don'ts of client authentication on the web". In: Proceedings of the 10th USENIX Security, Symposium, pp. 1–16..

Gaddam, Ajit, (2008). "Yahoo! CAPTCHA Cracked", Root 777 posts, January 31. <http://www.root777.com/hacking/yahoo-captcha-cracked> [Latest access on 18 August 2011]..

Hang, Bo, Hu, Ruimin, (2009) "A novel SYN cookie method for TCP layer DDoS attack."In: Proceedings of 2009 International Conference on Future BioMedical Information, Engineering, pp. 445–448..

Jensen, Meiko, Schwenk, Jorg, (2009). "The accountability problem of flooding attacks in service-oriented architectures." In: Proceedings of 2009 International Conference on Availability, Reliability and Security (2009 ARES), pp. 25–32..

Jensen, Meiko, Gruschka, Nils, Herkenhoner, Ralph, Luttenberger, Norbert, (2007). "SOA and Web Services: new technologies, new standards new attacks." In: Proceedings of the 5th IEEE European Conference on Web Services, pp. 35–44..

Jensen, Meiko, Gruschka, Nils, Luttenberger, Norbert, (2008). "The impact of flooding attacks on network-based services." In: Proceedings of The Third International Conference on Availability, Reliability and Security, pp. 509–413..

Kandula, S., Katabi, D., Jacob, M., Berger, A. W. (2005) "Botz-4-sale: Surviving organized DDoS attacks that mimic flash crowds", in proceedings of. Symposium on Networked Systems Design and Implementation (NSDI), Boston, May 2005..

Khor, Soon Hin, Nakao, Akihiro, (2011). "DaaS: DDoS mitigation-as-a-service."In: Proceedings of 2011 IEEE/IPSJ International Symposium on Applications and the Internet, pp. 160–171..

Oppermann, A., (2006) "FYI: Extended TCP syncookies in FreeBSD-current", Post to the tcpm mailing-list. <http://www.ietf.org/mailarchive/web/tcpm/current/msg02251.html>..

Panigrahi, Suvasini, Kundu, Amlan, Sural, Shamik, Majumdar, A.K., 2009. Credit card fraud detection: A fusion approach using Dempster–Shafer theory and Bayesian learning. International Journal on Information fusion 10, 354–363.

Park, Joon S., Sandhu, Ravi, 2000. Secure cookies on the Web. IEEE Internet Computing, 36–44.

Pujolle, Guy, Serhrouchni, Ahmed, Ayadi, Ines., (2009)."Secure session management with cookies." In: Proceedings of the 7th international conference on Information, communications and, signal processing, pp. 1–6..

Ranjan, S., Swaminathan, R., Uysal, M., Knightly, Edward W. (2006) "DDoS-Resilient Scheduling to Counter Application Layer Attacks Under Imperfect Detection", In: proceedings of 25th IEEE International Conference on Computer Communications (INFOCOM 2006), pp. 1–13..

Ricciulli, L., Lincoln, P., Kakkar, P., 1999. TCP SYN flooding defense. CNDS.

Sit, Emil., Fu, Kevin., 2001. Web cookies: not just a privacy risk. Communications of the ACM 44 (9), 120.

Srivatsa, Mudhakar, Iyengar, Arun, Mikalsen, Thomas A., Rouvellou, Isabelle, Yin, Jian, (2007). "An access control system for web service compositions". In: Proceedings of the IEEE International Conference on Web Services (ICWS), pp. 1–8..

Suriadi, Suriadi, Clark, Andrew, Schmidt, Desmond, (2010). "Validating denial of service vulnerabilities in web services". In: Proceedings of 2010 Fourth International Conference on Network and System, Security, pp. 175–182..

Wu, Heng, Chen, Weiting, Ren, Zhongjie, (2010). "Securing cookies with a MAC address encrypted key ring". In: Proceedings of 2010 Second International Conference on Networks Security, Wireless Communications and Trusted, Computing, pp. 62–65..

Xu, Donghua, Lu, Chenghuai, Dos Santos, Andre, (2002). "Protecting web usage of credit cards using one-time pad cookie encryption". In: Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC'02), pp. 51–58..

Yuan, Jian, Mills, Kevin, 2005. Monitoring the macroscopic effect of DDoS flooding attacks. IEEE Transactions on Dependable Secure Computing 2 (4), 324–335.

Yue, Chuan, Wang, Haining, 2009. Profit-aware overload protection in E-commerce Web sites. Journal of Network and Computer Applications 32 (2), 347–356.