Robust and Fast Authentication of Session Cookies in Collaborative and Social Media Using Position-Indexed Hashing

Amerah Alabrah
Department of Electrical Engineering and Computer Science
University of Central Florida, Orlando, Florida
College of Computer and Information Sciences
King Saud University, Riyadh, Saudi Arabia
amerah@knights.ucf.edu

Mostafa Bassiouni
Department of Electrical Engineering and Computer Science
University of Central Florida, Orlando, Florida
USA
bassi@cs.ucf.edu

ABSTRACT*—The use of insecure cookies as a means to authenticate web transactions in collaborative and social media websites presents a hazard to users' privacy. In this paper, we propose and evaluate a novel protocol for protecting transmitted cookies using two dimensional one-way hash chains. In the first dimension, there is a hash chain that computes secret values used in the second dimension hash function. Multiple hash chains use the secret values created by the first dimension to authenticate session cookies in the second dimension. For improved security, the hashing operations in the second dimension use a concatenation of the secret values and the position index of the hash function within the hash chain. The performance of the scheme is evaluated using a detailed simulation testbed and an analytical model. The optimal lengths of the chains are derived when the number of transactions in the session is known. The protocol is extended to efficiently handle the case when the number of transactions is not known. The evaluation of the proposed scheme reveals that it achieves tremendous improvement over straightforwardly configured one-way hash chain schemes. Also, by adopting the position-indexed hashing protocol, energy consumption is reduced significantly especially with longer sessions making our protocol ideal for battery operated devices.

Keywords: One-way hash chains, HTTPS, Session cookies.

I. INTRODUCTION

Many collaborative websites and social media networks utilize session cookies as a cheaper alternative to the wide utilization of the secure HTTPS protocol. The unprotected nature of cookies can compromise the collaborative environment. Evidently, the availability of social networks and collaboration websites where access to the website is extended to long durations has made this issue even more pressing. Although using a secure protocol (e.g. HTTPS) to connect to the web provides higher levels of security, it is not always applied by many web servers and is replaced by cookie protection. The nature of cookies as plain text stored at the client's side makes it not too complicated for an adversary to hack these cookies and steal the Internet session leading to a compromise in the users' overall Internet experience.

To avoid this shortcoming of Internet cookies, researchers such as [2, 5] suggest using one-way hash chains to secure the transmission of cookies. The idea of one-way hash chains is based on Lamport's one-way chains for one-time passwords authentication [7], which was later formulated by Haller to the S/Key standard [6]. The main advantage of one-way hash chains is that once the authentication credentials are used, they are recycled and never used again. This minimizes the chances of cookies being sniffed out and abused for unlawful utilization by entities other than the respective parties.

A. Contribution

Despite the capability of one-way hash chains in transmitting Internet cookies securely if appropriate cryptographic hash functions are adopted, their high computational overhead makes them far from optimal. In this paper, we address this particular shortcoming and propose a scheme to deal with the computational overhead of one-way hash chains for a faster cookie transmission. Our scheme utilizes the idea of layered one-way hash chains in which hashing is conducted using the concept of position-indexing.

The remainder of this paper is organized as follows. In section 2, we survey the related literature. In section 3, we introduce the protocol. In section 4, we provide a discussion of how the protocol functions when the number of transactions is known. We also overview the testbed used and the analytical model and present the simulation results. In section 5, we address the case when the number of transactions is not known and present the evaluation results. We conclude the paper with section 6.

COLLABORATECOM 2013, October 20-23, Austin, United States

DOI 10.4108/icst.collaboratecom.2013.254126

II. PREVIOUS WORK

The issue of session hijacking or 'sidejacking' due to sniffing out of Internet cookies is one of the important Internet security concerns. Session hijacking results from unlawful control over cookies during an ongoing internet session in an unprotected network where plaintext traffic is unencrypted. Illustrations such as [9] and [1] show how cookies are vulnerable to attacks, which makes their current deployment questionable and warrants a search for more reliable and secure techniques. Several researchers have tried to solve the vulnerability of cookies. For example, the use of an external proxy where authentication and sensitive information management is carried out completely at the proxy or some other external device (e.g. a user's cell phone) is a possible alternative proposed by [10] and [14]; however, this solution's implementation can pose difficulties as it might not be optimal in all situations. Specifically, if a user does not have access to the proxy for any reason or in case the external device is not available at the time when the service is desired (e.g. cellphone battery dead, no coverage…etc.), he will not be able to use the service.

Several proposals have tried to address the problem of session cookies' exploitation by adopting schemes which rely on Lamport's one-way passwords. For instance, [8] proposed a solution that targets the read-only property of the session cookies in the website's databases. They achieve their protection by leveraging the read property so that it becomes hard for an attacker to correctly guess the cookie value. Conceptually, they suggest including an iterated hashed value of the user's password and its pre-image in the session cookies. These two values are compared each time a communication between the server and client is desired. To strengthen the cookies, they add a *salt* value which they claim makes it even harder for an adversary to detect the users' private information.

In a recent paper, Dacosta et al [5] proposed using a modified hash construction to generate disposable credentials (One-time cookies; OTC) in lieu of cookies to be used only once during a session. While their solution achieves session cookies integrity, it suffers from an unjustified computational overhead. The overhead is a result of the need to establish a certain number of transactions between the server and client expected to be handled during the lifetime of the session. If this number is underestimated, the session will be terminated prematurely and the user will be required to initiate a new session. When the opposite happens (i.e. the session is overestimated), the connection will suffer from an unjustified overhead due to the high cost of the early transactions.

In an attempt to lower this computational overhead, the authors of [3] proposed a protocol which essentially imitates the rolling code technology used to protect garage codes from being detected and compromised. The Rolling Code protocol replaces the hash chain performed by the OTC in each transaction by two hash operations: one to update and randomize the value of a variable d = hash(d), and the other to produce a one-time authentication token by applying a hash function on the Exclusive-OR of a secret seed and the new value of d. The Rolling code protocol is less robust than the one-way hash chain approach (e.g., the OTC protocol), but is lightweight and more suitable for mobile phones and PDA's.

The SCRHC scheme proposed in [2] improves the performance of one-way hash chains by utilizing a flexible caching component in which the hash values at certain points in the chain are stored for use in future iterations. The basic step executed in the Repeat_Chain routine of the SCRHC scheme updates the secret s by computing the concatenation of the secret s with itself, s := hash (s ‖ s). This is considered a security weakness that reduces randomness and makes the scheme vulnerable to certain types of attack. The two dimensional scheme proposed in this paper eliminates this weakness because it updates the secret s using a second dimension of one way hash chains and employing a position indexing technique as explained in section 3.

In designing our protocol, we took into consideration different cryptographic approaches. While one of our main objectives is reducing the computational overhead of one-way hash chains based cookies, we wanted our protocol to benefit from the features of current cryptographic approaches especially their strength and resistance to replay attacks, collision attacks, pre-image attacks and second pre-image attacks. Hence, our protocol is designed with the state-of-the art cryptographic approaches in mind.

A. One-way Hash Cookie (OHC) Protection

Since we are using the *one-way hash cookie protection scheme* as the backbone for our solution, it is worth illuminating its main aspects and how its hashing operation is carried out to protect cookies. In the OHC scheme, a one-way hash chain of length N is used to protect a stream of N transactions of a web session. During the initial HTTPS login step, the server and the client exchange a shared secret value S_0, and a value N which refers to the chain length or number of transactions expected to be handled during a session. The OHC protects the j^{th} transaction by computing an authentication token $V_j=H^{N-j+1}(S_0)$, where the notation $H^m(x)$ implies applying the hash function m times, for example, $H^2(x)= H(H(x))$. For instance, if N=100, then the authentication tokens for the 1^{st}, 2^{nd}, and 3^{rd} transactions are $V_1=H^{100}(S_0)$, $V_2=H^{99}(S_0)$, $V_3=H^{98}(S_0)$, respectively. Figure 1 illustrates how the one-way hash chains are configured. The straight arrow going from the left to the right corresponds to the length of the chain. In this specific figure, the length is 5 transactions. The small arrows going from the right to the left represent the points where authentication tokens are generated and checked. At each point in the hash chain, the server and client must be able to derive the same value of the authentication token.

Otherwise, a red flag is raised and the whole session might have been compromised. Therefore, the user needs to be asked for login information again.

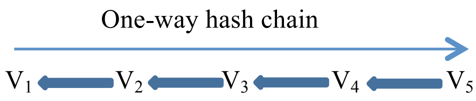

Figure 1: One-way hash chains

The main drawback of the OHC approach is its high computational overhead described above (i.e. overestimation or underestimation of the number of transactions in a session.) In this paper, we propose a scheme to significantly reduce the overhead of OHC without deploying cache memory to store the authentication tokens.

III. THE PROPOSED PROTOCOL

A. Conceptualization

Conceptually, the one way hash chains in our protocol are arranged in two dimensions (Figure 2). In the first dimension (i.e. horizontal axis), there is a single hash chain that computes the seeds for the second dimension chains (i.e. vertical axis). In the second dimension, multiple hash chains use these seeds to generate authentication tokens. The authentication tokens are derived by hashing the seeds and the position of the hashing functions in the hash chains (e.g. via a concatenation process ||). Given the cryptographic hash function used is resistant to attacks (e.g. SHA-1, SHA-2 or SHA-3), a slight change in the argument to be hashed is expected to result in a significantly different output. Figure 2 provides a conceptual view of how our protocol functions.

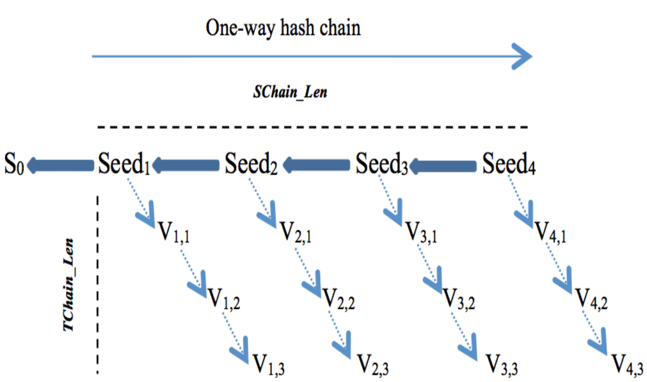

Figure 2: Position-indexed hashing for 12 transactions *TChain_Len*= 3, *SChain_Len*= 4

The proposed protocol is composed of three main stages: the *Initialization stage, Authenticate_Token stage* and *Next_Seed stage*. The notations we use in our scheme are summarized in Table 1.

1) Initialization stage

During the initialization stage, which is done using an HTTPS protocol, information about the session length (i.e., number of transactions N), an initial secret S_0 and *TChain_Len* is exchanged between the server and the client. Once this information is exchanged, the *SChain_Len* is determined by dividing N by *TChain_Len.* The result of this division will give us the number of seeds that will be needed during an internet session. Our definition of a session refers to the communication activities between the web application and the client during the login time (i.e. between log-in and log-out). A transaction on the other hand is a set of request and response between the web application and the client. The session is composed of N transactions. Information about the session length, *TChains* are predetermined and exchanged during this stage.

TABLE I. NOTATIONS USED IN THE PROPOSED SCHEME

Notations	**Description**
N	Number of transactions to be handled during an internet session.
SChain	The chain where seeds are generated.
TChain	The chain where authentication tokens are generated.
S_0	The initial seed used by the *SChain*
SChain_Len	Length of the *SChain*.
TChain_Len	Length of the *TChain*.
H	Hash function used to generate seeds or authentication tokens.
V	Authentication token.

2) Authenticate_Token stage

The next stage *Authenticate_Token*, is where the authentication tokens are actually produced. The authentication tokens are denoted $V_{i,j}$ where the variable *i* represents the current *TChain* and *j* represents the current transaction number within the *TChain*. The tokens are created by hashing the seed concatenated || with a variable indicating the position of the hash function in the *TChain*. This position indexing technique is a well-known technique for boosting security because Birthday Attacks can be avoided if all hash functions used are indexed by their position in the chain [15]. As will be explained in the *Next_Seed stage,* we also update the seed several times during the session. The number of times the seed is updated depends on the number of transactions and the value of *TChain_Len*. This number is used to indicate how many *TChains* we will have during the session. In other words, each updated seed is only used by the transactions of one *TChain* and then discarded and never used again.

3) Next_Seed stage

The third component of the protocol is the *Next_Seed* routine. This routine is responsible for updating the seeds used in the *TChains* to generate authentication tokens. It should be noted that each *TChain* has its own seed. This routine is invoked once the authentication tokens of the first *TChain* are created and transmitted. Based on the number of transactions and *TChain_Len* exchanged in the initialization stage, we know the number of times the seed is expected to be updated. The length of the seed chain, *SChain_Len,* is a result of dividing the number of transactions *N* by the value *TChain_Len.* Once the authentication tokens have all used a seed once (i.e. *TChain_Len* is exhausted), the *Next_Seed* routine is invoked to produce an updated seed for the next authentication token chain; *TChain*. We illustrate in the following section how our protocol works with a pseudo code and detailed examples. The performance evaluation results of the proposed scheme are presented in sections 4 and 5.

B. Selecting a Cryptographic Hash Function

A cryptographic hash function is an algorithm which changes a certain set of data into a string of a fixed size, called the block size. Examples of cryptographic hash functions include MD4, MD5, SHA-1 and SHA-2. It was proven that the MD5 hash function is prone to collision attacks [11] , [4] as well as pre-image attacks [12], and therefore, we did not consider it in our scheme. While SHA-1 is resistant to pre-image attacks, it was proven by [13] that it is theoretically prone to collision attacks. However, since it is not practically susceptible to collision attacks, we have used it in our protocol for the purpose of illustration.

In our implementation, the original block size is 160-bit corresponding to SHA-1, but it can easily be expanded to accommodate stronger cryptographic techniques that require larger block sizes such as SHA-2 (in all its sizes) and SHA-3 once it is released by NIST.

IV. CASE OF KNOWN NUMBER OF TRANSACTIONS

Accurate statistics about network traffic related to social networking sites can be helpful in identifying the length of the one-way hash chain. However, it is not always the case that these are readily available. Dacosta et al [5] conducted basic traffic analysis of the social networking site “Facebook” and concluded that a typical session requires hundreds of transactions, and thus they set their chain length at 1000. In our study, we have varied this chain length since different social networking sites might have different requirements. Following are the steps of the protocol when the number of transactions is known.

A. The proposed protocol’s steps

The *initialization* stage takes place using an HTTPS connection. During the HTTPS authentication, the initial value of the secret key S_0, the number of transactions *N* and the length of *TChain* (i.e. *TChain_Len*) are selected and exchanged between the server and the client. The following code is executed at both the client and the server sides.

```
SChain_Len:= N ÷ TChain_Len // length of the SChain
I:= SChain_Len              // I is the global index for the SChain
J:= TChain_Len              // J is the global index for the TChain
Seed:= H^I(S0)              // Seed is now Seed1=seed for the first TChain
```

The routine *Authenticate_Token* is executed once for each transaction to compute the authentication tokens that will be transmitted with the transaction cookie.

```
Authenticate_Token(Seed, J)
Begin
V:= H^J(Seed||J )    // J is the global index for the TChain where || is a
                     concatenation of the seed with the hash function
                     position in the chain
J:= J -1
if (J==0) then                  // TChain length is exhausted
  Seed:= Call Next_Seed( )      // Seed has to be updated
  J:= TChain_Len                // TChain length is reset
end-if
Return (V)
End

Next_Seed( )
Begin
I:= I-1                         // I is the global index for the SChain
Seed:= H^I(S0)                  // updating the Seed value
Return (Seed)
End;
```

Let us now illustrate how the protocol works with an example. In case the number of transactions is known to be *N* =200, and the *TChain_Len* =4, the seed is going to be updated 50 times (i.e., *SChain_Len = 50*) to carry out the hashing functions for 200 transactions.

We have I=50, J=4, $Seed_1$= $H^{50}(S_0)$

The first *TChain* of four transactions will create the following authentication tokens.

$V_{1,1}$= $H^4(Seed_1||4)$

$V_{1,2}$= $H^3(Seed_1||3)$

$V_{1,3}$= $H^2(Seed_1||2)$

$V_{1,4}$= $H^1(Seed_1||1)$

Once these authentication tokens have been transmitted, the *Seed* has to be updated to $Seed_2$= $H^{49}(S_0)$ and *J* has to be reset to 4.

The next step is to generate the second set of four transactions which will be:

$V_{2,1}$= $H^4(Seed_2||4)$

……..

$V_{2,4}$= $H^1(Seed_2||1)$

The code continues to calculate the authentication tokens in each *TChain* until we reach the 50th *TChain*. The 50th *TChain* will have the $Seed_{50}$= H(S_0) and its authentication tokens will be:

$V_{50,1}$= $\mathrm{H}^4(Seed_{50}||4)$

……..

$V_{50,4}$= $\mathrm{H}^1(Seed_{50}||1)$

B. Protocol Evaluation

1) The Testbed

In this section, we present the protocol evaluation results when the number of transactions in a session is known. We developed a detailed benchmark in Java which allowed us to test different session scenarios. An important metric used in our tests is *SessionCost* which is the total number of hash operations performed during the lifetime of the session. The metric *SessionCost* represents the overall execution overhead of the protocol including the overhead of the *Initialization stage* and the overhead of the *Authenticate_Token* routine for all transactions as well as the overhead of the *Next_Seed* routine.

Figure 3 shows the performance of the protocol for different values of the number of transactions *N* and the length of *TChain*. It is interesting to see that the value of *SessionCost* decreases as the value of *TChain_Len* increases until a certain point then starts to increase again. For each value of N, there is a certain value of *TChain_Len* that minimizes the value of *SessionCost*. We validate this behavior by an analytical model.

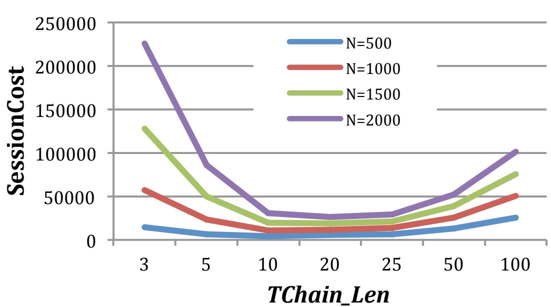

Figure 3: Protocol Evaluation (known number of transactions)

2) Analytical Model

As we described earlier our protocol is composed of two chains: i) the seed generating chain *SChain* represented by the horizontal axis in Figure 2, and ii) the authentication generating chain *TChain* represented by the vertical (slanted) axis in Figure 2. For simplicity, we assume that the cost of a single hash operation used in the *SChain* and *TChain* is the same because they both use the same hashing algorithm (i.e. SHA-1); we will examine this assumption later at the end of this section. The cost of a single session *SessionCost*= *C* is the sum of the hashing operations required to generate authentication tokens in the vertical chains, C_V, and the hashing operations required to update the seeds in the horizontal chain, C_H. Here is how *SessionCost* is calculated.

N= number of transactions
M= *SChain_Len*
K= *TChain_Len*

Cost of one vertical chain = $K(K+1)/2$

$C_V = MK(K+1)/2 = N(K+1)/2$

$C_H = M(M+1)/2$

$C = C_V + C_H$
$\quad = N(K+1)/2 + M(M+1)/2$

The next formula can be used to plot *C* as a function of *N* and *K*.

$$C = \frac{N(K+1)}{2} + \frac{M(M+1)}{2} = \frac{NK}{2} + \frac{N}{2} + \frac{M^2}{2} + \frac{M}{2}$$

$$C = \frac{NK}{2} + \frac{N}{2} + \frac{N^2}{2K^2} + \frac{N}{2K} = \frac{N}{2}(K + 1 + \frac{N}{K^2} + \frac{1}{K}) \qquad (1)$$

To find the optimal value of *K* which minimizes the cost *C*, we differentiate formula (1) and equate to 0

$$\frac{\partial C}{\partial K} = \frac{N}{2}(1 - \frac{2N}{K^3} - \frac{1}{K^2})$$

$$1 - \frac{2N}{K^3} - \frac{1}{K^2} = 0$$

$$K^3 - K - 2N = 0 \qquad (2)$$

Equation 2 can be used to derive the optimal value of *K* which corresponds to *TChain_Len*. Table 2 gives the optimal value of *TChain_Len* obtained by solving the above cubic equation numerically.

TABLE I: *TChain_Len* OPTIMAL VALUE

Number of Transactions	Optimal *TChain_Len*
500	10.03
1000	12.625
1500	14.445
2000	15.895

Comparing Table 1 with Figure 3, we can see that the *TChain* optimal values which we obtained from the simulation are very close to the optimal values obtained from the analytical solution.

It should be mentioned that the above analytical solution was derived based on the assumption that the hash operation used in the *SChain* and *TChain* have the same cost because both use the same hash function SHA-1. A more accurate model can be easily developed to account for the extra overhead of the concatenation operation used in the *SChain*. Since the cost of the concatenation operation is much smaller than the cost of the hash operation, the slight increase due to concatenation can be accurately modeled by multiplying the cost of C_H by a factor r which is slightly larger than 1. This will cause Equation 2 to be slightly modified as follows: the second and third terms will be multiplied by the factor r. Solving this modified equation gives optimal values very close to those given in Table 2.

C. Protocol Comparison with OHC

In order to validate the efficiency of our protocol, we compared its performance with the OHC protocol proposed in [5].

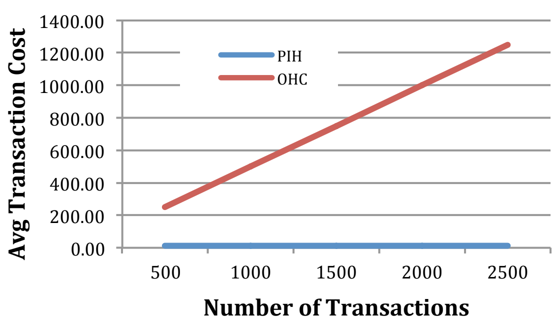

Figure 4: Performance comparison between position-index hashing (PIH) and OHC

In Figure 4, we demonstrate the performance comparison between our protocol and the OHC protocol. We compared the average cost of a single transaction in both protocols. In our protocol, we were able to lower the average cost of the transaction tremendously. For instance, for 500 transactions the average transaction cost of the OHC protocol is 250, whereas our protocol lowers this average to a little over 11. While this average tends to increase significantly with the increase in transaction numbers for the OHC protocol, our average stays relatively low even with N= 2500 transactions where the average cost is 13.65.

In order to further gain insight in the benefits of adopting our protocol, we measured the performance improvement ratio when our protocol is chosen over the OHC. The performance improvement ratio is defined as the ratio *SessionCost* of OHC : *SessionCost* of our PIH protocol. Figure 5 displays the significant improvement our protocol achieves over OHC.

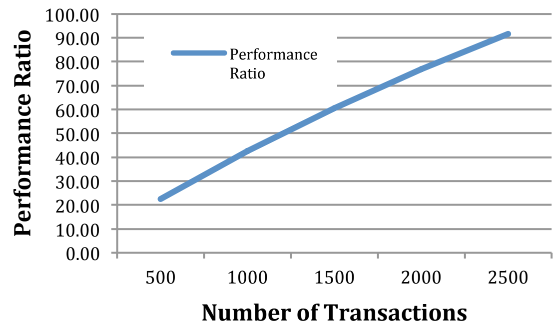

Figure 5: Performance improvement ratio of OHC

Looking at Figure 5, we can easily gain insight on how choosing our protocol is beneficial. Our protocol outperforms the OHC protocol by a little over 22 times when the number of transactions is 500. This improvement ratio is much higher with higher transaction numbers. When we have 2500 transactions to be handled in a session, our protocol outperforms the OHC by over 91 times. This is a relatively wide margin and makes our protocol plausible. The ratio is expected to be higher for longer sessions with higher transaction numbers.

V. UNKNOWN NUMBER OF TRANSACTIONS

A. Certainty versus Uncertainty in Transaction Number

In designing the protocol, we took into consideration the possibility of certain versus uncertain number of transactions. More often than not, it is very hard to estimate the exact number of transactions to be handled in a single session in a social networking site. As we have seen above in our discussion of Facebook session length statistics introduced in [5], the length can range from a few hundred transactions to several hundreds. This has led us to devise two different versions of the position-indexed hashing protocol to accommodate the two scenarios: *known number of transactions* and *unknown number of transactions*.

We have already accounted for the case of known number of transactions in the previous section, and this section is devoted to explicating the case of unknown number of transactions. When the number of transactions is unknown, there is no way to calculate the *SChain_Len* and hence the number of *TChains* in a session. Therefore, we needed to change the code slightly to account for this discrepancy. During the *Initialization stage* instead of exchanging the number of transactions N and the *TChain_Len*, the client and server exchange the *TChain_Len* and another value representing the *SChain_Len*. The importance of *SChain_Len* specification comes from the need to update the seed during the session multiple times. Since the transaction number is unknown, we have no way of determining how many times the seed is going to be updated. Given this scenario, we are faced with another problem. If the specified *SChain_Len* is not long enough (i.e., the actual number of times we will have to update the seed is more than the value of *SChain_Len*), we will need to repeat using one or more seeds, which could compromise the security of the session. To solve this problem, we utilize the number of *TChains* in a session as an index to be attached to the updated seed via a concatenation process ||. The index for *TChain* number can be derived from the *TChain_Len* and the *Next_Seed* routine. The first *TChain* in a session is the one that uses the first seed and once the *Next_Seed* routine is invoked, the *TChain* index is incremented and the new value is attached to the hashed seed.

B. The Modified Protocol

Here, we introduce how we modified the protocol to account for the case of unknown number of transactions. We still have the three stages we had in the unknown number of transactions case.

1) Initialization

The initialization stage takes place using an HTTPS connection. During the HTTPS authentication, the initial value of the secret key S_0, the length of the authenticate-token chain *TChain_Len* and the length of the next-seed chain *SChain_Len* are selected and exchanged between the server and the client. The following code is executed at both the client and the server sides.

```
I:= SChain_Len        // I is the global index for the SChain

J:= TChain_Len        //J is the global index for the TChain

index:= 1             //a global variable indicating TChain
                        number where 1 refers to first TChain
Seed:= H^I(S0||index)   // Seed is now Seed1=seed for the first
                          TChain

Authenticate_Token(Seed, J)
Begin
V:= H^J(Seed||J )              // J is the global index for the TChain
J:= J -1
if (J==0) then                 // TChain_Len length is exhausted
   index:= index + 1           // index incremented for the next TChain
   Seed:= Call Next_Seed( )    // Seed has to be updated
   J:= TChain_Len              // TChain length is reset
end-if
Return (V)
End

Next_Seed( )
Begin
I:= I-1
if (I==0) then            // SChain length is exhausted
  I:= SChain_Len          // I is reset to SChain_Len
end-if
Seed:= H^I(S0||index);
Return (Seed)
End
```

Here is an example to help illustrate how the protocol handles the transmission of authentication tokens when the number of transactions is unknown. During initialization, the values of *TChain_Len*=4 and *SChain_Len*=10 will be selected and exchanged between the server and client.

We have I=10, J=4, *index*=1. We assign an index which represents the first *TChain*, and therefore the first seed will be:

$Seed_1 = H^{10}(S_0 \| 1)$

The first *TChain* will use $Seed_1$ in the hashing function to derive the first set of authentication tokens as follows…

$V_{1,1} = H^4(Seed_1\|4)$

$V_{1,2} = H^3(Seed_1\|3)$

$V_{1,3} = H^2(Seed_1\|2)$

$V_{1,4} = H^1(Seed_1\|1)$

Now *index* becomes 2 which represents the second *TChain*, I =9, *Seed* has to be updated to $Seed_2 = H^9(S_0\|2)$ and J has to be reset to 4.The second *TChain* will have the following authentication tokens:

$V_{2,1} = H^4(Seed_2\|4)$

……..

$V_{2,4} = H^1(Seed_2\|1)$

After finishing ten *TChains* (i.e. 40 transactions) *index* becomes 11 which represents the 11th *TChain*, I has to be reset to 10, *Seed* has to be updated to $Seed_{11} = H^{10}(S_0\|11)$ and J has to be also reset to 4. If we did not use the *TChain* number as a value attached to S_0 , we would have been forced to recycle $Seed_1$ as $Seed_{11}=Seed_1=H^{10}(S_0)$. This could potentially compromise our protocol as it becomes easier to detect the initial seed. By indexing the *TChain* number and using its value in the hashing function, we are able to solve this problem.

Therefore, the 11th *TChain* (11th set of four transactions) will have authentication tokens which will be:

$V_{11,1} = H^4(Seed_{11}\|4)$

……..

$V_{11,4} = H^1(Seed_{11}\|1)$

The protocol goes on according to this routine until the user or the server terminates the session.

C. Protocol Evaluation (unknown number of transactions)

In this section, we present the evaluation of our protocol when the number of transactions is unknown. Figure 6 illustrates the results when the *SChain_Len* is fixed at 5, while Figure 7 demonstrates the protocol's performance when the *SChain_Len* is fixed at 20. Our main goal from this is to determine the best value of *TChain_Len* where the protocol performs relatively well.

In both Figure 6 and Figure 7, regardless of the *SChain_Len*, the protocol seems to perform well when the *TChain_Len* is set at 3. Unlike the case of known number of transactions where we noticed some kind of correlation between *TChain_Len* and performance, in the case of unknown number of transactions it is better to set *TChain_Len* at a relatively low value.

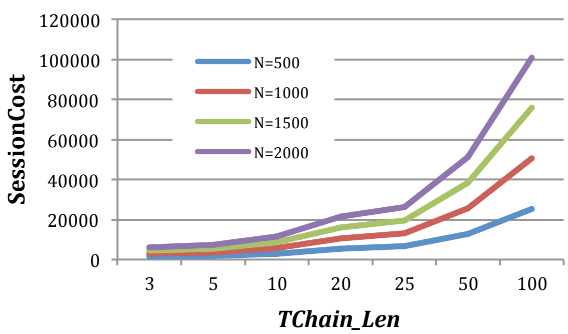

Figure 6: Protocol Evaluation (Unknown number of transactions) *SChain_Len= 5 all the time*

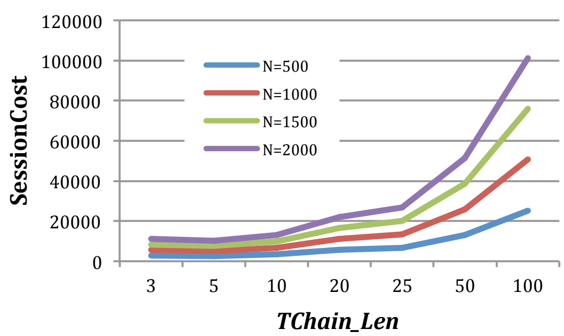

Figure 7: Protocol Evaluation (Unknown number of transactions) *SChain_Len= 20 all the time*

Our next task was to see what the best value of *SChain_Len* is when the *TChain_Len* is fixed at 3. The next graph (Figure 8) represents the session cost of different number of transactions with *TChain_Len*= 3 and different *SChain_Len* values.

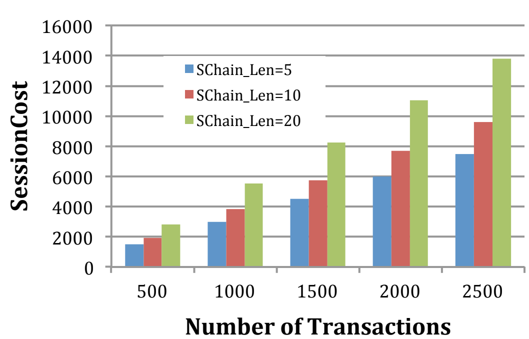

Figure 8: Protocol performance comparison when *TChain_Len*=3 and different *SChain_Len*

Figure 8 indicates that there is a steady and direct relationship between *SChain_Len* and performance measured in *SessionCost*. The lower the value *SChain_Len*, the better performance we can achieve when we do not know the number of transactions during a session. In other words, we need to start with relatively short chains in both the authenticate-token chain *TChain* and the next-seed chain *SChain*.

D. Protocol Evaluation (Energy consumption)

When designing any authentication protocol, equally important to efficient computational overhead is how much energy is expended. According to [16], there are at least three approaches to preserving battery life in mobile devices: efficient hardware, accurate knowledge of energy consumption of different cryptographic approaches and light weight security mechanisms.

Energy consumption is largely influenced by the cryptographic hash function used in the authentication scheme as different hash functions have different energy consumption levels. The authors of [17] conducted an extensive analysis of energy characteristics of various cryptographic approaches and found that energy varies according to the cryptographic approach utilized. For SHA, SHA1 and HMAC, the energy required to conduct a single operation is 0.75, 0.76 and 1.16 microjoule/byte, respectively (for a complete list of energy consumption characteristics of different cryptographic approaches we refer the reader to [17]).

Since the cost of the session is what determines the amount of hashing operations required to implement authentication, energy consumption is correlated with the session cost described in section 4.2. In our evaluation, we demonstrate the energy consumption of our position-indexed hashing protocol and compare it with the OHC in the unknown number of transactions case.

It should be noted though that the initialization phase is not included in this comparison because it is conducted using an HTTPS. As [17] indicates the energy consumption in the SSL protocols is influenced by the transaction size which is not the scope of the current paper.

Figure 9 demonstrates the energy consumption comparison between our position-indexed hashing protocol and the straight forwardly configured one-way hash chains. While the two protocols are comparable in the lower number of transactions, our PIH protocol clearly wins in the longer sessions as can be seen when the number of transactions is high. This results in a better battery conservation as the OHC drains energy resources much faster.

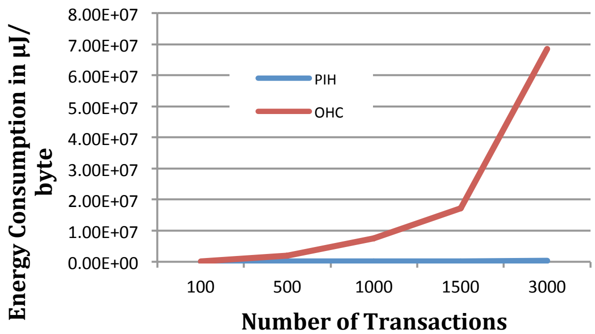

Figure 9: Energy consumption comparison between the OHC and PIH in the unknown number of transactions case

VI. CONCLUSION

One-way hash chains can be efficiently used in collaborative and social media networks to overcome the problem of session hijacking in Internet sessions caused by stealing cookies. Due to the computation overhead caused by overestimating the length of internet sessions, they have not been widely utilized. In this paper, we proposed a one-way hash chain protocol to address the problem of overestimating the number of transactions during a session in the straightforwardly configured one-way hash chains. Our solution achieves its goal by utilizing two one-way hash chains; one is responsible for updating the secret and the other for creating the authentication tokens attached to the cookies using the secrets produced by the first chain. We also employ the position of the hashing function in the chain in order to strengthen our protocol against attacks such as Birthday attacks.

Our extensive evaluation of the protocol and comparison with other protocols yielded encouraging results. We have been able to improve the performance of one-way hash chains significantly while keeping the same levels of security. By adopting the position-indexed hashing protocol, energy consumption is reduced significantly especially with longer sessions making our protocol ideal for battery operated media.

References

[1] E. Butler. FireSheep: Cookie Snatching Made Simple. ToorCon Conference, San Diego, CA, October 22-24, 2010. Software available at http://codebutler.com/firesheep

[2] J. Cashion, and M. Bassiouni. Protocol for mitigating the risk of hijacking social networking sites. Proceedings of the 7th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom'11), Orlando, FL, October 15-18, 2011.

[3] J. Cashion, M. Bassiouni. Robust and Low-Cost Solution for Preventing Sidejacking Attacks in Wireless Networks using a Rolling Code. In: Proceedings of the 7th ACM International Symposium on QoS and Security of Wireless and Mobile Networks (Q2SWinet'11), Miami Beach, Florida, pp. 21-26 (2011)

[4] S. Chen, and C. Jin. An Improved Collision Attack on MD5 Algorithm. Lecture Notes In Computer Science, pages 343–357, 2007.

[5] I. Dacosta, S. Chakradeo, M. Ahamad, P. Traynor. One-Time Cookies: Preventing Session Hijacking Attacks with Disposable Credentials. Technical Report, Georgia Institute of Technology, April 2011. Available at: http://smartech.gatech.edu/bitstream/handle/1853/37000/GT-CS-11-04.pdf

[6] N. Haller. The S/KEY one-time password system. RFC 1760, February 1995.

[7] L. Lamport. Password authentication with insecure communication. Communication of the ACM, Vol. 24, No. 11, 1981, pp. 770-772.

[8] S. Murdoch. Security Protocols XVI Hardened Stateless Session Cookies. Lecture notes in computer science. 2011;6615:93-101.

[9] B. Ponurkiewicz. FaceNiff- A new Android download application. Available at http://faceniff.ponury.net/.

[10] G. Pujolle, A. Serhrouchni, I. Ayadi. Secure session management with cookies. Information, Communications and Signal Processing, 2009. ICICS 2009. 7th International Conference on, vol., no., pp.1-6, 8-10 Dec. 2009

[11] M. Stevens. Fast collision attack on MD5, (2006) http://eprint.iacr.org/2006/104.pdf.

[12] X. Wang, X. Lai, D. Feng, H. Chen, X. Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 1–18. Springer, Heidelberg.

[13] X. Wang, Y. L. Yin, & H. Yu. Finding collisions in the full SHA-1, in V. Shoup, ed., 'CRYPTO', Vol. 3621 of Lecture Notes in Computer Science, Springer, pp. 17--36. 2005

[14] M. Wu, S. Garfinkel, R. Miller. Secure web authentication with mobile phones. In: DIMACS Workshop on Usable Privacy and Security Systems, July 2004

[15] Hu, Yih-Chun, Markus Jakobsson, and Adrian Perrig. "Efficient constructions for one-way hash chains." In *Applied Cryptography and Network Security*, pp. 423-441. Springer Berlin Heidelberg, 2005

[16] Chandramouli, R., Bapatla, S., Subbalakshmi, K., & Uma, R.: battery power-aware encryption. *ACM Transactions on Information and System Security (TISSEC)*, *9*(2), 162-180. 2006

[17] Potlapally, Nachiketh R., Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. "Analyzing the energy consumption of security protocols." In *Proceedings of the 2003 international symposium on Low power electronics and design*, pp. 30-35. ACM, 2003.