Int. J. Inf. Secur. (2014) 13:355–366
DOI 10.1007/s10207-013-0223-8

REGULAR CONTRIBUTION

Enhancing security of cookie-based sessions in mobile networks using sparse caching

Amerah Alabrah · Jeffrey Cashion · Mostafa Bassiouni

Published online: 19 December 2013
© Springer-Verlag Berlin Heidelberg 2013

Abstract The exponential growth in the use of mobile phones and tablets to gain wireless access to the Internet has been accompanied by a similar growth in cyber attacks over wireless links to steal session cookies and compromise private users' accounts. The popular one-way hash chain authentication technique in its conventional format is not optimal for mobile phones and other handheld devices due to its high computational overhead. In this paper, we propose and evaluate the use of sparse caching techniques to reduce the overhead of one-way hash chain authentication. Sparse caching schemes with uniform spacing, non-uniform spacing and geometric spacing are designed and analyzed. A Weighted Overhead formula is used to obtain insight into the suitable cache size for different classes of mobile devices. Additionally, the scheme is evaluated from an energy consumption perspective. We show that sparse caching can also be effective in the case of uncertainty in the number of transactions per user session. Our extensive performance tests have shown the significant improvement achieved by the sparse caching schemes.

Keywords Session cookies · Mobile devices · Wireless networks · Caching

A. Alabrah · J. Cashion · M. Bassiouni (✉)
Department of Electrical Engineering and Computer Science,
University of Central Florida,
Orlando, FL, USA
e-mail: bassi@cs.ucf.edu

A. Alabrah
e-mail: amerah@knights.ucf.edu

J. Cashion
e-mail: jcashion@knights.ucf.edu

1 Introduction

Wireless networks have more types of security threats and are much more prone to malicious attacks than wired networks. Serious security vulnerabilities exist in all types of wireless networks including 802.11 wireless LANs, wireless ad hoc networks, multihop wireless mesh networks and wireless sensor networks [1–4]. In this paper, we examine the particular security threat of hijacking user's private sessions over wireless links. This threat is increasingly on the rise due to two reasons: (1) the worldwide proliferation of smartphones, tablets and other handheld mobile devices as a primary tool for Internet access and (2) the increasing use of HTTP cookies by web applications to speed up responses to users and offer a better web experience that is more personalized and richer in interactivity. Since cookies are stored on the client machine, they are one of the most popular tools available to web developers to create better web applications without requiring significant server resources. Session cookies are being increasingly used for many purposes including session and transaction authentication, tracking of shopping cart contents, identification of user's preferences and tracking browsing behavior. Cookies usually exist as plain text on the client machine and can be tampered with by attackers if they succeed to compromise this machine.

Many Web sites encrypt the user's password and perform robust authentication using HTTPS during the initial login only, but do not apply the same level of costly HTTPS protection in further transactions. Web servers rely on session cookies saved locally at the client side to perform authentication after the initial login. Among the information stored in these cookies is a shared hashed secret, which is used as a proof that the user has been successfully authenticated at the initial login. As these cookies are transmitted over a wireless link using the unsecure HTTP protocol, the commu-

nication between the user and the web server is vulnerable to cookie hijacking. An attacker could take over a user's account by sniffing out the transmitted HTTP cookies. By hijacking session cookies, it becomes possible for the attacker to impersonate the victim and interact with the Web site without proper authorization.

In this paper, we investigate the problem of securing cookie-based sessions in wireless networks. We present techniques that enable mobile phones, tablets and similar wireless handheld devices to efficiently execute the one-way hash chain authentication technique and prevent attackers from getting a hold of a user's cookie in a wireless environment. Our proposed techniques are fast, lightweight and easy to implement in modern software.

The rest of the paper is organized as follows. In Sect. 2, we highlight some previous work that has been performed in this area. In Sect. 3, we define the one-way hash chain model used in our paper. In Sect. 4, we introduce the sparse caching approach and present its design details and introduce the Weighted Overhead (WO) formula. In Sect. 5, we discuss the results of our performance tests and the evaluation of our proposed techniques. In Sect. 6, we conclude the paper.

2 Previous work

The past few years have witnessed an exponential growth in the use of smartphones and tablets to gain wireless access to the Internet. This growth has been accompanied by a similar growth in cyber attacks over wireless links to steal session cookies and hijack private users' accounts. For example, the Android application FaceNiff [5] makes it easy to hijack other people's sessions using an Android smartphone with root access. The application is claimed to work over any private Wi-Fi network using any of the common protocols, including WEP, WPA-PSK, WPA2-PSK or no security at all. FaceNiff can be used to intercept web session profiles and easily hijack sessions for Facebook, Twitter, YouTube, Amazon, MySpace, Nasza-Klasa, blogger, etc. The only exception that disables FaceNiff is when the session is protected by EAP or SSL. In a similar vein, Firesheep [6], an extension for Firefox, clearly revealed that many sites fail to protect users against session hijacking attacks. To alert users of these vulnerabilities, a Firefox plug-in extension has been developed in [7] which notifies users if the server they are visiting is susceptible to cookie hijacking. This extension only gives warning for users to avoid risky web sessions but does not provide protection.

Several solutions have been proposed to address the problem of cookie hijacking in wired and wireless networks. Liu et al. [8] proposed a secure cookie protocol for ensuring integrity of each transmitted cookie by applying HMAC on the concatenation {username | expiration time | data | session key}. Their fundamental assumption, however, is that the secure cookie protocol would either encrypt the data using a session key or run on top of SSL.

Alternatively, a possible candidate to address this problem is the Lamport's well-known one-way hash chain technique for one-time passwords [9] which has been used in many authentication protocols. For example, the attack-resilient security architecture ARSA [10], proposed for multihop wireless mesh networks, uses a hierarchical one-way hash chain to authenticate beacons transmitted by mesh routers. The Ariadne secure on-demand Adhoc network routing protocol [11] uses hash chains in its Route Discovery phase and in later phases to thwart the effects of routing misbehavior. The SEAD protocol [12] proposed for Adhoc networks also uses a one-way hash chain for authenticating important routing information such as the routing metric and the sequence number. This prevents any malicious node from falsely advertising a better route or tampering with the critical routing information contained in the packet that it received from the source. Our main motivation for using one-way hash chain as a technique to secure session cookies is their cryptographic strength. Our objective is to propose a scheme which not only preserves this feature (i.e., cryptographic strength), but also uses it in a computationally efficient manner. Unlike HMAC-based schemes discussed above, using the one-way hash chains to secure cookies minimizes the chances of them being sniffed out and abused for unlawful utilization by entities other than the respective parties.

The one-way hash chain technique has been recently used to protect against cookie hijacking in wireless networks. The one-time cookies (OTC) protocol proposed in [13] is a straightforward implementation of Lamport's hash chain technique for one-time passwords. The authors implemented OTC as a plug-in for Firefox and Firefox for mobile browsers. OTC uses a hash chain construction to generate a sequence of values that can be used as one-time authentication tokens. These tokens, once verified by the web application, cannot be reused due to the pre-image resistance property of cryptographic hash functions. The rolling code protocol proposed in [14] is an attempt to reduce the computational overhead of the OTC protocol for mobile devices with constrained memory. The protocol replaces the hash chain performed by OTC in each transaction by two hash operations: one to update and randomize the value of a variable $d = \text{hash}(d)$ 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. In essence, the protocol is much like the rolling code technology used to prevent perpetrators from recording a code and replaying it to open a garage door. 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. Although the seed is guaranteed to be fresh during each iteration due to

the monotonic function used to increase the value of d, there is a high risk of discovering this value and consequently compromising the Internet session.

It should also be noted that ever since its inception, the Lamport's well-known one-way hash chain technique for one-time passwords has been used in many authentication protocols in a variety of environments including wireless sensor networks [15–18], smart card-based authentication schemes [19,20] and banking authentication schemes [21,22].

In this paper, we propose using a sparse caching approach to attain the full level of security of one-way hash chains but at a much reduced computational overhead. An earlier proposal for sparse caching was given by Gupta et al. [23] as a way to reduce directory memory requirements. The hallmark of this approach is that a memory block is allocated for each active entry, and invalidated data are discarded as they are no longer needed. Because of this feature, sparse caching techniques have been appealing in a variety of environments and applications. The authors of [24], for instance, proposed a method to improve users' perceived performance in wireless networks using sparse infrastructure. Also, the authors of [25,26] suggested using sparse caching in multimedia applications. As we describe our proposed solution, we also introduce several cache spacing configurations that are deployable in different scenarios.

3 The one-way hash chain model

We first present the notations for the one-way hash chain technique that we will use in this paper. We will refer to this **ha**sh **ch**ain technique as the HACH technique, which can be described as follows:

HACH uses an initial secret s which is the seed of the hash chain. We apply a cryptographic hash function $H()$ successively to obtain the following values

$$V_0 = s$$
$$V_1 = H(V_0) = H^1(s)$$
$$V_2 = H(V_1) = H^2(s)$$
$$\dots$$
$$V_j = H(V_{j-1}) = H^j(s)$$
$$\dots$$
$$V_N = H(V_{N-1}) = H^N(s)$$

Due to the pre-image resistance property of the hash function (e.g., SHA-1), the values V_i are distinct and can therefore be used to represent one-time authentication tokens in successive user transactions after the initial login, i.e., the value of the appropriate V token is stored in the session cookie transmitted by the client to the server in each transaction. To properly use the one-way hash chain, these values must be transmitted in reverse order. In the first transaction after login, the client browser transmits the value V_N. Similarly, the client transmits V_{N-1} in the second transaction and V_{N-i+1} in the ith transaction.

During the initial HTTPS authentication, a shared secret value s and a value chain length N representing the number of transactions are exchanged between the client and the server. After the initial login, the costly HTTPS protocol is replaced by HTTP and authentication is done by sending the one-time authentication tokens in the session cookies. Upon receiving the value of an authentication token, the server computes a similar value based on the values of N and s and accepts the transaction if the received value matches the computed value.

The variable N is the chain length and represents an upper bound on the number of transactions that can be handled by the above hash chain. One difficulty with the HACH technique is that the efficient use of the chain requires an accurate estimation of the value of N, i.e., the number of transactions expected during the lifetime of the session. If the number of transactions is overestimated, the authentication in the early steps will suffer from an unjustified large computational overhead. If the number of transactions is underestimated, there will be the undesirable synchronization overhead of establishing a new secret and a new number for the remaining transactions. For our initial presentation of the sparse caching technique, we will assume that the number of transactions in a session, N, is accurately known. Later in the paper, we will show how the sparse caching technique can be used to effectively deal with the case when the value of N is not accurately known.

We will use the number of hash operations executed per transaction as the metric to evaluate the execution overhead of the process of verifying this transaction using the authentication token transmitted by the client. We denote this metric for the ith transaction by $HCost_i$. Smaller values of this metric indicate faster speeds for the authentication process. The total cost of the entire session is the sum of the costs of the N transactions and is denoted by *TotalHCost*.

Assuming no caching, the first transaction computes $H^N(s)$ and has a cost $HCost_1 = N$. In general, the ith transaction computes $H^{N-i+1}(s)$ and has a cost $HCost_i = N - i + 1$. We have

$$Total\ Hcost = \sum_{i=1}^{N} (N - i + 1) = \frac{N(N+1)}{2}$$

The maximum and average value of the transaction execution overhead are given by

$$HCost_{\max} = N$$
$$HCost_{\text{avg}} = (N+1)/2$$

It is important to stress that the authentication tokens are generated by successive hashing from V_0 to X_N but are exposed (i.e., presented as authentication tokens) from V_N to V_0. The

reverse presentation of the values of the hash sequence is a fundamental feature that must be enforced in order to attain the good security properties of one-way hash chains. Exposing (presenting) the hash sequence values in the same forward order in which they are generated makes the scheme much less secure because if one hash value is compromised, the attacker can compute all the hash values that will be used in the future.

4 Sparse caching for HACH

4.1 Basic idea

The main drawback of the HACH protocol is the need to perform the hashing operation N times for the first transaction, $N-1$ times for the second transaction, and so on. For large values of N, the HACH technique is costly and is not generally suitable for many limited resource mobile devices. To remedy this problem using little storage, we compute a small number of the authentication tokens and cache their values during the initial setup. For example, if $N = 100$. We can reserve five storage units (of length 160 bits for SHA-1) at the initial setup to store the following values (note: cache[0] is included for ease of indexing)

$$\begin{aligned} &\text{cache}[0] = s \\ &\text{cache}[1] = H^{20}(s), \\ &\text{cache}[2] = H^{40}(s), \\ &\text{cache}[3] = H^{60}(s), \\ &\text{cache}[4] = H^{80}(s). \end{aligned}$$

The above values can be computed using a simple loop that computes the hash function 80 times. The availability of the above cache values reduces the execution overhead of transactions considerably. For example, the first transaction can now start by fetching the value of cache[4] then performing 20 more hashes to obtain $H^{20}(H^{80}(s)) = H^{100}(s)$. The value $HCost_1$ for the first transaction has been reduced from 100 to 20. Basically, the sparse caching scheme has divided the initial hash chain into five minichains, each with length 20.

The above example uses sparse caching with uniform (equal) spacing. We will examine non-uniform spacing later, but we will first formally define the uniform spacing model and examine its characteristics.

4.2 Sparse caching with uniform spacing

This scheme is defined by two parameters: the size of the cache, *cache_size*, and the uniform space interval, *minichain_len*. In the above example, $cache_size = 5$ and $minichain_len = 20$.

The pseudocode to compute the authentication token of the ith transaction is presented below. Notice that in the ith transaction, the client transmits the value $V_{N-i+1} = H^{N-i+1}(s)$.

Authentiaction_Token(i);

$$\begin{aligned} &k = (N-i)/minichain_len \quad //integerdivision \\ &m = N - (k \times minichain_len) \\ &V_{N-i+1} = H^m(\text{cache}[k]) \end{aligned}$$

Using the uniform sparse caching scheme, the total cost of a session including the cost of the initial cache setup is given by

$$TotalHCost = C \times \frac{M(M+1)}{2} = \frac{N(M+1)}{2}$$

where $C = cache_size$, $M = minichain_len$, and $N = C \times M$. The above formula is derived under the simplifying assumption that N is divisible by C. The cost *Total HCost* is the sum of two components: a cost of $(C-1) \times M$ for the initial filling of the cache values and a cost of $(C-1) \times M \times (M+1)/2$ for the N transactions. Notice that $(C-1)$ of the N transactions will not need to perform any hashing since the required value is already in the cache. For the above example of $N = 100$ and $C = 5$, transaction #21 will simply read V_{80} from cache[4]. The maximum and average value of the transaction execution overhead are given by

$$\begin{aligned} &HCost_{\max} = M \quad // \text{ for the 1st transaction} \\ &HCost_{\text{avg}} = 0.5 \times (C-1) \times (M-1)/C \end{aligned}$$

In Sect. 5, we evaluate the trade-offs of the sparse caching scheme and examine policies for selecting the cache size C for different values of N. It is important to notice that the sparse caching scheme is applied only to the user mobile device, not to the server. The server can compute the same authentication tokens using any caching scheme; for example, it may use a complete caching scheme that stores all authentication tokens in the cache during initialization.

4.3 Weighted Overhead formula

For a session with N transactions and a user with a certain mobile device, what would be the best *cache_size* C? In answering this question, we first recognize that wireless users can use mobile devices having a wide range of capabilities. Some users may use high-end laptops that have plenty of storage resources, while others may use mobile phones with limited memory. We use the following WO formula to obtain insight into the suitable value of the *cache_size* C.

$WO = w \times cache_size + Total\ HCost$

where w is the weight assigned to the cost of using memory in the mobile device. The WO formula is simply a pragmatic approach to select the size of the cache for the different categories of mobile devices. The WO obtained from the formula can be viewed as the combined cost of memory and execution overhead where w is the cost of memory relative to a unit cost of execution. The value of w may be assigned based on classes. For example, we may have three classes of mobile devices with the following weights.

Class 1: w_1 is used for high-end laptops with plenty of memory.

Class 2: w_2 is used for mobile devices, e.g., high-end smartphones and tablets, with reasonable but constrained memory resources.

Class 3: w_3 is used for mobile devices, e.g., low-end mobile phones, with very limited memory resources.

We have the obvious relationship $w_3 > w_2 > w_1$. To choose the best cache size for a mobile device, we simply minimize the value of the WO. For Class 1, the value of w_1 is nearly zero and the minimization problem reduces to minimizing the second term *TotalHCost*. Minimizing *TotalHCost* is achieved by choosing the largest possible value of cache size, which is simply N. This means that for high-end laptops, the sparse caching scheme is replaced by complete caching in which the authentication token value V_j is obtained by fetching cache[j] without performing any hash computation. For Classes 2 and 3, plotting the value of WO versus *cache_size* could reveal the best size that minimizes the combined cost.

In Sect. 5.4, we present performance results that show how the WO formula can be used to gain insight into selecting the size of cache.

4.4 Uncertainty in the number of transactions

In all previous discussions, we assumed that the number of transactions in a session, N, is accurately known. In real-life applications, however, the value of N is usually only approximately known. If the application developer chooses a value for N that is too small, the hash chain will be exhausted before finishing all of the transactions of the session. If the selected value of N is too large, the transactions will have large execution overhead. It is difficult for an application developer to consistently strike a good balance of N for the different sessions of the different users. The sparse caching approach presented earlier comes into play to elegantly solve this problem. Below, we elaborate on this issue.

Suppose that we know that a session will have approximately 1,000 transactions, but there is some likelihood that it could have up to 2,000 transactions. Without any caching, the developer will be tempted to choose a value of N between 1,000 and 2,000 to strike a balance between reducing *HCost* for the individual transactions and avoiding the scenario of exhausting the chain and resorting to costly HTTPS authentication and additional hash chain setup. By placing a single cached value cache[1] $= H^{1,000}(s)$ at the middle of the range, the developer can safely select $N = 2,000$ with a guarantee that $HCost_{\max}$ will be 1,000 and the costly HTTPS re-initialization will not be needed. By placing a second cached value at 2,000, the developer can extend the value of N to 3,000 with the same guarantee that $HCost_{\max}$ will be 1,000 and the costly HTTPS initialization will not be needed. The sparse caching scheme can be used with few cached units to extend the range of N to a large safe value without incurring an increase in the execution overhead of individual transactions or running the risk of additional HTTPS setup.

In Sect. 5, we present test results that illustrate the application of sparse caching for the case of approximate values of N.

4.5 Sparse caching with non-uniform spacing

The case of uncertain values of N motivates the use of sparse caching with non-uniform distribution. We elaborate on this by an example.

Suppose it is highly probable that the number of transactions in a user session will be 100 or less, but there is some small probability that this number could go up to 500. As shown earlier, we could use the sparse caching scheme to set the value of N to 500 without increasing the value of $HCost_{\max}$ above 100. We can actually do better than this by choosing non-uniform cache spacing to significantly improve the execution overhead of the first likely 100 transactions. For example, we can distribute 9 cache values non-uniformly as follows:

cache[0] $= s$
cache[1] $= H^{100}(s)$
cache[2] $= H^{200}(s)$
cache[3] $= H^{300}(s)$
cache[4] $= H^{400}(s)$
cache[5] $= H^{420}(s)$
cache[6] $= H^{440}(s)$
cache[7] $= H^{460}(s)$
cache[8] $= H^{480}(s)$

The above scheme gives priority to the first 100 transactions with a guaranteed value of $HCost_{\max} = 20$. The other less likely 400 transactions will be guaranteed a value of $HCost_{\max} = 100$.

Schemes for the forward generation and reverse presentation of one-way hash chains can use different topologies including tree topologies. But the simplicity of the proposed sparse caching authentication scheme and its flexibility in dealing with different scenarios (such as the non-uniform

spacing discussed in this section or the geometric spacing discussed in the next section) make the scheme more practically appealing than other schemes for implementing one-way hash chains [27].

In Sect. 5.5, we present the results of our tests to evaluate non-uniform spacing for sparse caching.

4.6 Caching with geometric spacing

In the previous section, the non-uniform spacing of cached values was achieved by creating two groups: the high-priority group (first 100 transactions in the above example) and the low-priority group (the remaining less likely 400 transactions). Within each group, the cached values are distributed uniformly. This scheme is suitable when the value of N is not accurately known but there is knowledge about the minimum value of N, i.e., the number of transactions that are most likely or are guaranteed to occur. If this knowledge is not available (i.e., the value of N could range from a small number to a large number), it would be better to distribute the cached values at progressively increasing intervals. One possible progressive strategy is the geometric distribution scheme. We illustrate this scheme using the previous example in which the value of N could be as small as 1 but could go up to 500. We use nine cached values geometrically distributed as follows:

$\text{cache}[0] = s$
$\text{cache}[1] = H^{246}(s)$
$\text{cache}[2] = H^{374}(s)$
$\text{cache}[3] = H^{438}(s)$
$\text{cache}[4] = H^{470}(s)$
$\text{cache}[5] = H^{486}(s)$
$\text{cache}[6] = H^{494}(s)$
$\text{cache}[7] = H^{498}(s)$
$\text{cache}[8] = H^{500}(s)$

The first transaction uses $\text{cache}[8] = H^{500}(s)$ and will not need to perform any hash operations. The second and third transactions use $\text{cache}[7] = H^{498}(s)$, resulting in performing one hash operation for the second transaction and no operation for the third transaction. The transactions numbered 4, 5, 6 and 7 use $\text{cache}[6] = H^{494}(s)$ resulting in performing 3, 2, 1 and 0 hash operations, respectively, for these four transactions. Notice that the cached values are anchored at points that are geometrically spaced apart. The difference between the number of hash operations performed for cache[8] and cache[7] is 2, between cache[7] and cache[6] is 4, between cache[6] and cache[5] is 8, between cache[5] and cache[4] is 16, and so on. As the number of transactions in the real session increases, the range of the number of hash operations will increase geometrically and high-numbered transactions will need to perform larger number of hash operations on average.

The result of the geometric distribution scheme is to give smaller $HCost_{\text{avg}}$ value for earlier transactions. Shorter sessions will benefit more from the geometric distribution.

In Sect. 5.7, we present the results of our tests to evaluate sparse caching with geometric spacing.

4.7 Energy consumption

When designing any authentication protocol for mobile devices, it is important to reduce the energy expended by this protocol. According to [28], 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. In designing our protocol, one of our major goals was to come up with a light security mechanism while ensuring the highest cryptographic strength available.

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 [29] 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, please refer to [29]). The level of energy consumption by our authentication protocol in a user session is correlated with the value of *TotalHashCost* described in Sect. 4.2.

In Sect. 5.8, we compare the energy consumption of our sparse caching protocol and compare it with the case of no caching. It should be noted though that the initialization phase is not included in this comparison because it is conducted using HTTPS.

5 Evaluation and performance results

To experiment with the sparse caching HACH scheme, we developed a benchmark which was written using Java. The benchmark fully implements the one-way hash chain model and the different sparse caching configurations.

In the tests and experiments to evaluate the performance of HACH, we considered several situations and different scenarios. Our objective was to simulate real-life connections which are characterized by differing needs as far as storage and performance are concerned. Therefore, we used the benchmark to evaluate the HACH performance with numbers of transactions having different ranges: from 1 to 200 for short sessions, from 500 to 2,500 for long sessions and up to 4,000 and 5,000 transactions in some tests. We also

varied the storage availability to make sure that we address different users' needs. The storage spaces used to evaluate performance ranged from 20 to 500 spaces.

5.1 Impact of cache size on HACH performance

Figure 1 shows the impact of cache size on the performance of HACH. The execution overhead of HACH is measured by *TotalHCost*, the total number of hash operations in a session including the initial cache setup. The spacing scheme used in Fig. 1 is the uniform sparse caching scheme described in Sect. 4.2. The results in Fig. 1 are obtained from the Java testbed, and they agree with the cost estimation derived in Sect. 4.2. The results clearly show that the more cache we have, the less the *TotalHCost* would be.

Figure 2 demonstrates the impact of cache size on the average value of the execution overhead of one transaction (i.e., $HCost_{\text{avg}}$). Again we see that the more cache we have, the less the value of $HCost_{\text{avg}}$. The sparse caching scheme speeds up transaction authentication and helps in reducing the turnaround time of user requests.

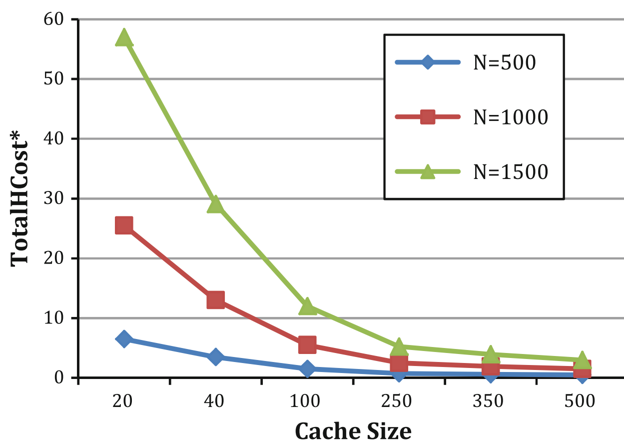

Fig. 1 Impact of cache size on *TotalHCost*. *In thousands

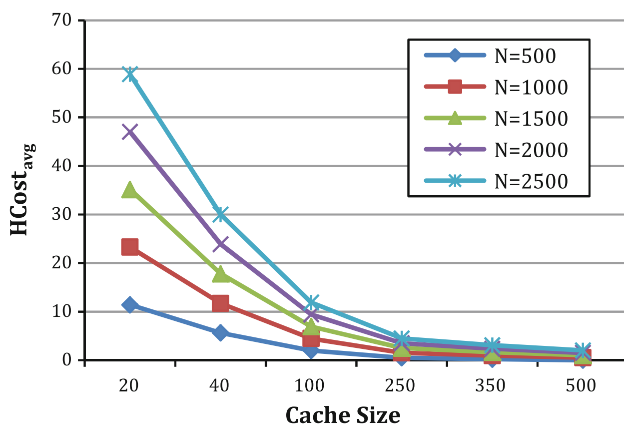

Fig. 2 Impact of cache size on $HCost_{\text{avg}}$

5.2 Cache space allocation policies

In Sect. 4.3, we proposed a WO formula and gave an example of three classes for mobile devices used in wireless networks. For Class 1 (high-end laptops with plenty of memory), we can afford to allocate large cache memory to get the best benefit of HACH. For Classes 2 and 3 (mobile phones with limited capability), the memory allocated to cache will be at a lesser level. To investigate the performance of sparse caching on the different mobile devices, we used the following two policies for allocating cache memory to a user session with N transactions:

1. The *square root (sqrt)* policy which allocates $cache_size = (N)^{0.5}$ for a session with N transactions.
2. The *logarithm* ($\log_2$) policy which allocates $cache_size = \log_2(N)$ for a session with N transactions.

We have tested a few other policies, but we select the above two policies for presentation in this paper because they nicely suit the classification of mobile devices described in Sect. 4.3. The $\log_2$ policy is an aggressive policy which reserves more memory and therefore can be used for Class 1 devices. The *sqrt* policy, on the other hand, exhibits a more conservative behavior and reserves smaller amount of cache memory, which is suitable for low-end mobile devices of Classes 2 and 3. Both policies can be multiplied by a scale factor (ranging from a small fraction to a large number) to adapt the rate of memory allocation based on the capability of the mobile device, e.g., to differentiate between Class 2 and Class 3.

Figure 3 shows the HACH execution overhead, and Fig. 4 shows the HACH storage overhead of the *sqrt* and $\log_2$ cache allocation policies. The $\log_2$ policy allocates more cache space and consequently incurs a much less *TotalHCost*. The *sqrt* policy allocates less cache space and incurs much higher *TotalHCost*.

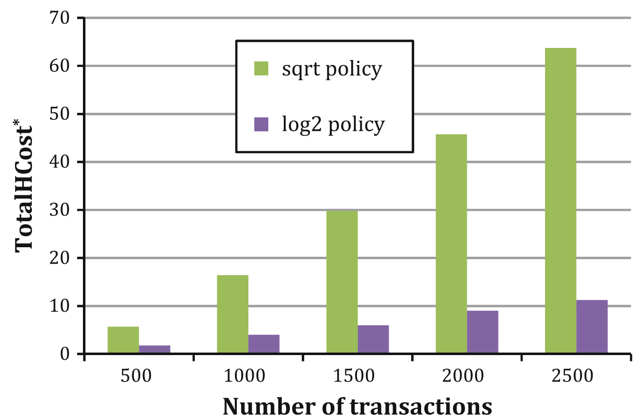

Fig. 3 HACH performance with two policies for sparse caching. *In thousands

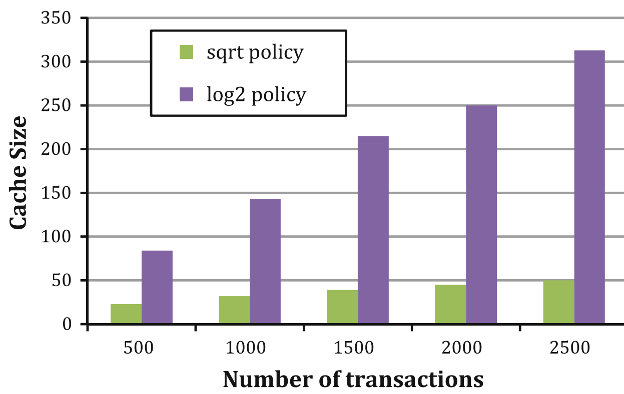

Fig. 4 HACH storage requirements with two policies for sparse caching

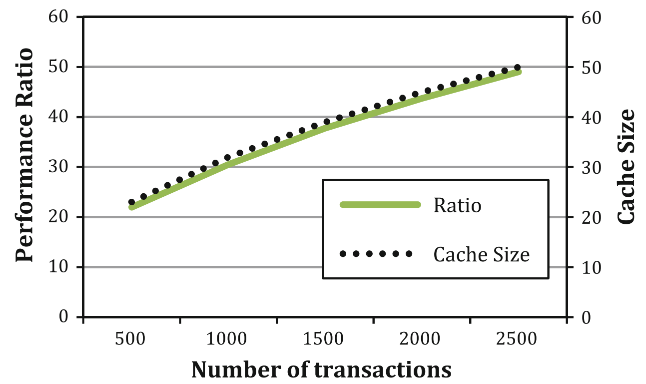

Fig. 5 HACH performance improvement ratio

Table 1 HACH performance with/without sparse caching using the *sqrt* policy

Number of transactions	*TotalHCost* no caching	*TotalHCost* sparse caching	Performance ratio
500	125,250	5,702	21.97
1,000	500,500	16,404	30.51
1,500	1,125,750	29,811	37.76
2,000	2,001,000	45,750	43.74
3,000	4,501,500	83,625	53.83
4,000	8,002,000	127,506	62.76

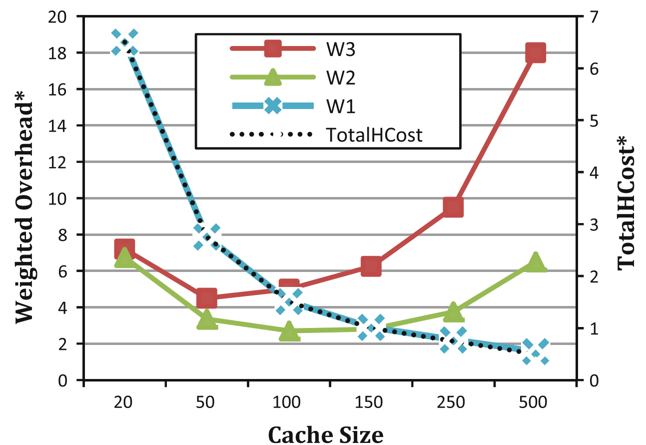

Fig. 6 Weighted Overhead results for $N = 500$. *In thousands

5.3 Effectiveness of sparse caching

In this section, we present the simulation results when we compared the performance of HACH with and without sparse caching. Table 1 summarizes the results we obtained from tests using the *square root (sqrt)* policy. The performance ratio in the last column is the ratio *TotalHCost with no caching/TotalHCost with sparse caching*. The sparse caching scheme significantly improves performance of HACH. Sparse caching is able to decrease *TotalHCost* in transactions by an average of approximately 41 times over the six values of N shown in Table 1.

In Fig. 5, we further illustrate the performance improvement ratio of the HACH using sparse caching. The x-axis represents the number of transactions, and the left y-axis represents the performance improvement ratio associated with every value of N. The right y-axis shows the storage requirement for each value of N. We notice that each added cache unit decreases the value of *TotalHCost*. This is intuitive since if we can afford more memory, we would definitely improve performance. However, in the case of mobile phones and other low-end mobile devices, the storage may not always be readily available and we need to find a compromise between performance and storage.

5.4 Selecting cache size for mobile devices

In order to find the best trade-off between cache size and performance (measured in *TotalHCost*), we introduced the WO formula

$$\text{WO} = w \times cache_size + Total\ HCost$$

In the simulation tests, we experimented with different values of w to represent the different classes. For Class 1 devices, the value of w was a small value close to zero because memory consumption is not a substantial issue with high-end devices. Higher values of w are used for Class 3 devices, and lower values are used for Class 2 devices. We ran tests for different values of N.

Figure 6 shows the best cache size when the number of transactions is 500. The left y-axis represents the value of the WO, and the right y-axis is the value of *TotalHCost*. The values of weights are $w1 = 0.1$ representing Class 1 devices, $w2 = 12$ representing Class 2 devices and $w3 = 35$ representing Class 3 devices. The dashed curve is used to indicate *TotalHCost*. We notice that for this particular number of transactions ($N = 500$), a cache size of 50 units is the most

suitable for Class 3 devices as it exhibits the lowest WO value which strikes an acceptable trade-off between *TotalHCost* and memory. For Class 2 devices, the best cache size is 100. We notice for Class 1 devices represented by $w1$, the WO curve is very close to the *TotalHCost* curve, indicating that we should select the highest possible cache size $C = N$, i.e., complete caching of size 500.

Figure 7 shows the same test but with $N = 1,000$. The same w values reported for Fig. 6 were used here. For Class 3 devices ($w3$ curve), the best cache size is slightly over 100. For Class 2 devices ($w2$ curve), the best size of cache is 250. For Class 1 devices ($w1$ curve), it is best to use complete caching of size 1,000.

We further analyzed the performance of sparse caching for HACH using a metric called the *speedup factor per unit cache* (or simply the *speedup factor*) defined as follows:

$$Speedup factor = H1/(H2 * C)$$

where

$H1$ = the value of *TotalHCost* without caching and

$H2$ = the value of *TotalHCost* with sparse caching of size C

Table 2 gives the results for the *speedup* factor for Class 3 (weight $w3 = 35$) when the cache size used is the best cache size selected by the WO formula. Table 2 shows that the contribution of one cache unit is captured by a *speedup* value of approximately 0.9. As an example for $N = 500$ transactions and cache size $C = 50$, each cache unit improves the performance by a magnitude of 0.91 and the total cache of size 50 increases performance by a magnitude of 45.5 resulting in decreasing the total number of hashes from 125,250 to 2,750.

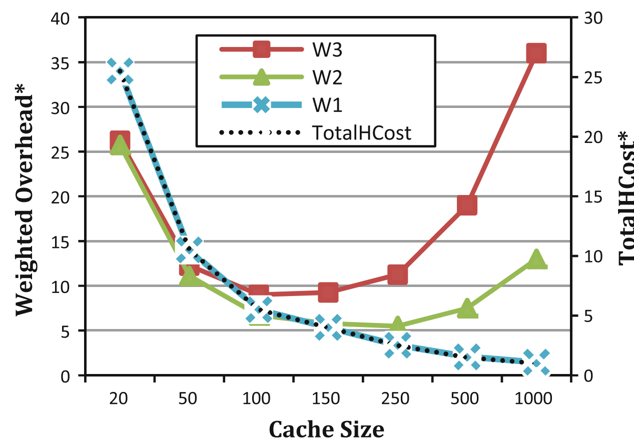

Fig. 7 Weighted Overhead results for $N = 1,000$. *In thousands

Table 2 *Speedup* factor per unit cache for Class 3

Number of transactions	Cache size	*H2*	*H1*	Speedup
500	50	2,750	125,250	0.91
1,000	100	5,500	500,500	0.91
1,500	150	8,250	1,125,750	0.91
2,000	250	9,000	2,001,000	0.89
3,000	250	18,000	4,501,500	1.00
4,000	500	18,000	8,002,000	0.89

Table 3 *Speedup* factor per unit cache for Class 2

Number of transactions	Cache size	*H2*	*H1*	Speedup
500	100	1,500	125,250	0.84
1,000	250	2,500	500,500	0.80
1,500	250	5,250	1,125,750	0.86
2,000	350	7,000	2,001,000	0.82
3,000	500	10,500	4,501,500	0.86
4,000	1,000	10,000	8,002,000	0.80

Table 4 *Speedup* factor per unit cache for Class 1

Number of transactions	Cache size	*H2*	*H1*	Speedup
500	500	500	125,250	0.50
1,000	1,000	1,000	500,500	0.50
1,500	1,500	1,500	1,125,750	0.50
2,000	2,000	2,000	2,001,000	0.50
3,000	3,000	3,000	4,501,500	0.50
4,000	4,000	4,000	8,002,000	0.50

Table 3 shows similar results for Class 2 devices ($w2 = 12$). The cache sizes used for these devices are bigger than those of Class 3. It should be mentioned that for values of N equal to 2,000 and higher, the weight $w2 = 12$ gave multiple best cache sizes (giving the same WO value). We therefore used a weight value $w2 = 15$ to pick the specific cache size shown in Table 3.

For Class 1 devices ($w1 \approx 0$), the WO formula suggests using complete caching. Table 4 gives the results for the *speedup* factor for Class 1 (weight $w1 \approx 0$) when the cache size used is equal to the value of N. The *speedup* factor for Class 1 with complete caching is 0.50.

5.5 HACH with non-uniform cache spacing

Tables 5 and 6 give comparisons between sparse caching with uniform spacing and with non-uniform spacing. For non-uniform spacing, we allocated 50 % of the cache for the first 20 % of transactions, which we call "high-priority transactions". The other 50 % of the cache is used to serve

Table 5 Average speedup for a high-priority transaction

Number of transactions	High-priority	Non-uniform spacing	Uniform spacing	Speedup
500	100	1.5	4.5	3.00
1,000	200	3.5	9.5	2.71
1,500	300	5.5	14.5	2.64
2,000	400	7.5	19.5	2.60
3,000	600	11.5	29.5	2.57
4,000	800	15.5	39.5	2.55

Table 6 Average slow-up of a low-priority transaction

Number of transactions	Low-priority N	Non-uniform spacing	Uniform spacing	*Slow-up*
500	400	7.5	4.5	1.67
1,000	800	15.5	9.5	1.63
1,500	1,200	23.5	14.5	1.62
2,000	1,600	31.5	19.5	1.62
3,000	2,400	47.5	29.5	1.61
4,000	3,200	63.5	39.5	1.61

the remaining 80 % transactions, which we call the low-priority (uncertain) transactions. Table 5 analyzes the average authentication time (average number of hash operations) of a high-priority transaction. The non-uniform spacing gives superior (much smaller) authentication time compared to the uniform spacing. The last column in Table 5 gives the value of *speedup* for a high priority transaction. For example when $N = 500$, the average authentication time for a high-priority transaction using *non-uniform spacing* is 1.5 hash operations, whereas the corresponding figure for *uniform spacing* is 4.5. This means that high-priority transactions enjoy a threefold speedup under the *non-uniform spacing* scheme compared to the *uniform spacing* scheme. As the number of transactions goes up, the *speedup* factor decreases slightly and becomes equal to 2.55 at $N = 4,000$.

Table 6 analyzes the average authentication time (average number of hash operations) of a low-priority transaction. The non-uniform spacing gives larger (slower) turnaround time compared to the uniform spacing. The last column in Table 6 gives the value of the *slow-up* for a low-priority transaction. Notice that the *uniform spacing* scheme gives the same speed for both high-priority transactions (Table 5) and low-priority transactions (Table 6). From Tables 5 and 6, we see that the non-uniform caching scheme has positively impacted high-priority transactions by a speedup factor of 2.5 or higher and has negatively impacted low-priority transactions by a slow-up factor of only 1.67 or less.

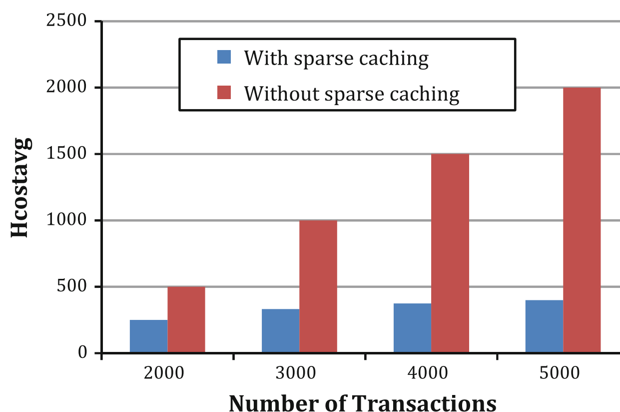

Fig. 8 Handling unknown large values of N: comparison between $HCost_{\text{avg}}$ with and without sparse caching

5.6 Approximate knowledge of N

In the previous sections, we performed our simulation tests assuming that the value of N is accurately known. The value of N in most real-life applications is not accurately known. A minimal level of sparse caching can help the developer apply hash chains of larger length without the fear of any increase in the average or maximum authentication time for a transaction. Figure 8 shows the impact of using sparse caching with *minichain_len* set at 1,000 for sessions with unknown number of transactions exceeding 1,000. It can be seen that sparse caching reduces $HCost_{\text{avg}}$ of transactions when N is not precisely known. For example, when $N = 4,000$ transactions, the value of $HCost_{\text{avg}}$ without sparse caching is approximately 1,500. By placing only three cache units, the value of $HCost_{\text{avg}}$ is reduced to 375. We observed that the reduction in $HCost_{\text{avg}}$ due to sparse caching increases as the number of transactions increases.

5.7 Geometric spacing

In Sect. 4.6, we introduced the idea of geometric spacing of cache which gives smaller $HCost_{\text{avg}}$ value for earlier transactions in cases when the exact number of transactions N is not known and could be as small as one and as large as N. The idea involves increasing the cache spacing intervals progressively as the real number of transactions increases.

Figure 9 shows the test results for the case when N is not known, and the developer has chosen the value $N = 500$ to be the length of the hash chain. The horizontal axis represents the real number of transactions, denoted K. The figure compares the performance of the following three schemes for values of $K = 5, 10, 20, 50, 100$ and 200.

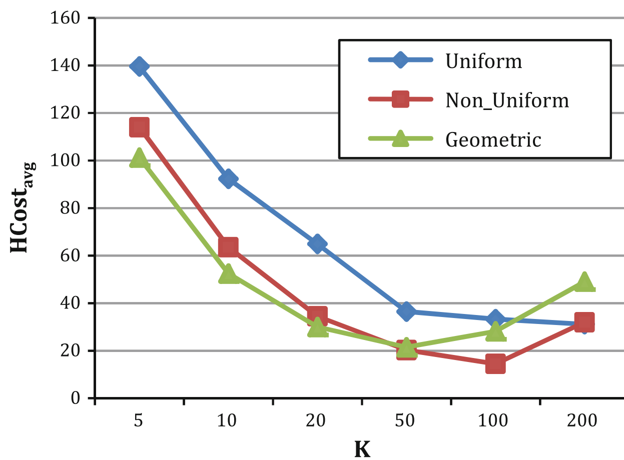

Fig. 9 Comparison of $HCost_{avg}$ between uniform, non-uniform and geometric spacing

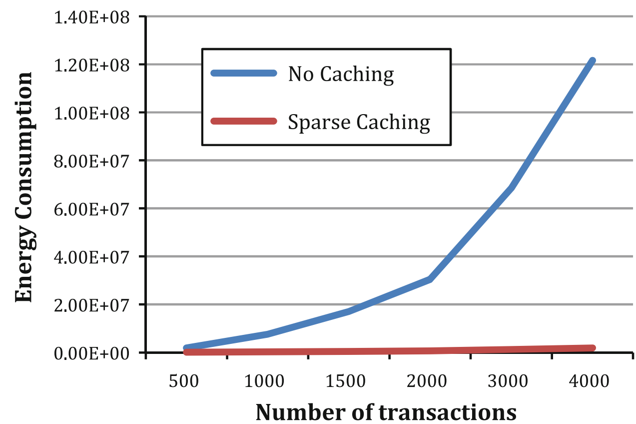

Fig. 10 Energy consumption comparison of HACH with and without sparse caching

- Uniform caching with equal spacing as described in Sect. 4.2.
- Non-uniform caching with two groups as described in Sect. 4.5.
- Geometric spacing as described in Sect. 4.6.

As seen in Fig. 9, using the geometric spacing policy improves the performance of HACH compared to the other two policies when the real number of transactions K is lower than 50. For the case $K = 50$, the geometric spacing performs better than uniform spacing and has the same performance as the non-uniform spacing policy. For $K = 100$ transactions, geometric spacing still performs better than uniform spacing but worse than non-uniform spacing. The geometric spacing scheme does not benefit HACH when $K \geq 120$ transactions.

5.8 Energy consumption

In Sect. 4.7, we introduced the energy consumption metric used to evaluate the performance of our sparse caching protocol. Since the energy consumption is largely influenced by the cryptographic hash function used, the type and amount of hashing operations required to carry out the authentication of the internet session translate into the energy consumption of the authentication scheme.

Figure 10 illustrates the energy consumption comparison between our sparse caching-based HACH protocol and its counterpart without sparse caching. It should be noted that the sparse caching in this comparison is conducted using the *sqrt* policy and SHA-1 hashing. As can be seen in the figure, our sparse caching scheme tremendously improves energy consumption of the HACH authentication protocol. It is also noted that the HACH protocol suffers from a huge increase in energy consumption making it far from ideal especially for platforms with limited energy capacities.

6 Conclusion

In this paper, we have shown that the use of lightweight easy to implement sparse caching approach can significantly improve the performance of the widely used cryptographic one-way hash chain technique to secure session cookies in terms of computational overhead and energy consumption. We introduced a WO formula to help select a best cache size depending on different users' storage requirements. Different cache spacing techniques have been investigated to demonstrate different connection behaviors.

We presented the results of extensive performance tests that have shown the significant reduction in authentication cost achieved by the sparse caching schemes. We have also shown how to deal with real-life situations in which the number of transactions per user session is largely unknown and cannot be accurately estimated.

References

1. Chen, J., Jiang, M., Liu, Y.: Wireless LAN security and IEEE 802.11i. IEEE Wirel. Commun. **12**(1), 27–36 (2005)
2. Sreedhar, C., Madhusudhana, S., Kasiviswanath, N.: A survey on security issues in wireless ad hoc network routing protocols. Int. J. Comp. Sci. Eng. **12**(2), 224–232 (2010)
3. Siddiqui, M., Hong, C.: Security issues in wireless mesh networks. In: Proceedings of IEEE International Conference on Multimedia and Ubiquitous Engineering (MUE'07). Seoul, Korea (2007)
4. Zhou, Y., Fang, Y., Zhang, Y.: Securing wireless sensor networks: a survey. IEEE Commun. Surv. **10**(3), 6–28 (2008)
5. Ponurkiewicz, B.: FaceNiff—A new Android download application. http://faceniff.ponury.net/. Accessed 26 Jan 2012
6. Butler, E.: FireSheep: cookie snatching made simple. In: ToorCon Conference. San Diego, CA (2010). Software available at http://codebutler.com/firesheep
7. Riley, R., Ali, N., Al-Senaidi, K., Al-Kuwari, A.: Empowering users against sidejacking attacks. In: Proceedings of the ACM SIGCOMM Conference on SIGCOMM. New Delhi, India (2010)

8. Liu, A., Kovacs, J., Huang, C., Gouda, M.: A secure cookie protocol. In: Proceedings of 14th International Conference on Computer Communications and Networks (2005)
9. Lamport, L.: Password authentication with insecure communication. Commun. ACM **24**(11), 770–772 (1981)
10. Zhang, Y., Fang, Y.: ARSA: an attack-resilient security architecture for multihop wireless mesh networks. IEEE J. Sel. Areas Commun. **24**(10), 1916–1928 (2006)
11. Hu, Y., Perrig, A., Johnson, D.: Ariadne: a secure on-demand routing protocol for ad hoc networks. Wirel. Netw. **11**(1–2), 21–38 (2005)
12. Hu, Y., Johnson, D., Perrig, A.: SEAD: secure efficient distance vector routing for mobile wireless ad hoc networks. In: Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2002), pp. 3–13. Calicoon, NY (2002)
13. Dacosta, I., Chakradeo, S., Ahamad, M., Traynor, P.: One-time cookies: preventing session hijacking attacks with disposable credentials. Technical Report Georgia Institute of Technology (2011). http://smartech.gatech.edu/bitstream/handle/1853/37000/GT-CS-11-04.pdf
14. Cashion, J., Bassiouni, M.: 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), pp. 21–26. Miami Beach, Florida (2011)
15. Liu, D., Ning, P.: Multilevel μTESLA: broadcast authentication for distributed sensor networks. Trans. Embed. Comput. Syst. (TECS) **3**(40) (2004)
16. Tan, H., Jha, S., Ostry, D., Zic, J., Sivaraman, V.: Secure multihop network programming with multiple one-way key chains. In: Proceedings of the First ACM Conference on Wireless Network Security-WiSec '08 (2008)
17. Khalil, I., Bagchi, S., Rotaru, C.N., Shroff, N.B.: UnMask: utilizing neighbor monitoring for attack mitigation in multihop wireless sensor networks. Ad Hoc Netw. **8**(2), 148–164 (2010)
18. Li, M., Yu, S., Guttman, J.D., Lou, W., Ren, K.: Secure ad hoc trust initialization and key management in wireless body area networks. ACM Trans. Sens. Netw. (TOSN) **9**(2), 18 (2013)
19. Chen, T.H., Hsiang, H.C., Shih, W.K.: Security enhancement on an improvement on two remote user authentication schemes using smart cards. Future Gener. Comput. Syst. **27**(4), 377–380 (2011)
20. Li, C.T., Hwang, M.S.: An efficient biometrics-based remote user authentication scheme using smart cards. J Netw. Comput. Appl. **33**(1), 1–5 (2010)
21. Dai, X., Grundy, J.: NetPay: an off-line, decentralized micropayment system for thin-client applications. Electron. Commer. Res. Appl. **6**(1), 91–101 (2007)
22. Liaw, H., Lin, J., Wu, W.: A new electronic traveler's check scheme based on one-way hash function. Electron. Commer. Res. Appl. **6**(4), 499–508 (2008)
23. Gupta, A., Weber, W., Mowry, T.: Reducing Memory and Traffic Requirements for Scalable Directory-based Cache Coherence Schemes. Springer, NY (1992)
24. Deftu, A., Murarasu, A.: Optimization techniques for dimensionally truncated sparse grids on heterogeneous systems. In: Proceedings of the 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 351–358 (2013)
25. Lau, W., Kumar, M., Venkatesh, S.: A cooperative cache architecture in support of caching multimedia objects in MANETs. In: Proceedings of the 5th ACM International Workshop on Wireless Mobile Multimedia, pp. 56–63 (2002)
26. Douglas, C. C., Hu, J., Iskandarani, M., Kowarschik, M., Rüde, U., Weiss, C.: Maximizing cache memory usage for multigrid algorithms. In: Chen, Z., et al. (eds.) Multiphase Flows and Transport in Porous Media: State of the Art. Lecture Notes in Physics, vol. 552, pp. 124–137. Springer, Berlin (2000)
27. Hu, Y., Jakobsson, M., Perrig, A.: Efficient constructions for one-way hash chains. In: Applied Cryptography and Network Security. Lecture Notes in Computer Science, vol. 3531, pp. 423–441. Springer, Berlin (2005)
28. Chandramouli, R., Bapatla, S., Subbalakshmi, K., Uma, R.: Battery power-aware encryption. ACM Trans. Inf. Syst. Secur. (TISSEC) **9**(2), 162–180 (2006)
29. Potlapally, N., Ravi, S., Raghunathan, A., Jha, N.: Analyzing the energy consumption of security protocols. In: Proceedings of the 2003 International Symposium on Low Power Electronics and Design, pp. 30–35 (2003)