Strengthening Password-Based Authentication Protocols Against Online Dictionary Attacks*

Peng Wang[1], Yongdae Kim[1], Vishal Kher[1], and Taekyoung Kwon[2]

[1] Computer Science and Engineering,
University of Minnesota - Twin Cities, Minnesota, USA
{pwang,kyd,vkher}@cs.umn.edu
[2] School of Computer Engineering, Sejong University, Seoul, Korea
tkwon@sejong.ac.kr

Abstract. Passwords are one of the most common cause of system break-ins, because the low entropy of passwords makes systems vulnerable to brute force guessing attacks (dictionary attacks). Existing Strong Password-based Authentication and Key Agreement (SPAKA) protocols protect passwords from passive (eavesdropping-offline dictionary) attacks, but not from active online dictionary attacks. This paper presents a simple scheme that strengthens password-based authentication protocols and helps prevent online dictionary attacks as well as many-to-many attacks common to 3-pass SPAKA protocols. The proposed scheme significantly increases the computational burden of an attacker trying to launch online dictionary attacks, while imposing negligible load on the legitimate clients as well as on the authentication server.

1 Introduction

Password-based authentication protocols cannot rely on persistent stored information on the client side. Instead, they rely on users' ability of *precise recall* of a secret information. It is mainly due to this precise recall requirement that users typically choose simple and low entropy passwords that are easy to remember [7, 15, 16, 21, 24]. The weakness of passwords becomes the weak link of the system, which attackers exploit by launching *offline* or *online* dictionary attacks. In online dictionary attacks, the attacker tries to guess the correct password by interacting with the login server. In offline dictionary attacks, the attacker first collects messages between the users and the server or finds a copy of the password file. Then, the attacker tries to guess correct passwords by matching the passwords in her dictionary with the collected information without requiring any feedback from the login server.

Typically, online dictionary attacks are prevented by using *account locking* or *delayed response* techniques. In account locking a server locks accounts after few unsuccessful attempts. However, account locking enables denial of service attacks against users' accounts and may increase administrators' load if the locks have

* This research is supported in part by the Intelligent Storage Consortium at Digital Technology Center (DISC), University of Minnesota.

J. Ioannidis, A. Keromytis, and M.Yung (Eds.): ACNS 2005, LNCS 3531, pp. 17–32, 2005.

to be opened manually. Delayed response aims to reduce the number of passwords attackers can check in a period of time. However, if the attacker wants to compromise any account in the target system, she can initiate many sessions simultaneously (parallel attacks) and can still check (using different usernames) a large number of passwords in her dictionary. For example, attacker can marshal the muscle of a few thousand computers and perform online dictionary attacks. Recent trends indicate that hackers are renting out vast networks of infected home computers (zombies) without their owner's knowledge [30, 31]. An attacker can rent a network of 20,000 computers for less than a few thousand dollars ($2,000). If the attacker aims to perform online dictionary attacks against highly sensitive networks, such as the military networks, the attacker has enough incentives to buy such network of zombies. Using these networks the attacker can launch parallel online dictionary attacks and verify large number of guesses within a short period of time. We stress that employing account locking and delayed response techniques in such scenarios does not help.

In general, strong password-based authentication protocols should not reveal any useful information about the users' passwords to the login server (which can be malicious) and should not be susceptible to online dictionary attacks and eavesdropping attacks. The goal of the proposed work is to strengthen existing password based authentication protocols against online dictionary attacks. *Strong Password-based Authentication and Key Agreement (SPAKA) protocols* [1, 12] are remote password only protocols that can provide authentication and key agreement over insecure channel without the support of previously shared cryptographic keys or a Public Key Infrastructure (PKI). Other authentication protocols, such as SSH [29] and protocols running on SSL [25] are vulnerable to man-in-the-middle attacks (since public key certificates are rarely checked) and these protocols forward the password (or some simple function of the password such as hash) to the server. SPAKA protocols are vulnerable to online dictionary attacks, but they do not reveal any secret information to the login server. Further, they are not susceptible to eavesdropping and man-in-the-middle attacks. Therefore, we choose to strengthen SPAKA protocols against online dictionary attacks, and, as a result, complete the general set of security requirements of a strong password based authentication protocols. However, the scheme presented in this paper is generic enough to be integrated with other password based authentication protocols.

In addition to online dictionary attacks, the 3-pass SPAKA protocols are vulnerable to a more powerful many-to-many guessing attack [18]. In 3-pass SPAKA protocols, when the client initiates the login protocol, the server sends out only one message (during the second pass) that contains both the server's challenge and its proof of the knowledge of the verifier. In many-to-many guessing attacks, an attacker can collect these values and terminate the protocol at the end of the second pass. The attacker can then use these values to mount guessing attacks offline. She can initiate multiple of such half-open sessions and gather a lot of information before the server detects the attack. Thus, the attacker can verify more number of guesses than that allowed by the server's policy. In this paper

we present a scheme that strengthens existing authentication protocols against online dictionary attacks. We integrate our scheme with SPAKA protocols to strengthen SPAKA protocols against online dictionary attacks as well as many-to-many guessing attacks. We call the modified SPAKA protocols as SPAKA+.
Overview of Our approach There is a fundamental difference between the login attempts performed by the legitimate users and the login attempts performed by the attackers trying to launch online dictionary attacks. A user who knows the password can successfully login within a couple of trials, while an attacker is expected to perform several magnitudes more trials than legitimate users do. In general, one of the main factors that limit the success of an attacker attempting to launch dictionary attacks is the amount of time required by a program (password cracker) to guess a user's password. The threat of parallel attacks can be eliminated by requiring the client to send a "proof of work" with an aim to keep the attacker busy and reduce the number of sessions that an attacker can initiate.

SPAKA+ strengthens SPAKA protocols against online dictionary attacks and many-to-many attacks by asking clients to solve a cryptographic puzzle (proof of work). The scheme is designed to distinguish between legitimate users and attackers and puts negligible computational burden on the legitimate users. Attackers are forced to solve puzzles, which increases the complexity of online dictionary attack approximately by the hardness of puzzles. If under attack, the authentication server can self-adjust the hardness of the puzzle. Therefore, our protocol will impose significant computation burden on sophisticated attackers using rented zombies, and, thus, greatly increase the amount of time required to break passwords. The computational burden on the authentication server is negligible and the server has to maintain only one long term state information (near-stateless) if users' computers are assumed to be secure. In case users' computers are not secure, we suggest a way to minimize the success of the attacker by maintaining some state information on the server. Our generic puzzle-based scheme can be generalized to non-SPAKA protocols, such as the authentication protocols used with SSL or SSH as long as the basic protocols generate shared secrets (e.g., session keys) between the client and the server. We use these secrets to "mark" the computers used by legitimate users.

Organization The rest of the paper is organized as follows. Section 2 introduces SPAKA. We present our protocols in section 3. Section 4 discusses security and performance issues. Section 5 reviews related work, and section 6 concludes the paper and outlines future work.

2 SPAKA Protocols

Since Lomas *et al.* introduced LGSN in 1989 [19], there have been considerable research efforts on Strong Password-based Authentication and Key Agreement (SPAKA) protocols, such as EKE [5], SPEKE [13], SRP [27], AMP [17], AuthA [4], PAK[6], etc, (refer [1] and [12] for a complete list of papers). SPAKA protocols are remote password-only protocols. They can provide authentication and key agreement over insecure channel without the support of previously shared cryptographic keys or a Public Key Infrastructure (PKI). In a SPAKA protocol, a party only commits high entropy information to the other party and never shows any information except the fact of knowing the password or the verifier of the password. Since messages transferred over the network do not leak information about passwords, attackers cannot launch offline dictionary attacks based on the eavesdropped massages.

SPAKA protocols typically have two stages: a key agreement phase that ends with two parties sharing a common secret that can be used to generate the shared session key, and a key confirmation phase in which two parties verify that they share the common key so that they can believe they are talking to the right party. At the end of a successful run of this protocol, each party holds a session key for subsequent secure communications. We list security properties of SPAKA protocols below.

- SPAKA protocols provide mutual authentication.
- They are secure against offline dictionary attacks.
- They are secure against *Denning-Sacco attack* [10]. Learning already distributed session keys will not help the attacker to discover passwords, verifiers or new session keys.
- They provide *perfect forward secrecy.* Learning the password and (or) the verifier will not help the attacker to discover previous session keys.
- They do not require clock synchronization between the client and the server.

3 SPAKA+

SPAKA Protocols protect passwords from eavesdropping-offline dictionary attacks. However, online dictionary attacks are still possible. Delayed response also failed because attackers can initiate parallel attacks. We eliminate the threat by requiring the client to solve a puzzle with an aim to keep the attacker busy and reduce the number of sessions that an attacker can initiate. Ideally, only attackers should be asked to solve puzzles. Since a user typically uses a limited set of computers that are not accessible to attackers, legitimate users can be distinguished from attackers based on the origin of the login request. SPAKA+ uses successful authentication sessions (old session keys) to "mark" computers of legitimate users. The following table lists the notations we will use in the rest of this paper.

Symbol	Meaning	Symbol	Meaning
C	Client's ID	S	Server's ID
π	Client's password	q	System parameter of SPAKA protocols
v	Client's verifier	$f_i(\cdot)$	Functions used in SPAKA protocols, $i \in [1, 6]$
IP_c	Client's IP address	$h^{-1}(\cdot)$	Procedure used to solve the puzzle
$h(\cdot)$	Hash function	k_1, k_2, k_1', k_2'	Temporary values of SPAKA protocols
N	Output size of $h(\cdot)$	k_p, k_q, k_p', k_q'	Temporary values of SPAKA+
x, y	Random numbers	$E_k(\cdot)$	Encryption function with key k
X, Y	Challenges	k_s	Server's symmetric encryption key
sk	Session key	sk_{old}	Previous session key
z	Random number	a, a'	Hash values of previous session keys
z'	Solution of puzzle	l	Lifetime of cookies and tickets
n	Length of z	t, t'	Timestamps on the tickets and cookies

3.1 Overview

Similar to private-public key pairs in public key systems, in SPAKA protocols, Alice (the client) generates a `password` π and a `verifier` v that is the public part (trapdoor one-way function) of her password. The verifier is only revealed to Bob (login server) via a secure channel. During a successful execution of a SPAKA protocol, Alice and Bob authenticate each other and agree on a session key. At this time, in our proposed protocols, Bob issues a $cookie = E_{k_s}(C,\ a',\ t',\ l)$ to Alice.

The *cookie* contains the client ID (C), the hash value of the session key ($a' = h(sk_{old})$), a timestamp (t'), and the lifetime (l) of the cookie. (The session key will be called sk_{old} when the *cookie* is used next time.) The *cookie* is encrypted using Authenticated Encryption [3] with a key k_s known only to Bob. Note that both the encryption and the decryption of the *cookie* are done by Bob himself. Also note that the timestamp records Bob's local time. Only Bob will check the timestamp when the *cookie* is used.

Alice generates a $ticket = \{C,\ a,\ t,\ l\}$, which contains the same fields as in the *cookie* except the timestamp (t) of the *ticket* records Alice's local time. For the sake of clarity, we denote the hash value ($h(sk_{old})$) generated by Alice as a to distinguish it from the hash value (a') generated by Bob. Only Alice will check t before using the (*cookie*, *ticket*) pair. Hence, Alice and Bob do not need clock synchronization for using *cookie* and *ticket*. Alice stores both the *cookie* and the unencrypted *ticket* in her local computer. Bob does not store any of them.

When Alice tries to login again, she sends the *cookie* to Bob and finds a in her corresponding *ticket*. Then they run the SPAKA protocol. In the proposed protocols, in order to proceed, Alice must prove that she knows a. If Alice tries to login from a computer without a valid (*cookie*, *ticket*) pair, Bob makes a *puzzle* as described in section 3.2. In order to proceed, Alice must solve the puzzle first.

The proposed protocols achieve following properties:

1. If a user tries to login from a computer without a valid (*cookie*, *ticket*) pair, the client must solve a *puzzle*. In other words, an attacker cannot verify her guesses without solving a *puzzle*, even if she launch many-to-many guessing attacks to 3-pass protocols.
2. A client with a valid (*cookie*, *ticket*) pair does not have to solve a puzzle. In this case, our protocol adds negligible computation on the legitimate client side.
3. They add negligible computation on the server side.
4. They do not increase the number of messages exchanged between the clients and the servers.
5. Servers can easily self-adjust the hardness of puzzles as well as the lifetime of (*cookie*, *ticket*) pairs.
6. If an attacker can somehow get access to one of the user's computers, she may steal a (*cookie*, *ticket*) pair. Only this user will be affected. Her account still has the strength of the original SPAKA protocols. In addition, in section 4 we present a scheme to counter the stolen tickets problem.

3.2 A Puzzle Tailored for SPAKA Protocols

For our purpose, a puzzle shall satisfy the following requirements.

- Creating a puzzle and verifying a solution shall be inexpensive.
- The cost of solving the puzzle shall be easy to adjust.
- It shall not be possible to precompute solutions to the puzzles.
- An attacker shall not be able to relay the puzzles to a third party.
- It shall not require clock synchronization between the client and the server.
- It shall not use encryption.

The first four requirements are common to most cryptographic puzzle schemes. Clock synchronization and encryption are not used because they are not used in the most of SPAKA protocols. Since we integrate our puzzle to SPAKA protocols and aim to use the new protocols in any environment where the original SPAKA protocols are used, we do not ask for more cryptographic primitives or system services than those already used in the original SPAKA protocols. Note that statelessness, which is a common requirement of a client puzzle system aiming to prevent TCP SYN flood attack [8] is not a concern in our scheme, because our puzzle is integrated to SPAKA protocols that are not stateless.

The solution to our puzzle is the brute-force reversal of a cryptographically strong hash function, such as SHA-1 [20]. Suppose a server requires a client to solve a puzzle, the server computes $puzzle = h(z,\ Y,\ IP_c)$, where $z \in_R \mathbb{Z}_{2^n}$ is a n-bit random number ($0 \leq n \lll N$), n controls the hardness of the $puzzle$. Y is a random (long and unpredictable) challenge sent by Bob to Alice in the SPAKA protocols. IP_c is the IP address of the client. If IP_c is not included then a relaying attack is possible where the attacker also runs his own SPAKA+ server. The attacker acquires puzzles from the legitimate SPAKA+ server and relays these puzzles to her clients through her SPAKA+ server; thus, forcing her clients to solve puzzles on her behalf. The server sends $puzzle$, Y, and n to the client. Since Y is long and changes per session, it is not possible for the client to precompute solutions to the puzzles. Creating a puzzle that requires one random number generation and one hash computation is inexpensive. The server can adjust the cost of solving the puzzle by simply tuning n.

Even if the client knows Y and IP_c, due to the one-way property of the cryptographic hash function $h(\cdot)$, the client has no efficient way to solve the puzzle than trying different numbers $z' \in \mathbb{Z}_{2^n}$ until $puzzle = h(z',\ Y,\ IP_c)$. We denote $h^{-1}(\cdot)$ as the procedure used to solve the puzzle. On average, it takes 2^{n-1} trials to solve this puzzle. Next, the client proves to the server that a solution of the puzzle is found. The verification of the solution requires one hash computation on the server side. This step is integrated into SPAKA protocols (see sections 3.3 and 3.4 for detail).

Since above computations only use functions already used in SPAKA protocols, the puzzle can be used in any type of client platform running SPAKA protocols. It is easy to see that the simple puzzle satisfies all requirements listed above.

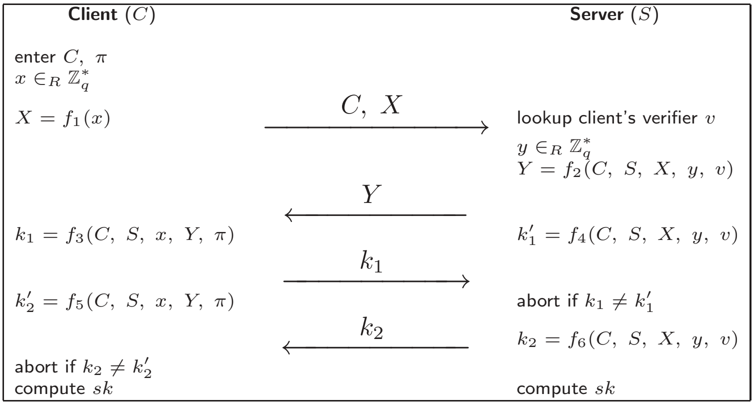

Fig. 1. General 4-pass protocol.

3.3 Strengthening 4-Pass Protocols

Figure 1 depicts the general structure of 4-pass SPAKA protocols described below.

- Alice enters her username C and password π.
- Alice generates a random number x and composes a challenge X based on x. She sends C and the challenge X to Bob.
- Bob looks up Alice's entry in the password file and finds her verifier v. He generates a random number y and composes his challenge Y based on y and other information such as v. Y is sent to Alice.
- Alice computes k_1 and sends it to Bob. The value k_1 serves as her response to Bob's challenge, the proof of her knowledge of π, and the key confirmation.
- Bob computes k_1' that should be the same as k_1 if Alice entered the correct password. If k_1' is equal to k_1, then Bob computes k_2 and sends it to Alice. Similarly, the value k_2 serves as his response to Alice's challenge, the proof of his knowledge of v, and the key confirmation.
- Alice computes k_2' and checks if k_2 and k_2' match.
- Both Alice and Bob believe they are talking to the right party. They then compute the session key sk.

When Alice (or Eve who is an attacker) tries to login from a machine **without** a valid (*cookie*, *ticket*) pair, she must solve a puzzle. The situation is depicted in Figure 2. Bob creates a *puzzle* and sends *puzzle*, Y and n to Alice together in the second message. Alice solves the puzzle by brute-force reversing $h(\cdot)$. Instead of sending k_1 to Bob, she sends $k_p = h(k_1,\ z')$. Bob computes k_1' and $k_p' = h(k_1',\ z)$. He proceeds only if k_p is equal to k_p'. To keep the protocol to be 4-pass, Bob computes sk before sending out the 4th message. Finally, he sends $cookie_{new}$ and the lifetime of $cookie_{new}$ in the 4th message.

Assume the output of $h(\cdot)$ is random. Then the probability that $\exists z' \in_R \mathbb{Z}_{2^n}$, s.t. $z' \neq z$ and $h(z',\ Y,\ IP_c) = h(z,\ Y,\ IP_c)$ is $\frac{1}{2^{N-n}}$. Since $n \ll N$, with

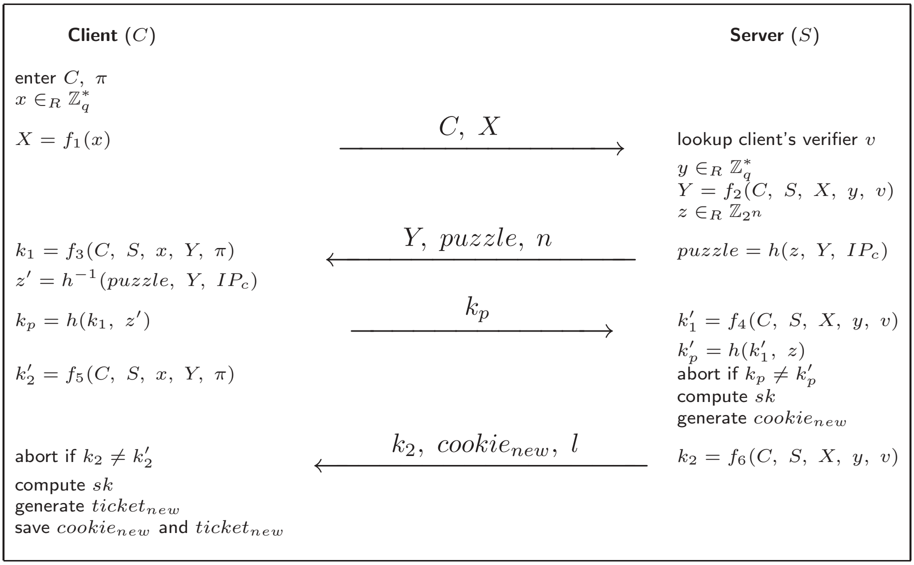

Fig. 2. 4-pass protocol without *cookie* and *ticket*.

high probability z' is equal to z. If k_1 is equal to k_1', i.e., the password and the verifier match, then k_p is equal to k_p'. In this case, mutual authentication between Alice and Bob is successful. On the other hand, suppose Eve is running an online dictionary attack. She has to solve the puzzle. Without correct z', with the probability $1 - \frac{1}{2^n}$, $k_p \neq k_p'$ even if k_1 is equal to k_1', (i.e., even if she guessed the correct password). So checking if k_p is equal to k_p' implicitly verifies if k_1 is equal to k_1' as well as if z is equal to z'. In other words, it verifies if Alice (or Eve) entered the correct password and has solved the puzzle.

Figure 3 represents the scenario when Alice tries to login **with** a valid (*cookie*, *ticket*) pair. Alice sends the cookie to Bob in the first message. Bob decrypts the cookie and verifies if it is expired. If not, he saves a'. Instead of sending k_1 to Bob in the third message, Alice sends $k_p = h(k_1,\ a)$. Bob computes k_1' and $k_p' = h(k_1',\ a')$. He proceeds only if k_p is equal to k_p'. Similar to the above protocol, this step implicitly verifies if k_1 is equal to k_1' as well as if a is equal to a'. Finally, Bob sends $cookie_{new}$ and l in the 4th message. In this case, Alice is not asked to solve a puzzle, but she must have a valid $(cookie, ticket)$ pair.

3.4 Strengthening 3-Pass Protocols

In a 3-pass protocol, Bob computes k_1 and sends it to Alice in the second message with Bob's challenge. Alice verifies it, then sends k_2 in the third message. Figure 4 represents the general structure of 3-pass SPAKA protocols.

As depicted in figure 5, when a Alice (or Eve) tries to login from a machine **without** a valid (*cookie*, *ticket*) pair, she must solve a puzzle. Similar to 4-pass protocol, Bob creates a *puzzle*. He also computes $k_p = h(k_1,\ z)$ and sends them

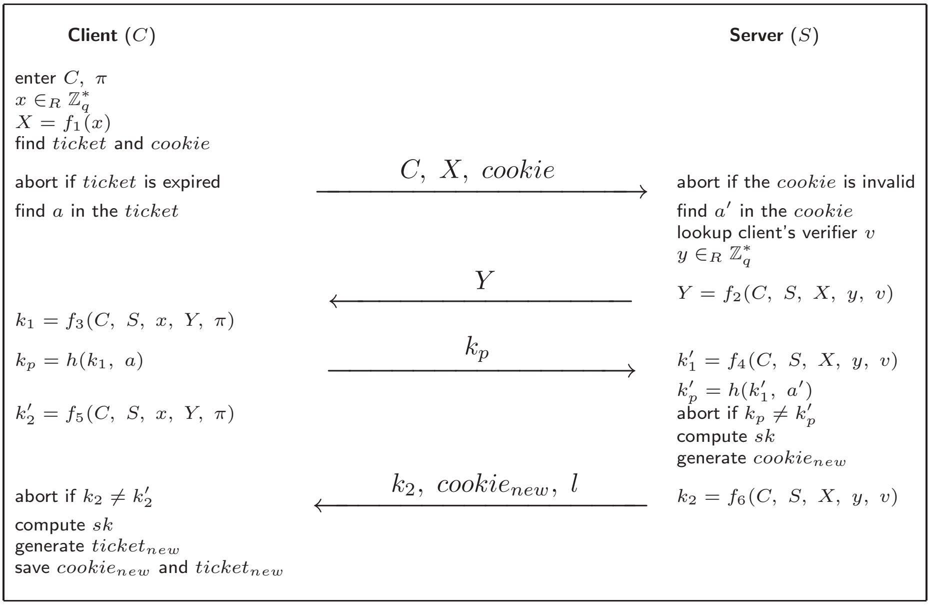
Client (C)
Server (S)
enter $C,\ \pi$
$x \in_R \mathbb{Z}_q^*$
$X = f_1(x)$
find $ticket$ and $cookie$
abort if $ticket$ is expired
find a in the $ticket$
$C,\ X,\ cookie$
abort if the $cookie$ is invalid
find a' in the $cookie$
lookup client's verifier v
$y \in_R \mathbb{Z}_q^*$
Y
$Y = f_2(C,\ S,\ X,\ y,\ v)$
$k_1 = f_3(C,\ S,\ x,\ Y,\ \pi)$
$k_p = h(k_1,\ a)$
k_p
$k_1' = f_4(C,\ S,\ X,\ y,\ v)$
$k_p' = h(k_1',\ a')$
abort if $k_p \neq k_p'$
compute sk
generate $cookie_{new}$
$k_2' = f_5(C,\ S,\ x,\ Y,\ \pi)$
$k_2,\ cookie_{new},\ l$
$k_2 = f_6(C,\ S,\ X,\ y,\ v)$
abort if $k_2 \neq k_2'$
compute sk
generate $ticket_{new}$
save $cookie_{new}$ and $ticket_{new}$

Fig. 3. 4-pass protocol with *cookie* and *ticket*.

to Alice in the second message. To keep the protocol to be 3-pass, Bob computes sk earlier and sends $cookie_{new}$ and l in the second message since that is the only message Bob sends to Alice. After receiving the second message, Alice solves the $puzzle$ by brute-force reversing $h(\cdot)$ and computes k_1' and $k_p = h(k_1,\ z')$. If k_p is equal to k_p', she concludes that the solution of the puzzle is correct and the verifier matches the password. Following the general 3-pass protocol she first computes k_2 and then computes $k_q = h(k_2,\ z')$ and sends k_q to Bob instead of sending k_2.

The value k_1 sent in the second message must be replaced with k_p. Otherwise Eve can disconnect after receiving the second message. Since k_1 serves as key confirmation, it gives Eve enough information to check if her guessing is correct or not. On the other hand, the value k_p is the hash value of $(k_1,\ z)$. To verify k_1, Eve has to first solve the $puzzle$. The solution of the $puzzle$ is also sent back to Bob in the third message implicitly. Suppose only k_2 is sent to Bob, Eve can bypass the $puzzle$ and wait to see if Bob aborts or not. If not, she knows her guess is correct. Note z and k_2 can be sent in clear. We use k_q to keep the protocol consistent with the 3-pass with cookie case since a cannot be sent in clear. To keep the new protocol to be 3-pass, Bob must send $cookie_{new}$ in the second message. If Eve gets $cookie_{new}$, but it is not useful to her as she cannot decrypt $cookie_{new}$.

Similar to 4-pass protocol, when Alice tries to login **with** a valid (*cookie*, *ticket*) pair, she is not asked to solve a $puzzle$. Figure 6 depicts this scenario. As in the 3-pass without cookie case, we use k_p and k_q instead of k_1 and k_2. Again, Bob sends $cookie_{new}$ and l in the second message.

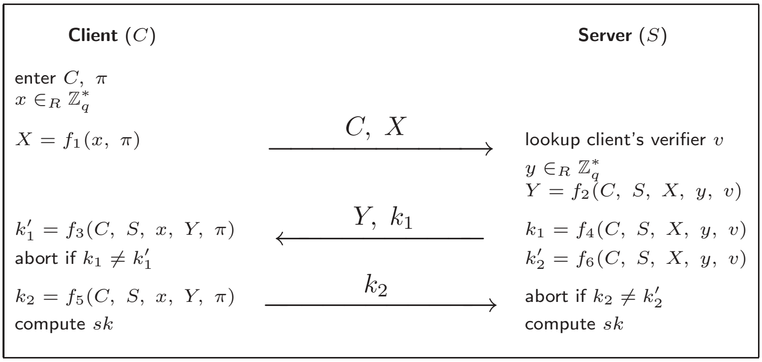

Fig. 4. General 3-pass protocol.

3.5 Adjusting Hardness of Puzzles

We modified a test program of the SRP implementation [28] to test the performance of generating a puzzle, solving the puzzle, and verifying the solution. We run the test program on a Pentium-4 2.4GHz computer running Linux with kernel version 2.4.20. As expected, generating a puzzle and verifying the solution takes negligible amount of time. The time required to solve a puzzle is roughly doubled, when we increase the hardness by one. Bob can self-adjust the hardness of puzzles by keeping a global counter to count the number of failed attempts to all accounts in the system within an interval. He adjusts the hardness of puzzles when the counter reaches predefined threshold values.

4 Discussion

In this section we highlight the various aspects of our scheme. Especially, we present a mechanism that does not give Eve significant advantage even if Eve successfully steals Alice's (*cookie*, *ticket*) pairs by exploiting vulnerabilities in the underlying system.

Usability. Given a computationally intensive cryptographic puzzle, different machines may spend different amount of time to solve it. If a legitimate user is using a slower machine, she has to spend more time solving a puzzle. However, after a successful login she will have a (*cookie*, *ticket*) pair, and, therefore, she does not need to solve puzzles as long as she keeps using the same set of machines. The usability of the system is not sacrificed.

Client-side cookies. In our approach, *cookie*s are stored in users' computers. Once the client receives a new *cookie* (after successful login), the client can simply delete the stale *cookie*s for that account; therefore, the maximum number of *cookie* stored on a user's computer is equal to the the number of accounts the user has. If *cookie*s are stored in the server, then the server has to store all *cookie*s that have not expired. If the authentication service is heavily used and if the lifetime of *cookie*s is long, then the server has to store a large number of *cookie*s.

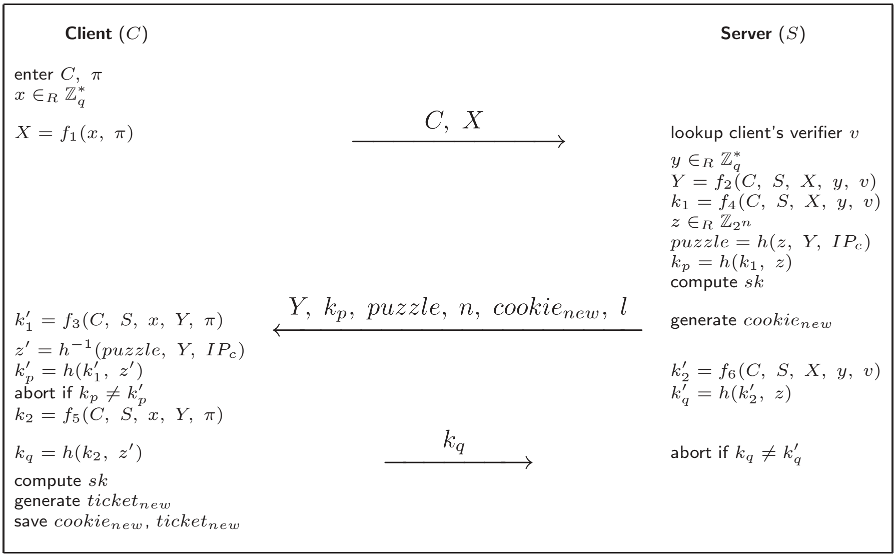

Fig. 5. 3-pass protocol without *cookie* and *ticket*.

No information leakage. Comparing to the 3-pass or 4-pass general protocols, more information is sent in our new protocols, namely $puzzle$, k_p, k_q, $cookie$, $cookie_{new}$. These pieces of information do not help Eve to find valuable information, such as the password, the verifier, previous session keys, or the new session key. The $cookie_{new}$ (or $cookie$) contains the hash of (previous) session key and is encrypted with a key known only to the server; therefore, Eve cannot find the (previous) session key. In the case when Alice attempts to login without a valid ($cookie$, $ticket$) pair, k_p (or k_q) is the hash value of (k_1, z) (or (k_2, z)). Since we assume that the general protocols are secure, k_1 (k_2) that can be easily eavesdropped in SPAKA protocols does not leak valuable information, neither does k_p (k_q) as a result. In the case when Alice attempts to login using a valid ($cookie$, $ticket$) pair, the same argument applies, k_p (k_q) does not leak valuable information including a.

Combatting many-to-many attacks. 3-pass SPAKA protocols are vulnerable to the many-to-many guessing attack [18]. This attack is common to 3-pass SPAKA protocols because Bob must send his challenge and the key confirmation that also helps Alice to authenticate Bob in one message. This enables Eve to disconnect an ongoing session earlier (right after receiving Bob's message) and verify her guess later offline. She can also run multiple sessions simultaneously and collect more useful values. With SPAKA+, Eve can still disconnect earlier. However, she cannot verify her guesses without solving puzzles. The amount of work Eve can save is one message per guess. The cost of sending one message is several magnitudes lower than the cost of solving a puzzle. Hence, Eve's cost of running online dictionary attacks and her cost of running many-to-many attacks

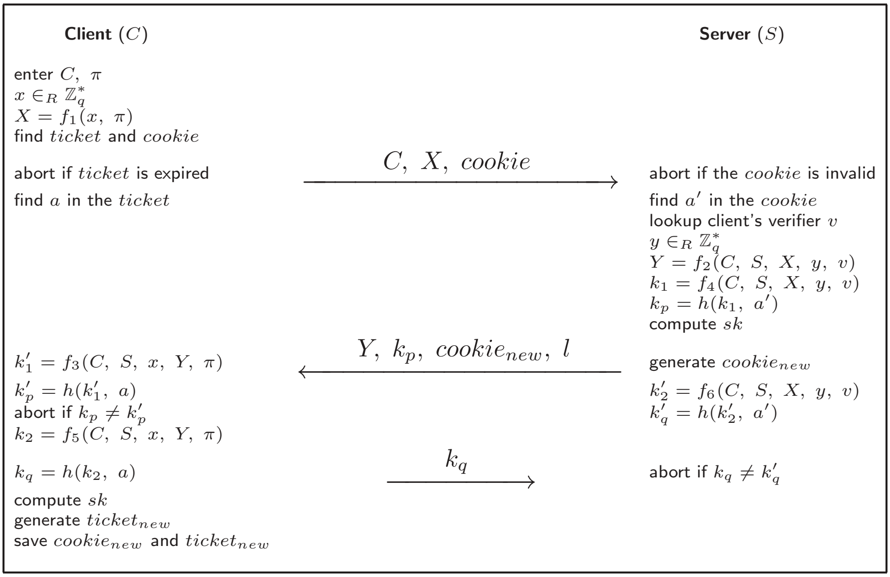
Client (C)
Server (S)
enter $C,\ \pi$
$x \in_R \mathbb{Z}_q^*$
$X = f_1(x,\ \pi)$
find $ticket$ and $cookie$
abort if $ticket$ is expired
find a in the $ticket$
$C,\ X,\ cookie$
abort if the $cookie$ is invalid
find a' in the $cookie$
lookup client's verifier v
$y \in_R \mathbb{Z}_q^*$
$Y = f_2(C,\ S,\ X,\ y,\ v)$
$k_1 = f_4(C,\ S,\ X,\ y,\ v)$
$k_p = h(k_1,\ a')$
compute sk
$Y,\ k_p,\ cookie_{new},\ l$
generate $cookie_{new}$
$k_2' = f_6(C,\ S,\ X,\ y,\ v)$
$k_q' = h(k_2',\ a')$
$k_1' = f_3(C,\ S,\ x,\ Y,\ \pi)$
$k_p' = h(k_1',\ a)$
abort if $k_p \neq k_p'$
$k_2 = f_5(C,\ S,\ x,\ Y,\ \pi)$
$k_q = h(k_2,\ a)$
k_q
abort if $k_q \neq k_q'$
compute sk
generate $ticket_{new}$
save $cookie_{new}$ and $ticket_{new}$

Fig. 6. 3-pass protocol with *cookie* and *ticket*.

is about the same. Therefore, the vulnerability of 3-pass SPAKA protocols is significantly reduced.

Analyzing the possibility of ticket theft. If an attacker is able to steal a ticket, then she may be able to bypass our puzzle. We now analyze the scenarios in which a client's ticket can be stolen. If a user is always using her home (or personal) computer or a well administered lab computer with appropriate user accounts, then the chances of stealing the client's ticket are slim, since the ticket is protected with a strong access control mechanism. Even in the case where a user is using a well administered lab computer, which is using a network file system, the chances that an attacker successfully steals a client's ticket are low as the attacker will have to run a sniffer on the local network to eavesdrop the ticket stored on the remote file server, which typically is difficult (or to some extent easy to detect) in a well administered lab. In the aforementioned scenarios, an attacker will have to hack into the user's account to get the ticket. If a client is travelling and uses for a short time a computer that is not well administered, then an attacker can steal the client's ticket, but just stealing one ticket does not give substantial advantage to the attacker. The threat of ticket theft is high only when a lot of users are frequently using computers in a public lab that does not have any notion of user accounts. In this scenario, an attacker can simply read the tickets stored on these computers and bypass the puzzle. Below, we explain in detail how SPAKA+ minimizes the threat of stolen tickets. The mechanism described is quite light weight if the number of stolen tickets is small as compared to the number of users in the system, which is true for most of the cases discussed above.

Resisting stolen tickets. To minimize the threat of stolen tickets, Bob maintains two lists, namely *Cookie Cache* and *Black List*, to store *cookie*s. Both of the lists are initially empty. He also maintains a counter (initialized to 0) for each *cookie* stored in the *Cookie Cache* and a small threshold value (*thresh*), say 5. When a user C tries to login with a (*cookie*, *ticket*) pair and his password π, Bob first checks if C is been served by one of his instances. If so, Bob aborts this session, because it is likely that C is Eve exploiting many-to-many attack [18]. Bob does not allow parallel SPAKA+ sessions for one user account. If not, Bob searches the *cookie* in the *Black List*. If found, Bob runs the SPAKA+ and asks C to solve a puzzle. If Bob does not find the *cookie* in the *Black List*, he runs the SPAKA+ protocol without asking C to solve a puzzle. If Bob cannot authenticate C, he checks the *Cookie Cache*. If the *Cookie* is not in the *Cookie Cache*, then he inserts the *Cookie* is the *Cookie Cache*. If the *cookie* is already in the *Cookie Cache*, then he increase the counter of this *Cookie*. If the counter is larger than *thresh*, then he adds the *Cookie* in the *Black list*. On the other hand, if C entered the correct password, then Bob allows C to login and delete the *Cookie* from the *Cookie Cache* if it is there. Bob will also delete expired *cookie*s from the *Cookie Cache* and the *Black List* periodically.

Following above scheme, when Alice tries to login with a valid (*cookie*, *ticket*) pair but entered a wrong password, the login attempt will fail. At this time, Bob temporarily caches the *cookie*. Since Alice knows the correct password, she is very likely to try again and enters the correct password within a couple of trails. Bob sees that Alice entered the correct password, then deletes the cached *cookie* and the corresponding counter. In this case, Bob only maintains a short term state.

If Eve runs online dictionary attacks with one of Alice's valid (*cookie*, *ticket*) pairs, she gets at most $(thresh - 1) + thresh$ chances without being asked to solve a puzzle. She tries $(thresh - 1)$ times then waits for Alice to login with the same (*cookie*, *ticket*) pair. Bob deletes the *cookie* from *Cookie Cache*, and, therefore, Eve gets *thresh* more chances. Note that Alice will delete this (*cookie*, *ticket*) pair from her computer since she gets a new pair on the next successful login. Therefore, she will not use the pair that Eve has again. It is very unlikely that Eve can guess the correct password within $(thresh-1)+thresh$ trials. Now Bob considers that this *cookie* and its corresponding *ticket* are stolen and puts Alice's ID and the *cookie* in his *Black List*. In this case, Bob maintains a long term state. The more insecure the computer is, the more (*cookie*, *ticket*) pairs Eve can steal, as a result bigger the Bob's *Black List*. As a side effect, the system employing above scheme benefits from the *Black List*. By periodically analyzing the *Black List*, Bob can estimate if Alice's computer is secure by counting the number of Alice's *cookie*s in the *Black List* and notifying Alice if he believes Alice's computer is not secure.

5 Related Work

Pinkas and Sander proposed a well crafted scheme that attempts to slow down online dictionary attacks on web applications with Reverse Turing Test (RTT)

[22]. To increase the usability, servers issue *cookie*s to clients. The *cookie*s must be protected against eavesdropping and *cookie* theft. Hence, it requires that the server keeps a counter for every *cookie* stored in a user's machine. First of all, these schemes are mainly restricted to GUI based applications and cannot be applied to non-GUI based applications. The scheme presented in this paper can be easily integrated with GUI based as well as non-GUI based authentication systems. In addition, if under attack, the hardness of the puzzle can be increased to further slow down the attacker. Cookies used in Pinkas and Sander scheme are vulnerable to web based cookie stealing attacks. We use tickets and encrypted cookies to avoid puzzles. Clients prove that they have tickets without revealing the tickets (sending the tickets on the network). The only way an attacker can steal a ticket is by breaking into a user's account. Hence, our cookie-ticket pair approach is more secure than the cookie only approach used in Pinkas and Sander's approach. Further, our scheme requires less server storage than that required in [22]. In their approach, a server keeps a counter for every cookie stored on users' computers. In our approach, on one hand, if we assume that users' computers are secure, then the server does not cache cookies, it only stores one global counter. On the other hand, if users' computers are not assumed to be secure and the "resisting stolen tickets" protocol (explained at the end of section 4) is used, then the number of cookies and the corresponding counters maintained by the server is equal to the number tickets stolen by an attacker, which is still less than their approach.

Cryptographic puzzles have been used in the literature for several related tasks. Rivest *et al.* used puzzles to create digital time capsules [23]. Juels and Brainard introduced the first proposal for using a client puzzle approach to defend against connection depletion attacks [14]. Aura *et al.* [2] proposed an approach to protect authentication protocols against denial-of-service. [9] reported a implementation of client puzzles in the context of TLS. Wang and Reiter's [26] approach enables each client to "bid" for resources by tuning the difficulty of the puzzles it solves, and to adapt its bidding strategy in response to apparent attacks. The sever allocates resources first to the client that solved the most difficult puzzle when the server is busy. Dwork and Naor introduced the *pricing via processing* paradigm and designed puzzles for combatting spam [11]. One interesting property of their puzzle is the *short cut*. The short cut information is only known by trusted agents and "pricing authority". Normal legitimate users do not know the short cut. For the various purposes of their applications, the above approaches are not required to distinguish between legitimate users and attackers. Every one is required to solve a puzzle.

6 Conclusion and Future Work

We introduced SPAKA+ that strengthens SPAKA protocols against online dictionary attacks using cryptographic puzzles. SPAKA+ significantly increases the complexity of online dictionary attacks as well as many-to-many guessing attacks. The server can self-adjust the computational burden of an attacker by tuning the hardness of the puzzles in real-time based on the server's estimate of ongoing online dictionary attacks. SPAKA+ is secure and adds negligible load on the legitimate clients. We designed a simple cryptographic puzzle that utilizes the nice structure of the SPAKA protocols for generating puzzles, transferring puzzles and solutions, and verifying solutions. The puzzle is designed such that attackers cannot relay it to others. The puzzle does not require any function other than those used in the SPAKA protocols. As a result, the new scheme can be used without requiring additional support from the underlying system and is easy to implement. If the users' computers are secure, the server's load in our scheme is low. Whereas if the users' computers are assumed to be insecure we have presented an approach that resists stolen tickets by maintaining state information on the server.

Future work includes integrating the new scheme into available SPAKA implementations, such as SRP and PAK. We plan to implement an experimental system to perform online dictionary attacks and many-to-many guessing attacks in order to evaluate the success of our scheme. A detailed evaluation both from performance perspective as well as usability perspective will be performed. We also plan to apply our idea to SSH.

References

1. Research papers on password-based cryptography. `http://www.jablon.org/passwordlinks.html`.
2. T. Aura, P. Nikander, and J. Leiwo. DOS-resistant authentication with client puzzles. In *the 8th International Workshop on Security Protocols*, 2001.
3. M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT*, 2000.
4. M. Bellare and P. Rogaway. The AuthA protocol for password-based authenticated key exchange, 2000. Submission to IEEE P1363.2.
5. S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *IEEE Symposium on Security and Privacy*, 1992.
6. V. Boyko, p. MacKenzie, and S. Patel. Provably secure password authentication and key exchange using diffie-hellman. In *EUROCRYPT*, 2000.
7. P. Buxton. Egg rails at password security. Netimperative, June, 24, 2002.
8. CERT. TCP syn flooding and ip spoofing attack. CERT Advisory CA-96.21, November 1996.
9. D. Dean and A. Stubblefield. Using client puzzles to protect TLS. In *the 10th Annual USENIX Security Symposium*, 2001.
10. D. Denning and G. Sacco. Timestamps in key distribution systems. *Communications of the ACM*, August 1981.
11. C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, 1993.
12. IEEE P1363 Working Group. IEEE P1363-2: Standard specifications for password-based public key cryptographic techniques. `http://grouper.ieee.org/groups/1363`.
13. D. P. Jablon. Strong password-only authenticated key exchange. *Computer Communication Review*, 26(5):5–26, 1996.

14. A. Juels and J. Brainard. Client puzzles: A cryptographic defense against connection depletion attacks. In *Network and Distributed System Security Symposium*, 1999.
15. D. V. Klein. "foiling the cracker" – A survey of, and improvements to, password security. In *the second USENIX Workshop on Security*, 1990.
16. E. Knight and C. Hartley. The password paradox. Business Security Advisor magazine, December 1998.
17. T. Kwon. Authentication and key agreement via memorable password. In *Network and Distributed System Security Symposium*, 2001.
18. T. Kwon. Practical authenticated key agreement using passwords. the 7th Information Security Conference (ISC), 2004.
19. T. Lomas, L. Gong, J. Saltzer, and R. Needhamn. Reducing risks from poorly chosen keys. In *the twelfth ACM symposium on Operating systems principles (SOSP)*, 1989.
20. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, October 1996.
21. R. T. Morris and K. Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, Nov 1979.
22. B. Pinkas and T. Sander. Securing passwords against dictionary attacks. In *the 9th ACM conference on Computer and communications security*, 2002.
23. R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical Report LCS/TR-684, MIT, 1996.
24. E. Spafford. Observing reusable password choices. In *the 3rd UNIX Security Symposium*, 1992.
25. Transport Layer Security Working Group. SSL 3.0 specification. `http://wp.netscape.com/eng/ssl3/`.
26. X. Wang and M. K. Reiter. Defend against denial-of-service attacks with puzzle auctions. In *the IEEE Symposium on Security and Privacy*, 2003.
27. T. Wu. The secure remote password protocol. In *Network and Distributed System Security Symposium*, 1998.
28. T. Wu. The stanford SRP authentication project, February 2004. `http://srp.stanford.edu`.
29. T. Ylonen. SSH - secure login connections over the internet. volume the 6th USENIX Security Symposium, 1996.
30. Scotland Yard and the case of the rent-a-zombies. `http://news.zdnet.com/2100-1009_22-5260154.html`, July 2004.
31. Zombie PCs for Rent. `http://securitynews.weburb.dk/show.php3?item=InformationSecurity&p%5Bne%wsletterId%5D=609`, September 2004.