A New Two-Server Approach for Authentication with Short Secrets

John Brainard, Ari Juels, Burt Kaliski, and Michael Szydlo
RSA Laboratories
Bedford, MA 01730, USA
E-mail: {jbrainard,ajuels,bkaliski,mszydlo}@rsasecurity.com

Abstract

Passwords and PINs continue to remain the most widespread forms of user authentication, despite growing awareness of their security limitations. This is because short secrets are convenient, particularly for an increasingly mobile user population. Many users are interested in employing a variety of computing devices with different forms of connectivity and different software platforms. Such users often find it convenient to authenticate by means of passwords and short secrets, to recover lost passwords by answering personal or "life" questions, and to make similar use of relatively weak secrets.

In typical authentication methods based on short secrets, the secrets (or related values) are stored in a central database. Often overlooked is the vulnerability of the secrets to theft *en bloc* in the event of server compromise. With this in mind, Ford and Kaliski and others have proposed various password "hardening" schemes involving multiple servers, with password privacy assured provided that some servers remain uncompromised.

In this paper, we describe a new, two-server secure roaming system that benefits from an especially lightweight new set of protocols. In contrast to previous ideas, ours can be implemented so as to require essentially *no intensive cryptographic computation* by clients. This and other design features render the system, in our view, the most practical proposal to date in this area. We describe in this paper the protocol and implementation challenges and the design choices underlying the system.

1 Introduction

In this paper, we consider a basic, pandemic security problem: How is it possible to provide secure services to users who can authenticate using only short secrets or weak passwords?

This problem is of growing importance as Internet-enabled computing devices become ever more prevalent and versatile. These devices now include among their ranks an abundant variety of mobile phones, personal digital assistants (PDAs), and game consoles, as well as laptop and desktop PCs. The availability of networks of computers to highly mobile user populations, as in corporate environments, means that a single user may regularly employ many different points of remote access. The roaming user may additionally employ any of a number of different devices, not all of which necessarily possess the same software or configuration.

While smartcards and similar key-storage devices offer a secured, harmonized approach to authentication for the roaming user, they lack an adequately developed supporting infrastructure in many computing environments. At present, for example, very few computing devices contain smartcard readers – particularly in the United States. Furthermore, many users find physical authentication tokens inconvenient. Another point militating against a critical reliance on hardware tokens is the common need to authenticate roaming users who have lost or forgotten their tokens, or whose tokens have malfunctioned. Today, this is usually achieved by asking users to provide answers to a set of "life" questions, i.e., questions regarding personal and private information. These observations stress that roaming users must be able to employ passwords or other short pieces of memorable information as a form of authentication. Indeed, short secrets like passwords and answers to life questions are the predominant form of authentication for most users today. They are the focus of our work here.

To ensure usability by a large user population, it is important that passwords be memorable. As a result, those used in practice are often highly vulnerable to brute-force guessing attacks [21]. Good credential-server designs must therefore permit secure authentication assuming a weak key (password) on the part of the user.

1.1 SPAKA protocols

A basic tool for mutual authentication via passwords, and one well developed in the literature, is *secure password-authenticated key agreement* (SPAKA). Most SPAKA protocols are descendants of Bellovin and Merrit's EKE protocol [3, 4], and are predicated on either Diffie-Hellman key agreement or key agreement using RSA. The client and server share a password, which is used to achieve mutual assurance that a cryptographically strong session key is established privately by the two parties. To address the problem of weak passwords, SPAKA protocols are constructed so as to leak no password information, even in the presence of an active attacker. When used as a means of authentication to obtain credentials from a trusted server, a SPAKA protocol is typically supplemented with a throttling or lockout mechanism to prevent on-line guessing attacks. Many roaming-credentials proposals involve use of a SPAKA protocol as a leverage point for obtaining credentials, or as a freestanding authentication protocol. A comprehensive, current bibliography of research papers on the topic of SPAKA protocols (of which there are dozens) is maintained by David Jablon, and may be found at [17].

The design of most SPAKA protocols overlooks a fundamental problem: The server itself represents a serious vulnerability. As SPAKA protocols require the verifying server to have cleartext access to user passwords (or to derivative material), compromise of the server leads potentially to exposure of the full database of passwords. While many SPAKA protocols store passwords in combination with salt or in some exponentiated form, an attacker who compromises the server still has the possibility of mounting off-line dictionary attacks. Additionally, these systems offer no resistance to server corruption. An attacker that gains control of the authenticating server can spoof successful login attempts.

To address this problem, Ford and Kaliski [13] introduced a system in which passwords are effectively protected through distribution of trust across multiple servers. Mackenzie, Shrimpton, and Jakobsson [24] extended this system, leading to more complex protocols, but with rigorous security reductions in a broadly inclusive attack model. Our work in this paper may be regarded as a complement, rather than a successor to the work of these authors. We propose a rather different technical approach, and also achieve some special benefits in our constructions, such as a substantially reduced computational load on the client. At the same time, we consider a different, and in our view more pragmatic security model than that of other distributed SPAKA protocols.

1.2 Previous work

The scheme of Ford and Kaliski reduces server vulnerability to password leakage by means of a mechanism called *password hardening*. In their system, a client parlays a weak password into a strong one through interaction with one or multiple hardening servers, each one of which blindly transforms the password using a server secret. Ford and Kaliski describe several ways of doing this. Roughly speaking, the client in their protocol obtains what may be regarded as a blind function evaluation σ_i of its password P from each hardening server S_i. (The function in question is based on a secret unique to each server and user account.) The client combines the set of shares $\{\sigma_i\}$ into a single secret σ, a strong key that the user may then use to decrypt credentials, authenticate herself, etc. Given an appropriate choice of blind function evaluation scheme, servers in this protocol may learn no information, in an information-theoretic sense, about the password P. An additional element of the protocol involves the user authenticating by means of σ (or a key derived from it) to each of the servers, thereby proving successful hardening. The harderened password σ is then employed to decrypt downloaded credentials or authenticate to other servers. We note that the Ford-Kaliski system is designed for credential download, and not password recovery; our system is specially designed to support both. Another important distinction is that in the Ford-Kaliski system, the client interacts with both servers directly. As we describe, an important feature of our proposed system is the configuration of one server in the back-end, yielding stronger privacy protection for users.

Mackenzie *et al.* extend the system of Ford and Kaliski to a threshold setting. In particular, they demonstrate a protocol such that a client communicating with any k out of n servers can establish session keys with each by means of password-based authentication; even if $k-1$ servers conspire, the password of the client remains private. Their system can be straightforwardly leveraged to achieve secure downloadable credentials. The Mackenzie *et al.* system, however, imposes considerable overhead of several types. First, servers must possess a shared global key and local keys as well (for a total of $4n+1$ public keys). The client, additionally, must store $n+1$ (certified) public keys. The client must perform several modular exponentiations per server for each session, while the computational load on the servers is high as well. Finally, the Mackenzie *et al.* protocol is somewhat complex, both conceptually and in terms of implementation. On the other hand, the protocol is the first such provided with a rigorous proof of security under the Decision Diffie-Hellman assumption [7] in the random oracle model [2].

Frykholm and Juels [15] adopt a rather different approach, in which encrypted user credentials are stored on a single server. In this system, *no* trust in the server is required to assure user privacy under appropriate cryptographic assumptions. Roughly stated, user credentials are encrypted under a collection of short passwords or keys. Typically, these are answers to life questions. While the Frykholm-Juels system provides error tolerance, allowing the user to answer some questions incorrectly, it is somewhat impractical for a general population of users, as it requires use of a large number of questions. Indeed, the authors recommend a suite of as many as fifteen such questions to achieve strong security. The work of Frykholm and Juels is an improvement on that of Ellison *et al.* [11], which was found to have a serious security vulnerability [5]. This approach may be thought of as an extension to that of protecting credentials with password-based encryption. The most common basis for this in practice is the PKCS #5 standard [1].

1.3 Our work: a new, lightweight system

It is our view that most SPAKA protocols are over-engineered for real-world security environments. In particular, we take the position that that mutual authentication is often not a requirement for roaming security protocols *per se*. Internet security is already heavily dependent upon a trust model involving existing forms of server-side authentication, particularly the well studied Secure Sockets Layer protocol (SSL) [14]. SSL is present in nearly all existing Web browsers. Provided that a browser verifies correct binding between URLs and server-side certificates, as most browsers do, the user achieves a high degree of assurance of the identity of the server with which she has initiated a given session. In other words, server authentication is certainly important, but need not be provided by the same secret as user authentication. Thus many SPAKA protocols may be viewed as replicating functionality already provided in an adequately strong form by SSL, rather than building on such functionality.

Moreover, it may be argued that SPAKA protocols carry a hidden assumption of trust in SSL or similar mechanisms to begin with. SPAKA protocols require the availability of special-purpose software on the client side. Given that a mobile user cannot be certain of the (correct) installation of such software on her device, and that out-of-band distribution of special-purpose software is rare, it is likely that a user will need to download the SPAKA software itself from a trusted source. This argues an *a priori* requirement for user trust in the identity of a security server via SSL or a related mechanism. In this paper, we assume that the client has a pre-existing mechanism for establishing private channels with server-side authentication, such as SSL.

Our system represents an alternative to SPAKAs in addressing "hardening" problem; it is a two-server solution that is especially simple and practical. The idea is roughly as follows. The client splits a user's password (or other short key) P into shares for the two servers. On presenting a password P' for authentication, the client provides the two servers with a new, random sharing of P'. The servers then compare the two sharings of P and P' in such a way that they learn whether $P = P'$, but no additional information. The client machine of the user need have no involvement in this comparison process.

As we explain, it is beneficial to configure our system such that users interact with only one server on the front-end, and pass messages to a second, back-end server via a protected tunnel. This permits the second server to reference accounts by way of pseudonyms, and thereby furnishes users with an extra level of privacy. Such privacy is particularly valuable in the case where the back-end server is externally administered, as by a security-services organization. Much of our protocol design centers on the management of pseudonyms and on protection against the attacks that naïve use of pseudonyms might give rise to.

1.4 Organization

In section 2, we describe the core cryptographic protocol our system for two-server comparison of secret-shared values. We provide an overview of our architecture in section 3, discussing the security motivations behind our choices. In section 4, we describe two specialized protocols in our system; these are aimed at preventing false-identifier and replay attacks. We provide some implementation details for our system in section 5. We conclude in section 6 with a brief discussion of some future directions.

2 An Equality-Testing Protocol

Let us first reiterate and expand on the intuition behind the core cryptographic algorithm in our system, which we refer to as *equality testing*. The basic idea is for the user to register her password P by providing random shares to the two servers. On presenting her password during login, she splits her password into shares in a different, random way. The two servers compare the two sharings using a protocol that determines whether the new sharing specifies the same password as the original sharing, without leaking any additional information (even if one server tries to cheat). For convenience, we label the two servers "Blue" and "Red". Where appropriate in subscripts, we use the lower-case labels "blue" and "red".

Registration: Let $\mathcal{H}$ be a large group (of, say, 160-bit order), and $+$ be the group operator. Let f be a collision-free hash function $f : \{0,1\}^* \rightarrow \mathcal{H}$. To share her password at registration, the user selects a random group element $R \in_U \mathcal{H}$. She computes the share P_{blue} for Blue as $P_{blue} = f(P) + R$, while the share P_{red} of Red is simply R. Observe that the share of either server individually provides no information about P.

Authentication: When the user furnishes password P' to authenticate herself, she computes a sharing based on a new random group element $R' \in_U \mathcal{H}$. In this sharing, the values $P'_{blue} = f(P') + R'$ and $P'_{red} = R'$ are sent to Blue and Red respectively.

The servers combine the shares provided during registration with those for authentication very simply as follows. Blue computes $Q_{blue} = P_{blue} - P'_{blue} = (f(P) - f(P')) + (R - R')$, while Red similarly computes $Q_{red} = P_{red} - P'_{red} = R - R'$. Observe that if $P = P'$, i.e., if the user has provided the correct password, then $f(P)$ and $f(P')$ cancel, so that $Q_{blue} = Q_{red}$. Otherwise, if the user provides $P \neq P'$, the result is that $Q_{blue} \neq Q_{red}$ (barring a collision in f). Thus, to test the user password submitted for authentication, the two servers need merely test whether $Q_{blue} = Q_{red}$, preferably without revealing any additional information.

For this task of equality testing, we require a second, large group $\mathcal{G}$ of order q, for which we let multiplication denote the group operation. The group $\mathcal{G}$ should be one over which the discrete logarithm problem is hard. We assume that the two servers have agreed upon this group in advance, and also have agreed upon (and verified) a generator g for $\mathcal{G}$. We also require a collision-free mapping $w : \mathcal{H} \rightarrow \mathcal{G}$. For equality testing of the values Q_{red} and Q_{blue}, the idea is for the two servers to perform a variant of Diffie-Hellman key exchange. In this variant, however, the values Q_{red} and Q_{blue} are "masked" by the Diffie-Hellman keys. The resulting protocol is inspired by and may be thought of as a technical simplification of the PET protocol in [18]. Our protocol uses only one component of an El Gamal ciphertext [16], instead of the usual pair of components as in PET. Our protocol also shares similarities with SPAKA protocols such as EKE. Indeed, one may think of the equality $Q_{red} = Q_{blue}$ as resulting in a shared secret key, and inequality as yielding different keys for the two servers.

There are two basic differences, however, between the goal of a SPAKA protocol and the equality-testing protocol in our system. A SPAKA protocol, as already noted, is designed for security over a potentially unauthenticated channel. In contrast, our intention is to operate over a private, mutually authenticated channel between the two servers. Moreover, we do not seek to derive a shared key from the protocol execution, but merely to test equality of two secret values with a minimum of information leakage. Our desired task of equality testing in our system is known to cryptographers as the *socialist millionaires' problem*. (The name derives from the idea that two millionaires wish to know whether they enjoy equal financial standing, but do not wish to reveal additional information to one another.) Several approaches to the socialist millionaires' problem are described in the literature, e.g., [8, 12, 19]. In most of this work, researchers are concerned in addressing the problem to ensure the property of *fairness*, namely that both parties should learn the answer or neither. We do not consider this issue here, as it does not have a major impact on the overall system design. (A protocol unfairly terminated by one server in our system is no worse than a password guess initiated by an adversary, and may be immediately detected by the other server.) By designing a version of the socialist millionaires' protocol without fairness, moreover, we are able to achieve much better efficiency than these previous solutions, which at best require a number of exponentiations linear in the bit-length of the compared values. Our protocol effectively involves only constant overhead. It is more efficient than the protocol in [18], the only other solution to the socialist millionaires' problem that we know of in the literature with constant overhead.

Note that in this protocol, the client need perform no cryptographic computation, but just a single (addition) operation in $\mathcal{H}$. (The client performs some cryptographic computation to establish secure connections with Blue and Red in our system, but this may occur via low-exponent RSA encryption – as in SSL – and thus involves just a small number of modular multiplications.) Moreover, once the client has submitted a sharing, it need have no further involvement in the authentication process. Red and Blue together decide on the correctness of the password submitted for authentication. Given a successful authentication, they can then perform any of a range of functions providing privileges for the user: Each server can send a share of a key for decrypting the user's downloadable credentials, or two servers can jointly issue a signed assertion that the user has authenticated, etc.

2.1 Protocol details

As we have already described the simple sharing protocols employed by the client in our system for registration and authentication, we present in detail only the protocol used by the servers to test the equality $Q_{red} = Q_{blue}$. We assume a private, mutually authenticated channel between the two servers. Should the initiating server (Blue) try to establish multiple, concurrent authentication sessions for a given user account, the other server (Red) will refuse. (In particular, in Figure 1, Red will reject the initiation of a session in which the first flow specifies the same user account as for a previously established, active authentication session.) Alternative approaches permitting concurrent login requests for a single account are possible, but more complicated. If Blue initiates an authentication request with Red for a user U for which Red has received no corresponding authentication request from the user, then Red, after some appropriate delay, will reject the authentication.

Let $Q_{blue,U}$ denote the current share combination that Blue wishes to test for user U, and $Q_{red,U}$ the analogous Red-server share combination for user U. In this and any subsequently described protocols in this paper, if a server fails to validate any mathematical relation denoted by $\overset{?}{=}$, $\overset{?}{\neq}$, $\overset{?}{>}$, or $\overset{?}{\in}$, it determines that a protocol failure has taken place; in this case, the authentication session is terminated and the corresponding authentication request rejected.

We let $\in_R$ denote uniform random selection from a set. We indicate by square brackets those computations that Red may perform prior to protocol initiation by Blue, if desired. Our password-equality testing protocol is depicted in Figure 1. We use subscripts red or 1 to denote values computed or received by Red and $blue$ or 0 for those of Blue. We alternate between these forms of notation for visual clarity. We let h denote a one-way hash function (modeled in our security analysis by a random oracle). In the case where a system may include multiple Blue and/or Red servers, the hash input should include the server identities as well. We let $||$ denote string concatenation.

For the sake of simplicity, we fix a particular group $\mathcal{G}$ for our protocol description here. In particular, we consider $\mathcal{G}$ to be the prime subgroup of order q in Z_p, for prime $p = 2q + 1$. Use of this particular group is reflected in our protocol by: (1) Use of even exponents e_0 and e_1 to ensure group membership in manipulation of transmitted values, and (2) Membership checks over $\{2, \ldots, p-2\}$. For other choices of group, group membership of manipulated values may be ensured by other means. All arithmetic here is performed $\bmod p$.

Implementation choices: A typical choice for p, and that adopted in our system, is a 1024-bit prime. Recall that we select $\mathcal{G}$ to be a subgroup of prime order q for $p = 2q + 1$. For $\mathcal{H}$, we simply select the group consisting of, e.g., 160-bit strings, with XOR as the group operator. We note that a wide variety of other choices is possible. For example, one may achieve greater efficiency by selecting shorter exponents e_0 and e_1, e.g., 160 bits. This yields a system that we hypothesize may be proven secure in the generic model for $\mathcal{G}$, but whose security has not been analyzed in the literature. One might also use smaller subgroups, in which case group-membership testing involves fair computational expense. Alternatively, other choices of group $\mathcal{G}$ may yield higher efficiency. One possibility, for example, is selection of $\mathcal{G}$ as an appropriate group over an elliptic curve. This yields much better efficiency for the exponentiation operations, and also has an efficient test of group membership.

Security: In brief, security in our model states that an adversary with active control of one of the two servers and an arbitrary set of users can do essentially no better in attacking the accounts of honest users than random, on-line guessing. Attacks involving such guessing may be contained by means of standard throttling mechanisms, e.g., shutting down a given account after three incorrect guesses. Of course, our scheme does not offer any robustness against simple server failures. This may be achieved straightforwardly through duplication of the Red and Blue servers. We also assume fully private server-authenticated channels between the client and the two servers. In this model, and with the random-oracle assumption [2] on the hash function, we claim that the security of our core cryptographic algorithm for equal-

$$
\begin{array}{lcl}
\textbf{BLUE server} & & \textbf{RED server} \\
 & & \\
e_0 \in_R \{2, 4, \ldots, q-1\} & & [e_1 \in_R \{2, 4, \ldots, q-1\}] \\
A = w(Q_{blue,U}) & & [Y_1' = g^{e_1}] \\
Y_0 = A g^{e_0} & & \\
 & \xrightarrow{Y_0, U} & \\
 & & B = w(Q_{red,U}) \\
 & & Y_1 = B Y_1' \\
 & & Z_{red} = (Y_0/B)^{e_1} \\
 & & Z_{red} \overset{?}{\in} \{2, \ldots, p-2\} \\
 & & H_{red} = h(Z_{red} \parallel Y_0 \parallel Y_1 \parallel U) \\
 & \xleftarrow{Y_1, H_{red}} & \\
Z_{blue} = (Y_1/A)^{e_0} & & \\
Z_{blue} \overset{?}{\in} \{2, \ldots, p-2\} & & \\
H_{blue} = h(Z_{blue} \parallel H_{red}) & & \\
 & \xrightarrow{H_{blue}} & \\
H_{red} \overset{?}{=} h(Z_{blue} \parallel Y_0 \parallel Y_1 \parallel U) & & H_{blue} \overset{?}{=} h(Z_{red} \parallel H_{red})
\end{array}
$$

Figure 1: Password-equality testing protocol

ity testing may be reduced to the computational Diffie-Hellman assumption on the group $\mathcal{G}$.

3 Architectural Motivation and Overview

The security of our equality-testing protocol in our system depends upon the inability of an attacker to compromise both Red and Blue. Heterogeneity in server configurations is thus an important practical security consideration here. At the simplest level, the Red and Blue servers may run different operating systems, thereby broadening the range of technical obstacles confronting the would-be attacker. A further possible step in this direction would be to situate Red and Blue within different organizations, with the hope of minimizing the risk of insider or social-engineering attacks.

The distribution of security across organizations also provides an appealing model of risk management in which legal and financial responsibility for compromise can be flexibly allocated. We can view this as a form of *privacy outsourcing*, in which one server (say, Blue) is operated by a service provider and the other (say, Red) is operated by what we may refer to as a *privacy provider*. The privacy provider might be an organization with specialized expertise that is willing to assume the primary burden of security maintenance and likewise to assume a large portion of the legal and financial liability associated with privacy breaches.

For a service provider to adopt this approach in a way appealing to a large and potentially mobile user population, there are two salient requirements:

- **Universality**: There should be no need for clients to install special-purpose software. In particular, clients should be able to interface with the system by way of standard browser components such as Java and HTML.
- **Pseudonymity**: Red, i.e., the privacy provider, should be unable to gain explicit access to the user names associated with accounts. At a minimum, clients should be able to interact with this server *pseudonymously*, i.e., by way of identifiers unlinkable with true account names or IP addresses. This provides a technical obstacle to abuse of account information on the part of the operator of Red. It is also useful to employ pseudonyms in this way so as to limit exposure of account identifiers in case of compromise of Red.

The requirement of universality in the service-provider model argues that the software in our system, while perhaps installed on some clients as a browser plug-in or standalone executable, should also be available in the form of a Java applet. This applet is dispensed by Blue in our system (although it could be dispensed elsewhere). The applet contains the software to execute our basic two-server protocol, and also contains a public key for Red. This public key serves to establish a private channel from the client to Red via Blue.

Distribution of such applets by Blue raises an immediate concern: Blue might serve bad applets. In particular, an attacker that has compromised Blue in an active fashion can cause that server to distribute applets that contain a false public key for Red – or indeed that do not even run the intended protocol. As we have already explained, the problem of trusted software is present even for SPAKA protocols, given the need of roaming clients to install such software on the fly. Applets or other software may be digitally signed, but most users are unlikely to understand how to employ browser-provided verification tools to check the correctness of the associated code-signing certificate. Rather, we make two observations on this score. First, active compromise of core components of Blue is likely to be much harder than passive compromise. Some hope may be placed in so-called "tripwire" tools that are designed specifically to detect hostile code modification. Additionally, the task of an attacker in compromising Blue in this way is harder than active compromise in traditional cryptographic settings, in the following sense: Any observer can in principle detect the compromise by inspecting applets. Thus, the privacy provider might periodically initiate authentication requests with Blue to monitor its integrity. Another complementary approach is for Red to distribute to interested clients a piece of software that verifies the hash of code served by Blue.

The pseudonymity requirement, particularly the notion that Red should not learn the IP addresses of clients, suggests that the privacy provider should operate Red as a back-end server, i.e., a server that only interacts with other servers, not clients. This is the server-configuration that we adopt in our system. In particular, the client in our system communicates with the Red server via an encrypted tunnel established using the public key for Red. There are in fact several other compelling reasons to operate Red as a back-end server:

- **Engineering simplicity:** Deployment of Red as a back-end server permits the client to establish a direct connection with only a single server, the normal mode of use for most services on the Internet. A service provider may maintain a single front-end server and treat Red as an external, supporting Web service.

- **System isolation:** In the outsourcing model, the major burden of liability and security is on Red, and the privacy provider is the primary source of security expertise. Hence it is desirable to isolate Red from open communication on the Internet, restricting its interaction instead to one or more Blue servers exclusively via the protocols in our system, effectively creating a kind of strong, application-layer firewall. This imparts to the system as a whole a higher level of security than if both servers were directly exposed.

- **Mitigation of denial-of-service attacks:** Isolation of Red as a back-end server is also helpful in minimizing the exposure of Red to denial-of-service attacks, which the operator of Blue, having better familiarity with its own user base, is better equipped to handle.

A serious concern does arise in conjunction with the pseudonymity requirement. Blue must identify a given user name U to Red according to a fixed pseudonym V. One possible attack, then, is for Red to pose as a client authenticating under identifier U, and then see which associated pseudonym V Blue asserts. Red thereby learns the linkage between U and V. There is effectively no good (and practical) way to defend against this type of attack. Instead, we rely on social factors to forestall this such behavior on the part of Red, namely: (1) As the service provider, it is Blue that will hold the list of account names, so that these may be difficult for Red to accumulate *en bloc*; and (2) Given the risk of damaged reputation, Red should be averse to mounting an attack against pseudonyms. Of course, use of pseudonyms is still beneficial in that passive compromise of Red will not reveal true account identifiers.

4 False Pseudonym and Replay Attacks

Our equality-testing protocol is designed to provide security against corruption of one of the two servers in a single session. Other security problems arise, however, as a result of the use of pseudonyms in our system and also from the need for multiple invocations of the equality-testing protocol. In particular, additional protocols are needed in our system to defend against what we refer to as *false-pseudonym* and *replay* attacks.

4.1 The false-pseudonym problem

The possibility of a massive on-line *false-pseudonym* attack by a corrupted Blue server represents a serious potential vulnerability. In particular, Blue might create an arbitrarily large set of fictitious accounts on Red under false pseudonyms $\tilde{V}_1, \tilde{V}_2, \ldots$, with a dictionary of passwords of its choice. It can then replay genuine authentication requests for a given user's account against the pseudonyms $\tilde{V}_1, \tilde{V}_2, \ldots$. By repeating replays until it achieves a match, Blue thereby learns the secret information for account U. This attack is particularly serious in that it might proceed indefinitely without detection. Behavior of this kind would not be publicly detectable, in contrast for instance to the attack involving distribution of bad applets.

To address this problem, we require that Blue use a secret, static, one-way function f to map user identifiers to pseudonyms. Blue (in conjunction with the client) then *proves* to Red for every authentication request that it is asserting the correct pseudonym. One challenge in designing a protocol employing this proof strategy is that the client cannot be permitted to learn the pseudonym for any account until after it has authenticated. Otherwise, Red can learn pseudonyms by posing as a client. A second challenge – as in all of our protocols – is to design a proof protocol that it lightweight for Red, Blue, and especially for the client. We demonstrate a protocol here that requires no intensive cryptographic computation by the client – just a modular inversion and a handful of symmetric-key computations. (With a small modification, the modular inversion can be replaced with a modular multiplication, leading to even lower computational requirements.)

The basis of our protocol is a one-way function of the form $f_x : m \to m^x$ in a group $\mathcal{G}'$ of order q' over which the Decision Diffie-Hellman problem is hard. This choice of one-way function has two especially desirable features for our protocol construction: (1) It is possible to prove statements about the application of f by employing standard non-interactive zero-knowledge proofs on discrete logarithms; and (2) The function f_x has a multiplicative homomorphism, namely $f_x(a)f_x(b) = f_x(ab)$. Naturally, so as to keep f_x secret, the value x is an integer held privately by Blue. We let g denote a generator and $y = g^x$ denote a corresponding public key distributed to Red.

To render the proof protocol lightweight for the client, we adopt a *cut-and-choose* proof strategy. The idea is that a client identifier U is represented as a group element in $\mathcal{G}'$. The client computes a random, multiplicative splitting of U over $\mathcal{G}'$ into shares U_0 and U_1; thus $U = U_0U_1$. The client also computes commitments to U_0 and U_1, and transmits these to Red. Blue computes V by application of f_x to each of the shares U_0 and U_1. In particular, Blue sends to Red the values $V_0 = f_x(U_0)$ and $V_1 = f_x(U_1)$. Observe that by the multiplicative homomorphism on f_x, Red can then compute the pseudonym $V = f_x(U) = f_x(U_0)f_x(U_1) = V_0V_1$. To prove that this pseudonym V is the right one, Red sends a random challenge bit b to Blue. Blue then reveals U_b and proves that $V_b = f_x(U_b)$, i.e., that the discrete logarithms $\log_g(y)$ and $\log_{U_b}(V_b)$ are equal. The probability that a cheating Blue is detected in this protocol is 1/2. (More precisely, it is extremely close to 1/2 under the right cryptographic assumptions.) Thus, if Blue attempts to mount a pseudonym attack, say, 80 times, this will be detected by Red with overwhelming probability. Our use of this cut-and-choose protocol, therefore, renders the threat of such an attack by Blue much smaller than the threat of a rogue client that submits password guesses. Meanwhile, Red learns only random, independent shares of U, not U itself. We defer further details of the pseudonym protocol and its integration with the other protocols in our system to the full paper.

4.2 The replay-attack problem

In the case where the client communicates directly with Red and Blue via private channels, an adversary in control of either server does not have the capability of mounting a replay attack, as it has access to only the messages sent by the client to one of the servers. In our implementation here, however, where the client communicates with Red via Blue, this is no longer the case. Indeed, without some additional mechanism to ensure the freshness of the share sent by the client to Red, an adversary in control of Blue can mount a replay attack simply by repeating all communications from the client. While the adversary would not learn the password P this way, she could falsely persuade Red that a successful authentication has just occurred; this would enable the adversary to initiate some joint operation on the user's behalf without the user's presence.

A simple countermeasure is to employ timestamps. In particular, Blue may transmit the current time to the client. Along with its other information, the client then transmits a MAC of this timestamp under R', the share provided to Red. Provided that Red stores for each user account the latest timestamp accompanying a successful authentication, Red can verify the freshness of a share it receives by verifying that the associated timestamp post-dates the latest one stored for the account. A drawback of this approach, however, is the engineering complexity introduced by time-synchronization requirements.

An alternative, therefore, is to employ counters. Blue and Red can maintain for each account a counter logging the number of successful authentication attempts. Blue, then, provides the most recent counter value to the client at the time of authentication, and the client transmits a MAC under R' of this counter as an integrity-protected verifier to be forwarded to Red. Using this verifier, Red can confirm the freshness of the associated authentication request.

The drawback to this type of use of counters is its leakage of account information. An attacker posing as a given user can learn the counter value for the user's account from Blue, and thus gather information about her login patterns. An adversary controlling Red can moreover harvest such counter values without initiating authentication attempts and thus without the risk of alerting Blue to potentially suspicious behavior. By matching these counter values against those stored by Red, such an adversary can correlate pseudonyms with user identities.

It is important, therefore, not to transmit plaintext counter values to clients. Instead, Blue can transmit to an authenticating client a *commitment* ζ of the counter value γ for the claimed user identity [6, 25]. The client then furnishes to Red (via Blue) a MAC under R' of ζ. On initiating an authentication request, Blue provides to Red the counter value γ and a witness ρ associated with ζ; together, these two pieces of data decommit the associated counter value. In this way, the client does not learn γ, but the integrity of the binding between the counter value γ and a given authentication request is preserved. A hash function represents an efficient way to realize the commitment scheme, and is computationally binding and hiding under the random oracle assumption. In particular, Blue may commit γ as $\zeta = h(\gamma \parallel \rho)$, where the witness ρ is a random bitstring of length l, for an appropriate security parameter l (e.g., $l \geq 160$). To decommit, Blue provides γ and ρ, enabling Red to verify the correctness of ζ. This protocol is depicted in Figure 2. The flows of this protocol are overlaid on those of the full authentication protocol in the our system. Let $\gamma_{blue,U}$ denote the counter value stored for the account of the user U attempting to authenticate and $\gamma_{red,U}$ be the corresponding counter value as stored by Red. At the conclusion of this protocol, on successful authentication by the user, Red sets $\gamma_{red,U} = \gamma_{blue,U}$ and Blue increments $\gamma_{blue,U}$ by one.

5 Implementation

In this section we describe the details of our implementation. In particular we describe how the protocols outlined in this paper are integrated, how the servers are configured, and what the components are of the software programs running on each server.

The goal of the prototype we describe here, following the configuration described in section 3, is to improve the security of a standard Web page login procedure. This prototype augments a Web application on a Blue server with the addition of a special authentication library. While a typical Web site would store or look up a password in its database, the enhanced server makes a function call to this library via an API. In order to fulfill these requests, the library makes requests to the Red server, which acts as a privacy provider.

The first component of the prototype is a small Web site on a Blue server with a user registration and login procedure. The second component is a function library which implements all protocol steps to be executed on Blue. The Web application accesses these functions according to our API. The third component is a Red server, which processes and responds to the requests coming from the Blue server, initiated by our library.

In general terms, the message flow may be understood in terms of the client machine making requests to the Web application on Blue. The Web application makes requests to our library on the same server, which in turn makes requests to the Red server. The client never needs to communicate directly with the the Red server. All messages, including encrypted messages destined for the Red server, are sent via Web requests to the Blue server. This encapsulation makes the user experience transparent; the user is not directly aware of the Red server.

Given that the desired client interface is a standard Web browser, we chose to use HTTP for all message communication. We set up the two servers with the Linux operating system (8.0), including the Apache Web server, configured to support CGI (Common Gateway Interface), and SSL. Using HTTPS automatically provides secure channels between Red and Blue, and between Blue and the client. We note that a different transport mechanism between Red and Blue could have been chosen. However, by formatting Blue's requests to Red as well formatted text messages over HTTP, Red effectively acts as a private "Web service", thereby increasing interoperability and design flexibility.

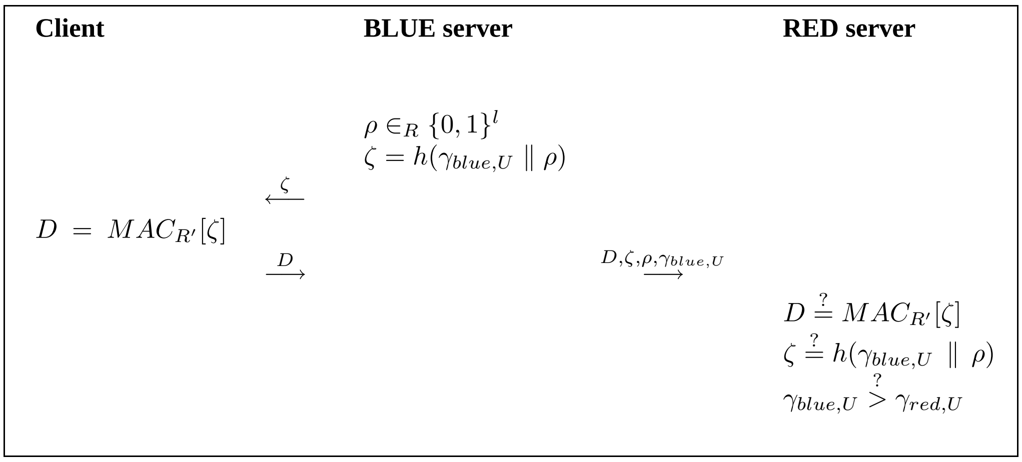
Client
BLUE server
RED server
$\rho \in_R \{0,1\}^l$
$\zeta = h(\gamma_{blue,U} \parallel \rho)$
$\xleftarrow{\zeta}$
$D = MAC_{R'}[\zeta]$
$\xrightarrow{D}$
$\xrightarrow{D,\zeta,\rho,\gamma_{blue,U}}$
$D \overset{?}{=} MAC_{R'}[\zeta]$
$\zeta \overset{?}{=} h(\gamma_{blue,U} \parallel \rho)$
$\gamma_{blue,U} \overset{?}{>} \gamma_{red,U}$

Figure 2: Replay countermeasure protocol

To serve the Web content and perform authentication protocol, two programs, compiled from C/C++ source code, were installed in the proper Web-server directories. To store permanent and transient user data each of the servers uses an SQL database. Standard libraries are used to interact with the database, to format messages, and to produce HTML.

These building blocks provide the secure on-line message communication, the data storage, and the basic cryptography needed for a variety of protocols. We now come to the most interesting part, the logic particular to our system. Upon receipt of any message, the main processing function in either Red or Blue checks first that it is a well formed message corresponding to a specific step of our protocol. If so, it executes the protocol step, the components of which are described in this paper.

To complete the description of the prototype we just need to describe how the equality checking protocol, and replay countermeasure protocols are integrated, and how the client makes well formed requests to the Web application on Blue without any addition of software to the Web browser.

This is accomplished as follows. A user wishing to authenticate first obtains from the Blue server an HTML form and a signed Java applet. The form has an input field for the user name and password and hidden fields containing a random salt value and Red's public key. On the client machine, the user enters her user name and password into the input field in the HTML form. When the user clicks the "submit" button, the applet hashes the salt with the password, splits the result into shares, and encrypts one share under Red's public key. The encrypted share, the other share, and a replay-prevention value are formatted into a composite message to be sent to Blue as an HTTP request. Of course, the user does not see this processing, nor the other message components prepared by the applet. The user is just served a confirmation or rejection Web page which indicates whether or not the authentication attempt has succeeded.

We now explain further how the two client requests trigger the remaining protocol steps described in this paper. We first remark that our actual implementation also accommodates authentication via life questions, the approach briefly mentioned in the introduction as an alternate authentication mechanism for users with forgotten passwords or unavailable hardware authentication tokens. This extra functionality entails a few technical details. For one, the Java Applet contains the text of personal questions posed to the user and also contains code to split the multiple answers in parallel. Since the questions may vary by user, the user name may be requested first in a separate form. The system permits decisions regarding the success of authentication via life questions to occur on a threshold basis. For example, an administrator may configure the system to authenticate users successfully if they answer any three out of five life questions correctly. The system need not reveal to the user which answers are incorrect if the authentication as a whole is unsuccessful. (The servers individually, however, will learn the number which questions were answered correctly.) Another feature worth remarking on is that Red does not learn or store the questions posed to individual users.

In Figure 3, we show how the protocol components for equality testing and replay-prevention are overlaid to form a the composite authentication protocol. For simplicity of presentation, we focus here on the basic case of authentication via a single password, not use of life questions. All messages in this figure use notation consistent with that in Figures 1 and 2. Additionally, we denote message components for the Java applet and final response to the client with $Applet$ and $\{PASS/FAIL\}$ and encryption under the Red server's public key by E_{Red}. Since we do not include details of our pseudonym-related protocols in this paper, we omit that part of our protocol from our description here. For brevity, we omit the description of certain secondary details such as data representation, and choice of cryptographic primitives here, but we do note that care must be taken to correctly handle session timeouts and the locking out of a user after too many failed login attempts.

Our prototype implements the client as a moderate-sized Java applet, running to about 2000 source lines. The applet can process a password in about 80 milliseconds on a 700MHz Pentium running Windows XP. Note that this does not include the time required to download and initialize the applet.

The prototype Blue server consists of a set of CGI programs written in C++ and C. The prototype code for the Blue server consists of about 10,000 lines of source, not including the communications and database libraries. The prototype Red server is a Linux application built from about 5,000 lines of C and C++ source code. The Blue and Red servers used in the prototype (two 500 MHz Pentium III systems running SUSE Linux) can verify about 10 passwords per second. The prototype was not optimized for efficiency; we expect that significantly better performance should be possible.

We also remark that a version of the protocol also runs under the Windows operating systems, and that our API is now being implemented as a set of Java classes that may be embedded in Servlets or Enterprise Java Beans. The encapsulation of this functionality within a API is particularly useful, having made its realization language independent, and convenient to integrate with a variety of Web applications.

6 Conclusion

As the protocol designs and prototyping experience presented in this paper demonstrate, our system is a highly practical approach to the problem of secure authentication via weak secrets. By employing two servers, the system is able to offer considerably more protection of sensitive user data than any single-server approach could permit. At the same time, the system architecture avoids many of the conceptual and design complexities of multi-server cryptographic protocols – SPAKA schemes and others – described in the literature.

There are a wealth of other multi-server cryptographic protocols that can doubtless be brought to practical fruition in the two-server framework that our system presents. Some examples include:

- credential download, where encrypted credentials are stored on one server and the decryption key is stored on the other;
- threshold digital signing (see, e.g., [22, 23] for discussion of a special two-party protocol);
- joint authorization (and auditing) of self-service user administration operations such as password reset;
- privacy-preserving information delivery as in, e.g., [9, 10, 20].

Our hope is that our system may serve as a useful springboard for the practical realization of these and related concepts from the security literature.

References

[1] PKCS (Public-Key Cryptography Standard) #5 v2.0, 2002. Available at www.rsasecurity.com/rsalabs/pkcs.

[2] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *1st ACM Conference on Computer and Communications Security*, pages 62–73. ACM Press, 1993.

[3] S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84. IEEE Press, 1992.

[4] S. M. Bellovin and M. Merritt. Augmented encrypted key exchange. In *1st ACM Conference on Computer and Communications Security*, pages 244–250. ACM Press, 1993.

Mes #	**Client** $\leftarrow (SSL) \rightarrow$ **BLUE**	**BLUE** $\leftarrow (SSL) \rightarrow$ **RED**
1	$\xrightarrow{U}$	
2	$\xleftarrow{Applet,\zeta}$	
3	$\xrightarrow{U,f(P')+R',E_{red}(R'),D}$	
4		$\xrightarrow{E_{red}(R'),Y_0,U,D,\zeta,\rho,\gamma_{blue,U}}$
5		$\xleftarrow{Y_1,H_{red}}$
6	$\xrightarrow{H_{blue}}$	
7	$\xleftarrow{\{PASS/FAIL\}}$	

Figure 3: Integrated message flow (without pseudonym protocol)

[5] D. Bleichenbacher and P. Q. Nguyen. Noisy polynomial interpolation and noisy Chinese remaindering. In B. Preneel, editor, *EUROCRYPT 2000*, pages 53–69. Springer-Verlag, 2000. LNCS no. 1807.

[6] M. Blum. Coin flipping by telephone. In *Proceedings of 24th IEEE Compcon*, pages 133–137, 1982.

[7] D. Boneh. The Decision Diffie-Hellman problem. In *ANTS '98*, pages 48–63. Springer-Verlag, 1998. LNCS no. 1423.

[8] F. Boudot, B. Schoenmakers, and J. Traoré. A fair and efficient solution to the socialist millionaires' problem. *Discrete Applied Mathematics*, 111(1-2):23–36, 2001.

[9] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, 1981.

[10] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *IEEE Symposium on Foundations of Computer Science*, pages 41–50, 1995.

[11] C. Ellison, C. Hall, R. Milbert, and B. Schneier. Protecting secret keys with personal entropy. *Journal of Future Generation Computer Systems*, 16(4):311–318, February 2000.

[12] R. Fagin, M. Naor, and P. Winkler. Comparing information without leaking it. *CACM*, 39(5):77–85, May 1996.

[13] W. Ford and B. S. Kaliski Jr. Server-assisted generation of a strong secret from a password. In *Proceedings of the IEEE 9th International Workshop on Enabling Technologies (WETICE)*. IEEE Press, 2000.

[14] A.O. Freier, P. Karlton, and P.C. Kocher. The SSL protocol version 3.0, November 1996. URL: www.netscape.com/eng/ssl3/draft302.txt.

[15] N. Frykholm and A. Juels. Error-tolerant password recovery. In P. Samarati, editor, *8th ACM Conference on Computer and Communications Security*, pages 1–9. ACM Press, 2001.

[16] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31:469–472, 1985.

[17] D. P. Jablon. Research papers on strong password authentication, 2002. URL: www.integritysciences.com/links.html.

[18] M. Jakobsson and A. Juels. Mix and match: Secure function evaluation via ciphertexts. In T. Okamoto, editor, *ASIACRYPT 2000*, pages 162–177. Springer-Verlag, 2000. LNCS no. 1976.

[19] M. Jakobsson and M. Yung. Proving without knowing: On oblivious, agnostic, and blindfolded provers. In *CRYPTO '96*, pages 186–200, 1996. LNCS no. 1109.

[20] A. Juels. Targeted advertising... and privacy too. In D. Naccache, editor, *RSA-CT '01*, pages 408–424, 2001. LNCS no. 2020.

[21] Daniel V. Klein. "Foiling the cracker" – A survey of, and improvements to, password security. In *Proceedings of the 2nd USENIX Workshop on Security*, pages 5–14, Summer 1990.

[22] P. Mackenzie and M. Reiter. Cryptographic servers for capture-resilient devices. In S. Jajodia, editor, *9th ACM Conference on Computer and Communications Security*, pages 10–19. ACM Press, 2001.

[23] P. Mackenzie and M. Reiter. Two-party generation of DSA signatures. In J. Kilian, editor, *CRYPTO 2001*, pages 137–154. Springer-Verlag, 2001. LNCS no. 2139.

[24] P. Mackenzie, T. Shrimpton, and M. Jakobsson. Threshold password-authenticated key exchange. In M. Yung, editor, *CRYPTO 2002*, pages 385–400. Springer-Verlag, 2002. LNCS no. 2442.

[25] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.