SSH Password Authentication Using Secure Function Evaluation

Louis Kruger, Somesh Jha, Matt Fredrikson
University of Wisconsin

Vitaly Shmatikov
University of Texas

Abstract

Over the years, SSH has evolved from a secure alternative to telnet into a robust and extensible layered protocol that can serve as a secure transport layer for applications that need strong cryptographic security. Unfortunately, interactive password-based authentication remains one of the most popular choices for common SSH deployments, leaving opportunity for an active malicious adversary to either learn the client's password, or impersonate the server or client. *Secure Password and Key Authentication* (SPAKA) protocols provide mutual authentication, and are designed to be resilient to these attacks. However, current SPAKA protocols impose specific requirements on the environments in which they are used, making them impractical for many common, widespread applications such as SSH. In this paper, we present a *practical* protocol that provides equivalent guarantees to existing SPAKA protocols, but is suitable as a "drop-in" authentication module in existing deployments. To accomplish this, we use secure function evaluation (SFE) to compare the password credentials between the client and server, embedding computation of the hash function into the comparison protocol. We have implemented an SSH client and server that use this authentication protocol, and released an open-source version of the software that is freely available for download.

1 Introduction

Originally designed as a secure alternative to telnet, SSH has since evolved into a layered protocol that serves as the secure transport layer over which many other protocols execute. This functionality has simplified the task of providing sophisticated cryptographic security to a number of applications that need it. Unfortunately, it has also encouraged SSH deployment in settings where strong authentication mechanisms are not available, or worse yet, take a backseat to more convenient measures such as interactive password login. This poses a significant problem, as many casual SSH users may assume that the mere presence of SSH guarantees security, unaware of the risks associated with password authentication and the improper use of public key cryptography [38]. For example, in Section 3 we describe a man-in-the-middle attack on SSH password authentication.

To address the attack described in Section 3.1. we design a practical, yet cryptographically secure protocol for password-based authentication and key establishment in SSH. Even though we use our protocol in the context of SSH, our technique can be applied to any scenario where password-based authentication is applicable. An implementation of our protocol is available at `http://www.cs.wisc.edu/~lpkruger/ssh`. Our protocol satisfies three important design principles.

(1) Compatible with legacy infrastructure. Our protocol is compatible with existing password authentication infrastructures. It does not require any changes to *legacy servers* beyond upgrading the SSH software and is thus deployable in common settings. The use of cryptographic hash databases to store passwords is common practice on both Unix and Windows systems [35]. Typical Linux systems (current versions of Ubuntu [36], RedHat [34], and Debian [11]) typically use either the MD5 or SHA-512 hash function, with salts and iterated rounds for added security against offline brute-force

attacks. Current Windows versions use a proprietary technology known as the NT Hash [32], but the principle is identical. Our protocol is specifically designed to support the use of hash functions to store passwords.

By contrast, other solutions for password-authenticated key exchange require users to re-generate passwords, which greatly limits their deployability. They cannot be installed on legacy servers with large existing user bases. Some also require additional information to be stored on the server or assume the existence of public-key infrastructure (PKI).

(2) Does not decrease security of password storage. At the very least, the password authentication mechanism should not provide weaker security guarantees than the current system, in which users' passwords are stored on the server in hashed form. If the server stores passwords in the clear, a compromise of the server will reveal the passwords of all users. Even without an external compromise, a malicious server operator may impersonate a user in other authentication domains.

Our protocol takes as inputs the password from the user and the hashed password from the client (it is essential that the user's input into the protocol is the actual password and not a hash; otherwise, a malicious server operator could impersonate the user). Therefore, from the viewpoint of password security, it is as strong as existing solutions, while providing significantly more protection against man-in-the-middle attacks.

(3) Enables derivation of a secure, shared cryptographic key. Our protocol enables the user and the server to derive a shared cryptographic key(s) which can be used to protect their subsequent communications. The key remains secure (*i.e.*, indistinguishable from random) even in the presence of a malicious man-in-the-middle adversary. Unlike existing methods for password authentication in SSH, our protocol does not require the user to check the validity of the server's public key by manually verifying its fingerprint (we argue that this requirement is largely ignored in practical deployment scenarios).

Against an active adversary, the protocol is as secure as can be hoped for in the case of password-based authentication. It does not leak any information except the outcome of an authentication attempt, *i.e.*, for any given password, the adversary can check whether the password is correct. Brute-force password-cracking remains feasible, but every attempt requires executing an instance of the protocol.

Exploiting the special features of password authentication. Our protocol uses Yao's "garbled circuits" protocol for secure function evaluation (SFE) as a basic building block. SFE is used to compute the hash of the SSH client's password and compare it for equality with the hash value provided by the SSH server.

Yao's original protocol is only secure against passive or semi-honest adversaries [27, 40], *i.e.*, under the assumption that all participants faithfully follow the protocol. This model is clearly unsuitable for SSH, which must be secure even if one of the participants maliciously deviates from the protocol specification. This includes the case when a malicious SSH client—who constructs the garbled circuits in our protocol—deliberately creates a faulty circuit in an attempt to learn the server's input into the protocol. For example, the client may put malformed ciphertexts into the rows of the garbled truth table which will only be evaluated when a certain input bit from the server is equal to "1," and correct ciphertexts into the rows which will be evaluated when this bit is equal to "0." By observing whether the server's evaluation of this circuit fails or not, the malicious client can learn the value of the bit in question. The malicious client may also submit a circuit which computes something other than the hash-and-check-for-equality function required by SSH authentication.

Yao's protocol can be modified to achieve security against malicious participants—either via cut-and-choose techniques [28, 37], or via special-purpose zero-knowledge proofs [22] which enable the server to verify that the circuit is well-formed—but the resulting constructions, while more efficient than generic transformations, are still too expensive for practical use.

Our SFE-based construction in this paper exploits the special structure of the authentication problem in a fundamental way. The purpose of the password authentication subprotocol in SSH is to compute a single bit for the client: whether the hash of the password submitted by the client is equal to the value submitted by the server or not. The standard cut-and-choose construction for SFE in the malicious model requires that the server evaluate several garbled circuits submitted by the client and the majority of them must be correct [28]. In the context of password authentication for SSH, it is sufficient that a *single* circuit is correct. Even if all but one circuits evaluated by the server are faulty, a malicious client does not learn any more than he would have been learned simply by submitting a wrong password.

Our key observation is that to prevent a malicious client from authenticating without the correct password, it is sufficient for the SSH server to either (a) detect that one of the circuits submitted by the client is incorrect, or (b) evaluate at least one correct circuit. In other words, the SSH server either detects the client's misbehavior or rejects the client's candidate password because its hash does not match the server's value. In either case, authentication attempt is rejected.

We prove the security of our protocol against malicious clients in a (modified) *covert* model of secure computation [5, 20]. Security in the covert model guarantees that any deviation from the protocol will be detected with a high probability. In our proof, instead, we show that, with high probability, either the deviation is detected, or the protocol computes the same value as it would have computed had the client behaved correctly. Security in this model can be achieved at a lower cost than "standard" security against malicious participants, enabling significant performance gains for our implementation viz. off-the-shelf SFE.

Security of an honest client against a malicious SSH server follows directly from the security of the underlying oblivious transfer (OT) protocol against malicious choosers, since the server's input into the protocol is limited to his acting as a chooser in the OT executed as part of Yao's protocol. While the server can always perform a denial-of-service attack by refusing to communicate the result of authentication to the client, this is inevitable in any client-server architecture.

The protocol is also secure against replay attacks since a man-in-the-middle eavesdropper on an instance of the protocol does not learn anything about the client's input (password), server's input (password hash), or the shared key established by the client and the server.

SPAKA protocols. Bellovin and Merritt pioneered a class of protocols that use the client password as a shared secret for mutual authentication [7]. These protocols, commonly referred to as SPAKA (*Secure Password and Key Authentication*) or PAKE (*Password Authenticated Key Exchange*), are resistant to the password compromise scenario described above, *even when the client is communicating directly with a malicious impersonator.* Furthermore, these protocols alert the client to the presence of an impersonator, allowing the SSH user to curtail further communications in high-risk situations. However, existing SPAKA protocols impose requirements that make them difficult to deploy in most settings, especially when legacy servers and legacy hashed-password files are involved (see Section 2).

In this paper, we present the first password-based authentication and key establishment protocol to satisfy, in the context of SSH, the three design principles listed above. We show that the secure password storage and the secure key establishment requirements can be achieved by comparing the authentication credentials of the user and the server using *secure function evaluation* (SFE) [39], in a legacy-compatible manner.

The main insight that enables backward compatibility with existing infrastructures is that SFE gives the protocol complete flexibility to compute arbitrary hash functions while performing authentication. This makes our protocol suitable as a "drop-in" authentication module in most legacy environments, requiring only that the server and client software be updated to use the new protocol.

Organization of the paper. In Section 2, we discuss related work, and explain why existing SPAKA protocols are not suitable for SSH in terms of the three requirements previously listed. In Section 3, we present a technical overview of our problem setting, as well as our proposed solution. In Section 4, we

	Legacy compatibility	*Secure password storage*	*Mutual authentication*
EKE	***X***	***X***	✓
AEKE	***X***	✓	✓
Ω-Method	***X***	✓	✓
Multiple-Server	***X***	✓	✓

Figure 1: A comparison of existing SPAKA protocols. The protocols listed are EKE [7], AEKE [8], Ω-method [17], and Multiple-Server [12]. There are a number of protocols in the literature similar in nature to EKE and AEKE; these are referenced in the text but left out of this table for the sake of clarity.

describe our contributions in further detail, and in Section 5 we evaluate our implementation.

2 Related Work

Secure Password and Key Authentication (SPAKA) is a class of authentication protocols designed to guarantee confidentiality of secrets even against active malicious adversaries. There have been several SPAKA protocols proposed in the literature with varying properties. The first such protocol was described by Bellovin and Merritt [7]. This protocol, *Encrypted Key Exchange* (EKE), was designed to allow two parties to communicate using a weak secret, such as an *easily memorable* password. The authors observed that a standard symmetric cryptosystem keyed on the weak secret does not provide strong security, and instead proposed the use of a temporary asymmetric key pair to exchange a stronger shared symmetric key to use for the full duration of the session. The key element that makes this secure is that the bit strings which represent keys in several asymmetric schemes are essentially random, and so difficult to verify in a brute-force attack on the key exchange messages. Thus, this protocol provides a way of accomplishing basic mutual authentication, and satisfies requirement *(3)* from our list. However, it does not satisfy *(1)* or *(2)*, as common deployments store only a hash of the client's password on the server. A number of subsequent protocols have the same properties in terms of our requirements [1, 3, 6, 9, 16, 29, 43], including a recent scheme by Abdalla *et al.* proved secure in the universal composability model [2], and one by Katz *et al.* in the standard model [24, 25], which was later extended to additional cryptographic assumptions by Gennaro and Lindell [14, 15].

Bellovin and Merritt developed *Augmented Encrypted Key Exchange* (AEKE) [8] to address these shortcomings by relaxing the requirement that the server possess knowledge of the client's password in clear text. Rather, they allow the server to possess only a hash of the password, which preserves the secrecy of the client's password in a scenario where the password database is compromised by an adversary. To prevent impersonation of the client by such an adversary, the protocol uses primitives that allow one party to verify that the other has knowledge of both the password and its hash, and proposed two schemes for selecting these primitives. The first scheme uses a class of *commutative one-way hash functions*. However, there are no known families of commutative hash functions that possess the information-hiding properties required to guarantee the security of the protocol, making this scheme of theoretical interest only. The second scheme defines the hashed password stored by the server as the public key in a digital signature scheme. Then, to prove knowledge of the original password, the client signs the session key with the corresponding private key. For our purposes, AEKE implemented with this scheme satisfies requirements *(2)* and *(3)* from our list. However, correctly storing the public key on the server may require substantial changes to infrastructure, and violates requirement *(1)*.

More recently, Gentry *et al.* [17] proposed the Ω-method for converting an arbitrary PAKE protocol that is *not* resilient to server compromise into one that is secure in such a scenario. They proved the security of their method in the universal composability framework. However, all known feasible implementations of their method require the server to store additional information, namely a public/private key pair with

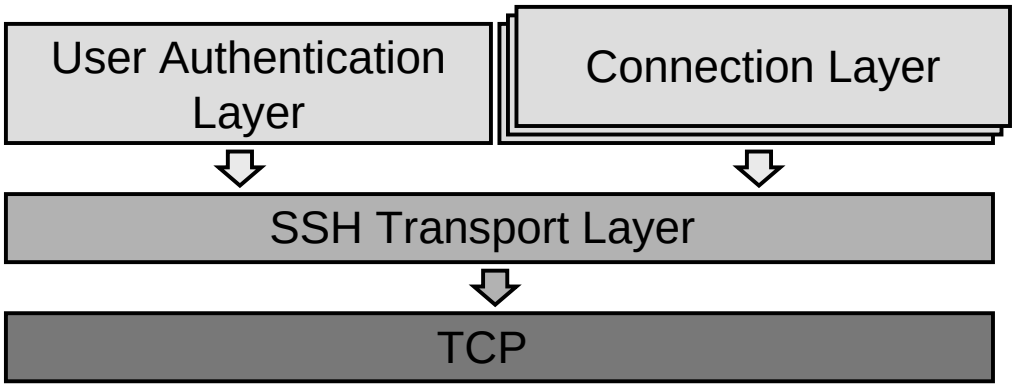

Figure 2: The hierarchy of layers within the SSH protocol.

the secret key encrypted. Thus, applying this method to one of the previously described protocols will result in a set of implementation constraints basically equivalent to AEKE [8], and ultimately fail to satisfy requirement *(1)*.

Ford and Kaliski [12] presented a SPAKA protocol that protects the secrecy of the client's password against server compromise by distributing it among many servers. When the client authenticates, it interacts with each server to establish a set of strong secrets, after which the servers collaborate to validate the client's identity. A number of subsequent protocols adopt this basic functionality; MacKenzie *et al.* generalize the protocol to a threshold setting [30], and Brainard *et al.* present a lightweight protocol that reduces the computation load on the client [10]. While these protocols satisfy our security requirements (*(2)* and *(3)*), they have the obvious drawback of requiring a specific server-side architecture that may not be common in many settings, and thus fail to satisfy requirement *(1)*.

Further details on each type of protocol described in this section, as well as specific explanations of why they are not suited for the case of SSH authentication, are discussed in the appendix.

3 SSH Protocol Overview

The SSH protocol enables secure network services, including remote login and traffic tunneling, over insecure networks such as the Internet [41]. The functionality of the protocol is partitioned into three layers, with each layer defined in terms of messages from the layer beneath it. This hierarchy is depicted in Figure 2; each layer plays a distinct role [41]:

- The *transport layer* provides privacy, integrity, and server authentication for the user authentication protocols, as well as the application connection protocols, running on top of it. In short, it provides the layers above it with a plaintext interface for sending encrypted packets reliably over the network.
- The *user authentication layer* authenticates the client to the server. In keeping with the modular design of the protocol, this layer is extensible to a number of authentication mechanisms. The specification for this layer includes public key, password, and host-based authentication sub-protocols [42]. However, the majority of deployments use password-based authentication for its convenience and simplicity.
- The *connection layer* multiplexes many distinct communication channels over the SSH transport layer. Several channel types have been defined for various applications, including terminal shell channels for remote login, and traffic forwarding channels for encrypted tunnels.

A typical SSH session proceeds by working through these layers in sequence: first, the SSH transport layer is established, after which the user is able to securely authenticate to the server, and finally the application-specific connections are initiated over the transport layer.

Session Initialization: To establish the SSH transport layer, the server and client must *(1)* perform a key exchange to establish a shared secret that is used to encrypt future communications, and *(2)* validate the server's key, to prevent a man-in-the-middle attack. The SSH specification includes a single Diffie-Hellman group for key exchange [41], although later proposals have extended this layer to allow new Diffie-Hellman groups to be added as needed [13]. As described in Section 1, the host's key is validated by querying the user. Thus, this layer is responsible for the vulnerability described in Section 1.

User Authentication: When the SSH transport layer has been established, the client and server have a secure channel over which they can communicate, and the server has supposedly been authenticated to the client. However, most applications require the client to authenticate to the server. The user authentication layer handles this in an extensible way, by defining a set of messages that can be used to relay general authentication data. The specification describes several mechanisms for authentication, including the username/password method familiar to all users of SSH, as well as public key-based authentication [42]. However, as long as the server and client software can agree on an authentication method, it is straightforward to extend this layer to use new mechanisms that provide better security. For example, Yang and Shieh proposed the use of smart cards for authentication [38], which has subsequently been implemented in at least one SSH software package [33]. Our proposed protocol fits into the SSH protocol in this layer.

When password authentication is used, the protocol proceeds as follows (depicted in Figure 3):

1. The client sends to the server a message of type `SSH_MSG_USERAUTH_REQUEST`, containing the username and password given by the user.
2. Based on the contents of the `SSH_MSG_USERAUTH_REQUEST`, the server responds to the client in one of two ways:
   - If password authentication is disallowed, or the username/password combination supplied is incorrect, then the server responds with an `SSH_MSG_USERAUTH_FAILURE` message.
   - If password authentication is allowed, and the username/password combination is valid, then the server responds with an `SSH_MSG_USERAUTH_SUCCESS` message.
3. If the server sends `SSH_MSG_USERAUTH_SUCCESS`, then the client has successfully authenticated and may begin requesting services. Otherwise, the protocol terminates.

3.1 Man-in-the-middle attack on conventional SSH password authentication

When SSH password authentication is used, the client and server first negotiate an encrypted tunnel, over which the client sends the password for verification. If an attacker was somehow able to eavesdrop on this encrypted tunnel, the password itself would be revealed to the attacker, who would then be able to impersonate the client at will. A man-in-the-middle attack on the encrypted tunnel, if succesful, would allow such a password interception. To prevent such an attack, SSH relies on *host keys* [42] to authenticate the server. However, host keys alone do not entirely solve the problem, as it is necessary to authenticate each server key when a session is initiated. Under certain circumstances, an attacker may still be able to mount a successful attack. Figure 4 depicts this situation when the Diffie-Hellman key exchange is used:

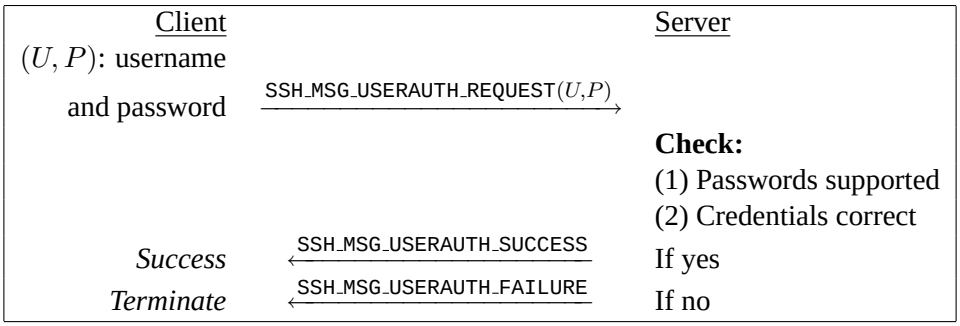

Figure 3: The SSH user authentication sub-protocol.

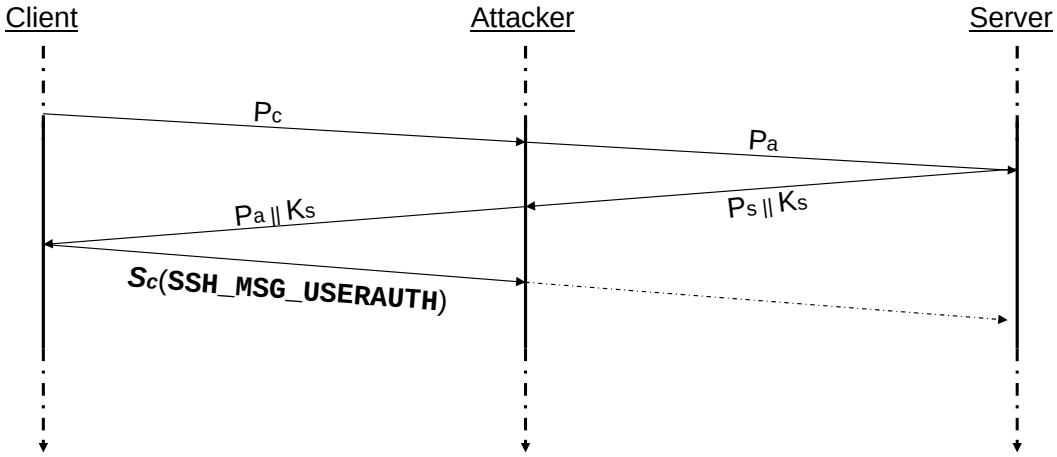

Figure 4: Man-in-the-middle attack on the SSH authentication protocol using Diffie-Hellman key exchange as described in the SSH-2.0 specification. Using this attack, an adversary can learn the client's password, eavesdrop on all communications between client and server, and impersonate the server.

1. After the client and server agree on a key exchange protocol, the client attempts to send the public exponentiated integer P_c to the client.
2. The attacker intercepts P_c, replaces it with a value known to him, P_a, and sends it to the server.
3. The server sends his public integer, along with a host key that is supposed to prove his identity: $P_s \parallel K_s$.
4. The attacker intercepts $P_s \parallel K_s$ and sends $P_a \parallel K_a$ to the server.
5. Using the exchanged public keys, the attacker constructs separate shared secrets S_c and S_s with the client and server, to be used for further communications.
6. The client's software hashes the key that it receives, K_a, and checks a local keystore to see if the hash is recognized. If the client does not have the real servers's public key K_a in his keystore, then the client software asks the user to verify the server key's authenticity:

```
The authenticity of host 'server (1.2.3.4)' can't be
established. RSA key fingerprint is
3f:76:22:43:c2:03:b9:71:b0:31:ce:87:37:45:cb:02.
Are you sure you want to continue connecting (yes/no)?
```

On the other hand, even if the user knows the correct server key, he may assume the key has changed for a non-malicious reason, such as a software upgrade, and allow the connection to proceed.

7. The user validates the authenticity of the key based on its hexadecimal fingerprint, thereby mistakenly asserting that the attacker is the authentic server.

8. The client attempts to send S_c(SSH_MSG_USERAUTH) to the server, containing login credentials encrypted with the shared secret S_c. In this case, the credentials consist of a username and password.

9. The attacker receives S_c(SSH_MSG_USERAUTH), and is able to decrypt it to read the password in clear text.

Critical to the success of this attack is that the user validates the authenticity of the attacker's public key as the server's, in step *(7), an action which we assert is highly probable*. Any OpenSSH user is familiar with the message displayed in step *(6)* – according to the SSH protocol RFC, the fingerprint "...can easily be verified by using telephone or other external communication channels." [41] Not surprisingly, recent research has indicated that one can expect the average user to simply *click through* this dialog without going to such trouble [4], thus accepting the attacker's key; this undermines the very purpose of presenting host key fingerprints to the user. Although the designers of the SSH protocol were aware of this problem when they released the specification, they assumed that widespread future PKI deployment would make it unimportant [41].

4 Protocols

We need a protocol that provides the following functionality: given the client's input x (presumably the password) and the server's input y (presumably hash of the password), the two parties would like to jointly compute whether $H(x) = y$, for some hash function H. In other words, we need to a protocol for the following functionality:

$$(x, y) \longmapsto (\delta(H(x), y), \delta(H(x), y))$$

Where $\delta(a, b)$ is equal to 1 if $a = b$, otherwise it is 0. The first question we must answer is: under which model should our protocol be secure? In the semi-honest model, the adversaries follow the correct protocol, but might try to infer additional information from the messages exchanged during the protocol. The classic protocol presented by Yao [40] can be used to produce a protocol for our problem that is secure in the semi-honest model. An extensive treatment of Yao's protocol along with a proof of correctness is given in [27]. However, the semi-honest model is not suitable in our context, because SSH is frequently used over wide-area networks (WAN) where we cannot expect the parties to obey the semi-honest model.

In the malicious model the adversaries may behave arbitrarily, i.e., lie about their inputs, abort, or not follow the instructions of the protocol. Given a protocol that is secure in the semi-honest model, the protocol can be transformed into a protocol secure in the malicious model [18, 19]. However, the resulting protocols are very inefficient. Lindell and Pinkas [28] present a more efficient protocol that is based on the informal cut-and-choose technique for the two-party case that is secure in the malicious model. However, their protocol is also too slow for our purposes. Protocols that are secure in the semi-honest model are efficient

but not secure in our context. On the other hand, protocols that are secure in the malicious model are too inefficient to be useful in our context.

The adversary model we use in this paper is inspired by the *covert model* of Aumann and Lindell [5]. In the covert model, any attempt to cheat by the malicious protocol participant $\mathcal{A}$ is detected by the honest parties with probability at least ϵ. In our model, we demonstrate that if a malicious SSH client cheats, then, with high probability, the SSH server either detects the cheating, or computes exactly the same result it would have computed if the client had not cheated.

4.1 Protocol 1: Strawman Protocol

Recall that the client (C) has the password $x = P$ and the server (S) has the hash of the password $y = H(P)$. The protocol works as follows:

- Client hashes the password and obtains $x' = H(x)$.
- Client and server use protocol that is secure in the covert model from [5, Section 6.2] for the function $f(x', y) = \delta(x', y)$.
- After the protocol the client and server know whether their inputs are the same.

The protocol given in [5] has several parameters. Note that the Naor-Pinkas OT protocol provides unconditional security for the server. Therefore, we have the client send the garbled circuits to the server, so the server acts as chooser in the underlying OT-protocol. Moreover, if each bit of the server's input is split into m bits and the cut-and-choose is performed over l circuits, then the protocol is ϵ-deterrent where $\epsilon = (1 - \frac{1}{l})(1 - 2^{-m+1})$.

The straw man protocol has a vulnerability which defeats the entire purpose of storing passwords on the server in the hashed form. To successfully authenticate as a client in this protocol, it is sufficient to know only the hash of the password rather than the password itself. First, this means that if the server is compromised, then the attacker can impersonate any client whose password was stored on the compromised server, even if these passwords were stored in a hashed form. Second, if the server is malicious, then it can impersonate any client who successfully authenticates to it.

Nevertheless, the straw man protocol may be useful in certain environments with relaxed security requirements.

4.2 Protocol 2: Main Protocol

We now present our main protocol. Recall that in the SSH context, there are two parties in the protocol: party 1 (client) has input x and party 2 (server) has input y. They want to jointly compute the functionality $(x, y) \longmapsto (\delta(H(x) = y), \delta(H(x) = y))$ where $\delta_{H(x)=y}$ is equal to 1 if $H(x) = y$; otherwise it is 0. If client gets output of 1, it means that client was authenticated by the server. The reader should interpret x as the password and y as the hash of the password (in other words, the client should only be able to successfully authenticate if he knows the password whose hash matches what the server has). The key idea is that the hash function H is included in the functionality, which makes our protocol resilient against malicious servers impersonating clients (see Section 4.1): knowledge of the password hash is *not* sufficient to authenticate as the client.

The following protocol description assumes that the reader is familiar with the basics of secure-function evaluation (such as garbled circuit construction and oblivious transfer).

- **(Step 1)** Client creates l garbled circuits $C_1, \cdots, C_l$ of the for $\delta(H(x), y)$. Let server's input $y = y_1 \cdots y_m$ be m bits. The wire keys corresponding to the j-bit of server's input for the i-th garbled circuit C_i is denoted by $k^0_{i,j}$ and $k^1_{i,j}$. Client sends circuits $C_1, C_2, \cdots, C_l$ to the server.

- **(Step 2)** Client and server execute the OT_1^2 protocol m times. In the j-th instance of OT_1^2 the client acts as a sender with inputs $k_{1,j}^0 \| k_{2,j}^0 \| \cdots \| k_{l,j}^0$ and $k_{1,j}^1 \| k_{2,j}^1 \| \cdots \| k_{l,j}^1$ and the server acts the chooser with input y_j (the j-th bit of the input). Notice that concatenating the keys prevents the server from learning keys corresponding to different bits, e.g., server cannot learn keys $k_{1,j}^0$ and $k_{2,j}^1$.

- **(Step 3)** Server chooses a random set $S \subseteq \{1, 2, \cdots, l\}$ and sends S to the client.

- **(Step 4)** Client reveals wire keys for circuits C_j such that $j \in S$ to the server (we call this step opening the circuits C_j such that $j \in S$). Client also provides wire keys for its input x for circuits C_j, $j \notin S$.

- **(Step 5)** If the circuits C_j ($j \in S$) are not well-formed (the circuits do not compute $\delta(H(x), y)$ or the keys are not consistent with what was sent in step 2), the server sends 0 to the client. Server computes C_j ($j \notin S$) and obtains answers o_j ($j \notin S$). Server sends $\bigwedge_{j \notin S} o_j$ to the client.

It is clear that if both client and server are honest, then the client will successfully authenticate to the server if and only if it has a password x whose hash $H(x)$ is equal to the input of the server y. Some of the important features of our protocol are:

- The server learns the wire keys corresponding to *one input*. In other words, it is not possible for the server to evaluate circuit C_i on input x_1 and circuit C_j ($j \neq i$) on a different input x_2. This is the rationale behind concatenating the wire keys in step 2 of the protocol. It ensures that a malicious server cannot enter more than one password hash into the computation in an attempt to learn the client's input.

- Assume that out of the l garbled circuits $C_1, \cdots, C_l$ the circuits with index $j \in B$ (where $B \subseteq \{1, 2, \cdots, l\}$) are not valid. The only way the client's cheating is not detected is if B is a subset of $\neg S$ (the complement of S), *i.e.,* all invalid circuits are in the unopened set.

- The server's response to the client is computed as the logical AND of the outputs of all unopened circuits (Step 5). This exploits the essential feature of password authentication, namely, that the client receives a single bit from the server.

  As long as the password submitted by the client is wrong and at least *one* of the unopened circuits is correct (*i.e.,* it correctly computes the hash of the client's input and compares it for equality with the server's input), the server's answer will be 0: "failed authentication attempt." Therefore, a malicious client does not learn anything by submitting invalid circuits, unless *all* unopened circuits are invalid. The outputs of the invalid circuits are effectively hidden from the client by the output of a single correct circuit. By contrast, the generic construction for the malicious model [28] requires that the majority of unopened circuits be correct to prevent information leakages.

  If the client's input is the correct password (*i.e.,* its hash is equal to the server's input), then the client can compute the server's input on his own. Therefore, the client cannot possibly learn anything from the protocol execution, except a single bit confirming that his input is correct.

  Observe that a malicious client who does not know the password will successfully authenticate (*i.e.,* receive bit 1 rather than 0 as his output of the protocol) if and only if *all* unopened circuits are invalid, *i.e.,* the set of invalid circuits B is exactly ¬S. Because S is chosen randomly, the probability of this event is 2^{-l}.

- There is no consistency check on the client's inputs. A malicious client may input different passwords into different circuits. Recall that with high probability, the unopened set contains at least one correct circuit. Clearly, submitting a wrong password to a correct circuit will result in authentication failure.

Therefore, the only situation in which the client will authenticate is if he consistently submits the correct password to every correctly formed, unopened circuit. We argue that this is equivalent to knowing the correct password in the first place, *i.e.*, submitting inconsistent inputs does not offer any benefits to a malicious client.

If the client submits inconsistent inputs and authentication fails, the client does not learn which of the inputs were correct and which were incorrect. Therefore, the client is still limited to a single password per authentication attempt.

We formally argue the protocol preserves the privacy of both parties' inputs.

Client's privacy: Assume that the client is honest and the server is controlled by an adversary $\mathcal{A}$. $\mathcal{A}$'s view consists of the l garbled circuits $C_1, \cdots, C_l$, messages received during the m OT_1^2 protocols, all keys for C_j ($j \in S$), where $S \subseteq \{1, 2, \cdots, l\}$ is chosen by $\mathcal{A}$), and keys corresponding to the client's input x for circuits C_j ($j \notin S$). Assume that views corresponding to the m OT_1^2 protocols only reveal the secrets corresponding to the input y of $\mathcal{A}$ (let the $\mathcal{A}$ input be $y = y_1 \cdots y_m$ then the server learns $k_{j,k}^{y_k}$ $1 \leq j \leq l$ and $1 \leq k \leq m$). This follows from the privacy of the underlying oblivious-transfer protocol. For example, if one uses the Naor-Pinkas oblivious-transfer protocol, then we have the information-theoretic security for the server. Assuming that the encryption scheme used to construct the garbled circuits is semantically secure, revealing the wire keys for circuits C_j ($j \in S$) does not reveal any information about the client's input. Consider the circuits C_j ($j \notin S$). Server can evaluate this circuit on (x, y) but learns nothing else. Consider an ensemble of garbled circuits C'_j ($j \notin S$), where C'_j computes the constant function $\delta(H(x), y)$.[1]. If the encryption-scheme is semantically secure, then $\mathcal{A}$ cannot distinguish between circuits C_j and C'_j (for $j \notin S$). Essentially $\mathcal{A}$ only learns whether hash of client's password is equal to its input and nothing else.

Server's privacy: First we give an informal sketch for server's privacy. Assume that server's input is y. We now show that in order for a malicious client who does not know a password x' such that $H(x') = y$, his probability of successful authentication to the server (or impersonation) is no better than 2^{-l}. In other words, *the probability that a malicious client who does not know the pre-image of the server's input successfully impersonating a honest client is bounded by* 2^{-l}. The use of even modestly large value of parameter l will make it more likely that the adversary can simply guess the password than to break the protocol. We consider this sufficient, but if desired, extremely large values of l can be used to make the probability of breaking the protocol negligible, with a performance penalty linear in the value of l.

In particular, we show that that the protocol is secure unless the client perfectly guesses the subset S of the l circuits that the server will choose and prepares the encrypted circuits accordingly.

It is sufficient to assume that the malicious client does not know the correct password. If the client knows the password then there is no information to be learned from the server that he does not already possess. There is no useful purpose to cheating the protocol, since he would achieve the desired outcome by executing it faithfully. In this case we simply do not care if the client cheats because he only hurts himself.

There are three possible cases:

- **(Case 1)** The client includes an invalid circuit in S. The server will detect this in and reject the authentication in step 5.
- **(Case 2)** Every circuit in S is correct and $\neg S$ (which denotes the complement of S) includes at least one valid circuit. When the server evaluates this circuit, it will evaluate to 0 and the server will reject the authentication. Recall that if a circuit is valid, it will evaluate to 0 on the inputs of the client and server because the client does not know the pre-image of the server's input.

[1]We assume C'_j is constructed from the same encryption scheme that was used to construct $C_1, \cdots, C_j$.

- **(Case 3)** Every circuit in S is correct and every circuit in $\neg S$ is incorrect. We make no claims of correctness about this case. In particular, the client could have made every circuit in $\neg S$ evaluate to 1, in which case the server would accept the impersonating client as authentic.

Since the server chooses the subset S uniformly from the space of proper subsets, the probability of case 3 happening is 2^{-l}.

Assume that the server is honest and the client is malicious. The client is controlled by an adversary $\mathcal{A}$. We construct a simulator Sim which works in the ideal model. Sim acts as the server for $\mathcal{A}$.

- $\mathcal{A}$ sends l copies of the garbled circuits $C_1, \cdots, C_l$ to Sim.
- Sim acts as TP for $\mathcal{A}$ for the m oblivious transfer protocols. Sim knows the inputs of $\mathcal{A}$ (which in the case of the honest client's are the wire keys corresponding to the server's inputs).
- Sim chooses a random set $S_1 \subseteq \{1, 2, \cdots, l\}$ and sends it to $\mathcal{A}$.
- $\mathcal{A}$ sends all the wire keys corresponding to circuits C_j $(j \in S_1)$.
- Sim rewinds $\mathcal{A}$, and sends the complement of S_1 to $\mathcal{A}$. $\mathcal{A}$ sends all the wire keys corresponding to circuits C_j $(j \notin S_1)$. Note that after this step Sim knows the wire keys corresponding to all the garbled circuits $C_1, \cdots, C_l$.
- Sim rewinds $\mathcal{A}$, picks a random set $S \subseteq \{1, 2, \cdots, l\}$, and sends it to $\mathcal{A}$. $\mathcal{A}$ sends wire keys corresponding to the circuits C_j $(j \in S)$.
- $\mathcal{A}$ provides the wire keys for all circuits C_j $(j \notin S)$. Note that since Sim knows the wire keys for all the garbled circuits, it can now construct $\mathcal{A}$'s inputs x_j to circuits C_j $(j \notin S)$. If the inputs are inconsistent (*i.e.*, not all equal to the same value), Sim sends 0 to $\mathcal{A}$. If all inputs x_j $(j \notin S)$ are equal to x, then Sim sends x to the TP. If TP returns 1 (which means that $\mathcal{A}$ knew the pre-image of server's input), then Sim sends 1 to $\mathcal{A}$ (which essentially means that the malicious client was authenticated). If TP returns 0, we proceed to the next step.
- Sim checks the validity of all the circuits C_j $(j \in S)$. If any of these circuits is found to be invalid, then Sim sends 0 to $\mathcal{A}$. Otherwise Sim sends 1 to $\mathcal{A}$.

Assume that $\mathcal{A}$ does not know the pre-image corresponding to the server's input. Suppose the inputs x_j $(j \notin S)$ are not equal. In this case, $\mathcal{A}$ receives 0 from Sim. We argue that in the real model $\mathcal{A}$ would receive 1 only if it knows the pre-image corresponding to the server's input (a contradiction). The only way $\mathcal{A}$ receives a 1 if all of his inputs into correctly formed, unopened circuits are pre-images of the server's input, which means $\mathcal{A}$ knew the pre-image of the server's input to begin with, contradicting our assumption. Hence the views of $\mathcal{A}$ in the ideal and real world are the same when the inputs x_j $(j \notin S)$ are not equal, unless all opened circuits are valid *and* all of the unopened circuits are invalid (the probability of this event is 2^{-l}).

Now assume that inputs x_j $(j \notin S)$ are all equal to x. Let E_1 be the event that a honest server denies authentication to $\mathcal{A}$ in the real model, and E_2 be the event that Sim denies authentication to $\mathcal{A}$ in the ideal model. Recall that denying authentication is tantamount to $\mathcal{A}$ receiving 0. It is easy to see that if E_1 and E_2 are true, then the view of $\mathcal{A}$ in the real model is indistinguishable from the view of $\mathcal{A}$ in the ideal model. The probability of event $E_1 \wedge E_2$ not happening is bounded by 2^{-l+1}.

We conclude that if the client does not know the pre-image of the server's input, then the probability that the view of $\mathcal{A}$ in the real model is indistinguishable from the view of $\mathcal{A}$ in the ideal model with probability atleast $1 - 2^{-l+1}$. In other words, conditioned on the event that the malicious client does not know the

pre-image of the server's input, the probability of the simulator failing is bounded by 2^{-l+1}. This model is very similar to the "failed simulation" model given by Aumann and Lindell [5, Section 3.2].

Other attacks: Observing an instance of our protocol yields no information that will be useful in subsequent instances of the protocol (*e.g.*, it does not reveal the parties' inputs). Therefore, our construction is secure against replay attacks.

Consider a replay attack in which a malicious part (Eve) captures all messages exchanged between the client and the server. Now Eve replays message from an old session to impersonate the client. This attack is thwarted because various steps in the protocol use fresh, randomly generated values. For example, in each instance of the Naor-Pinkas oblivious-transfer protocol, the chooser (in our case the SSH server) generates a random value k and sends g^k or $\frac{C}{g^k}$ (where g is generator of the underlying group and C is an element in the group). Therefore, observing the server's inputs to the protocol does not reveal the password hash, nor any information that can be used in subsequent sessions.

4.3 Protocol 3: Adding key establishment

In the context of SSH, client and server need to compare their inputs and also establish a session key if the comparison between their inputs is successful. The protocol is an easy extension of our main protocol.

- Client picks a random key K.
- Client and server execute a variation of the main protocol that computes the following functionality:

$$((x, K), y) \;\mapsto\; (\textbf{if } (H(x) = y) \;\textbf{ then }\; (K, K) \textbf{ else } (\perp, \perp))$$

A malicious client may pick key K that is not truly random. However, if the client is malicious, it can unencrypt and forward its messages to an adversary regardless of this. On the other hand, suppose the server is malicious. If the malicious server knows the hash $H(x)$ of the client's input x, then it knows the session key K. In this case, the server can again forward the unencrypted messages to the adversary. If the server does not know $H(x)$, then it cannot obtain the session key. In other words, adding the extra functionality of distributing session keys does not affect the security of protocol 2 (which only compares $H(x)$ and y).

Note on the oblivious transfer protocol. In our implementation, we use the oblivious transfer (OT) protocol by Naor-Pinkas [31]. This protocol provides information-theoretic security for the chooser (SSH server in our implementation) and computational security, based on the Diffie-Hellman assumption, for the sender (SSH client). This OT protocol is a good choice for the SSH environment due to its efficiency. As an alternative, we could have implemented our system using a fully simulatable oblivious transfer protocol such as, for example, the new Diffie-Hellman-based OT protocol by Hazay and Lindell [21] whose computational complexity is similar to the Naor-Pinkas protocol. We leave an implementation of this OT protocol as part of our system to future work.

5 Evaluation

We modified an existing open-source SSH client and server to use each of the protocols described in Section 4, and took several performance measurements to evaluate the feasibility of our approach. Our findings can be summarized as follows:

- Protocols that are secure in the semi-honest model, which is only secure against passive adversaries, can be executed very quickly.

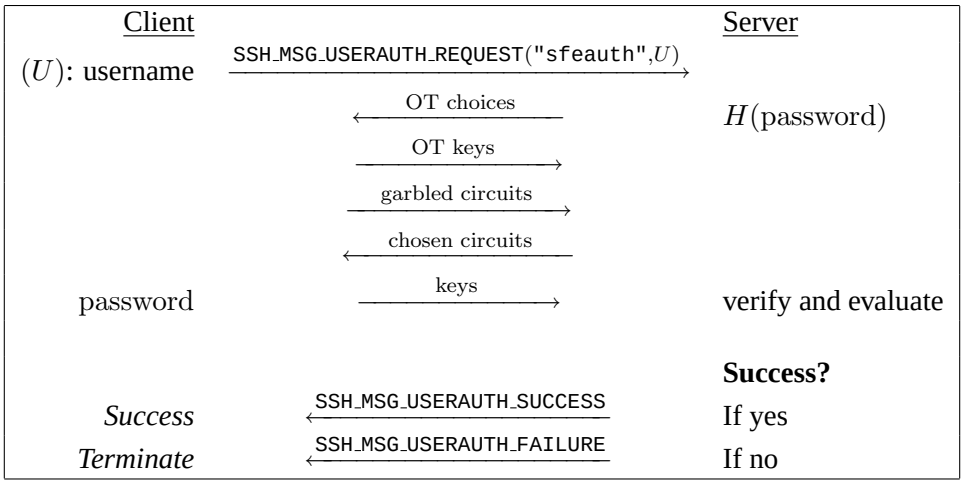

Figure 5: The SFE user authentication protocol 2.

- Making the protocols secure against active adversaries increases authentication time substantially, depending on the size of the authentication circuit and the security parameter. For example, calculating an MD5 hash using 90 circuits increases the authentication time from around 2 seconds to 12 seconds. Although this may seem tedious for some users, we achieve a high level of security with only modest delays on inexpensive modern hardware. We conclude that the technique can achieve a favorable balance between efficient practicality and high security.

- Due the simplicity of private equality testing, Protocol 1 (the Straw man protocol) can run extremely quickly even under the covert model at the expense of resisting impersonation by an adversary who has gained knowledge of a user's password hash. If this security requirement can be relaxed (for example, in an environment where passwords are stored on the server using an identity hash, i.e. in plaintext) protocol 1 could be a useful as high-speed authentication protocol.

5.1 Implementation

We implemented the protocols by modifying the Dropbear 0.52 SSH client and server to support a new authentication protocol, to which we assigned the name "sfeauth" in the SSH authentication protocol namespace. The scheme used for incorporating the protocols into the SSH protocol is shown in Figure 5. The Yao sub-protocol is conducted through the encrypted SSH tunnel using a reserved message which we dubbed `SSH_MSG_USERAUTH_SFEMSG`. If at any time the server detects a cheating attempt by the client, the server fails the authentication and terminates the protocol.

The MD5 and private equality circuits were implemented using a prototype circuit compiler we developed first described in [23], which also contains an embeddable implementation of the Yao "garbled circuit" protocol. The protocol was extended using the techniques due to Lindell and Pinkas [28] to add resistance to malicious parties in the covert model. The implementation also uses the oblivious transfer protocol due to Naor and Pinkas [31]. All of the Yao protocol and authentication code was written in C++ and integrated with the Dropbear SSH client and server.

5.1.1 Optimizations

To improve performance of the protocols, we introduced several optimizations to our implementation.

1. The client computes the garbled circuits used in the authentication protocol in advance of the online protocol. By precomputing and storing garbled circuits, the time spent in this CPU intensive step is

# Circuits	*Online Time (seconds)*	*Ratio to Semi-Honest*	*Probability of attack success*
1 (*Semi-Honest*)	0.52	1.0	100%
30	3.99	7.7	1.86×10^{-7}%
60	7.86	15.2	1.73×10^{-16}%
90	12.35	23.8	1.62×10^{-25}%
120	16.46	31.8	1.50×10^{-34}%
150	20.57	39.6	1.40×10^{-43}%

Table 1: Wall-clock performance and security guarantees for the optimized protocol in both semi-honest and covert settings. All times are given in seconds.

removed from the user's perceived wait time to login to a server.

2. We implemented an optimization described by Goyal *et al.* [20]. In constructing the garbled Yao circuits, the client generates a set of seeds for a cryptographically-secure pseudorandom number generator. The circuits are garbled using this PRNG, with one seed per circuit, and hashes of the circuits are sent in place of the whole circuits. After the server has chosen which circuits to evaluate, the seeds for the non-chosen circuits are revealed to the server, who then uses the PRNG to reconstruct the garbled circuit and verify the hash values, and only the circuits to be evaluated are transferred in full. This saves many megabytes of wire communication, improving the overall protocol performance.

The prototype SSH client and server, as well as further documentation, can be downloaded from our project website.[2]

5.2 Experiments

We conducted several usage experiments to measure the performance of the authentication, and determine its feasibility in real settings. Note that the semi-honest version is not secure for real-world usage where the possibility of active malicious adversaries cannot be ruled out, but the experiment is useful to establish an upper bound for the potential performance with further optimization. The tests were performed over a local network using computers with eight core Intel Xeon processors and 8GB of RAM.

The performance results of our experiments are shown in Table 1. The first row corresponds to the semi-honest version of the protocol, and is the time on which the *ratio to semi-honest* column for other rows is based. The column titled *probability of attack success* refers to the probability of a malicious client successfully convincing the server that he has the proper credentials to authenticate. This calculation is discussed in detail in section 4.2. As our results indicate, the time required to complete the protocol increases linearly as the number of circuits increases, while the security guarantee increases exponentially in this measure. Note that for less than an order of magnitude increase over semi-honest implementation, sufficient security guarantees for many practical settings can be attained. These trends indicate that our technique is suitable for common use in real applications.

We note that the performance is sensitive to available processing power due to the many cryptographic primitives employed. For example, our implementation takes advantage of parallelization on multi-core processors to encrypt multiple circuits in parallel as the server performs its circuit verifications. Due to the independence of the verification of each circuit, parallel scaling can be achieved that is extremely efficient with respect to available processors, potentially allowing high security authentication with minimal delays to clients on server machines with enough processing power.

[2] `http://www.cs.wisc.edu/~lpkruger/ssh`

Overall, we believe that the results we have achieved so far demonstrate the potential of this technique as a practical and secure addition to the body of research in secure password authentication.

6 Conclusion

In this paper, we have addressed the problem of providing secure mutual authentication for the SSH protocol, maintaining full backwards compatibility with existing server/client infrastructures as a primary design principle. Leveraging the unique flexibility of SFE to compute arbitrary hash functions within the protocol, we constructed a protocol that provides an identical set of security guarantees as previous work in the area, while remaining suitable as a "drop-in" authentication module on nearly all existing servers. Furthermore, we demonstrated that a conservative set of optimizations to the protocol made it practical for common use from a performance standpoint. In the future, we intend to study further optimizations to certain parts of the protocol that improve performance without sacrificing security, as well as additional applications that could benefit from our basic technique.

References

[1] Michel Abdalla, Emmanuel Bresson, Olivier Chevassut, Bodo Möller, and David Pointcheval. Provably secure password-based authentication in TLS. In *ASIACCS '06: Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 35–45, New York, NY, USA, 2006. ACM.

[2] Michel Abdalla, Dario Catalano, Céline Chevalier, and David Pointcheval. Efficient two-party password-based key exchange protocols in the uc framework. In *CT-RSA*, pages 335–351, 2008.

[3] Michel Abdalla and David Pointchavel. Simple password-based encrypted key exchange protocols. In *CT-RSA 2005*, pages 191–208. Springer, 2005.

[4] Anne Adams and Martina Angela Sasse. Users are not the enemy. *Commun. ACM*, 42(12):40–46, 1999.

[5] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *TCC*, pages 137–156, 2007.

[6] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In *EUROCRYPT*, pages 139–155, 2000.

[7] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secureagainst dictionary attacks. In *SP '92: Proceedings of the 1992 IEEE Symposium on Security and Privacy*, page 72, Washington, DC, USA, 1992. IEEE Computer Society.

[8] Steven M. Bellovin and Michael Merritt. Augmented encrypted key exchange: a password-based protocol secure against dictionary attacks and password file compromise. In *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security*, pages 244–250, New York, NY, USA, 1993. ACM.

[9] Victor Boyko, Philip D. MacKenzie, and Sarvar Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In *EUROCRYPT*, pages 156–171, 2000.

[10] John Brainard, Ari Juels, Burt Kaliski, and Michael Szydlo. A new two-server approach for authentication with short secrets. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 14–14, Berkeley, CA, USA, 2003. USENIX Association.

[11] Debian reference manual chapter 4 – authentication. `http://www.debian.org/doc/manuals/debian-reference/ch04.en.html`.

[12] W. Ford and Jr. Kaliski, B.S. Server-assisted generation of a strong secret from a password. In *IEEE 9th Internation Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 176–180, 2000.

[13] M. Friedl, N. Provos, and W. Simpson. RFC 4419: Diffie-Hellman group exchange for the secure shell (SSH) transport layer protocol (proposed standard), March 2006.

[14] Rosario Gennaro. Faster and shorter password-authenticated key exchange. In *TCC*, pages 589–606, 2008.

[15] Rosario Gennaro and Yehuda Lindell. A framework for password-based authenticated key exchange. In *EUROCRYPT*, pages 524–543, 2003.

[16] Craig Gentry, Philip D. MacKenzie, and Zulfikar Ramzan. Password authenticated key exchange using hidden smooth subgroups. In *ACM Conference on Computer and Communications Security*, pages 299–309, 2005.

[17] Craig Gentry, Philip D. MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In *CRYPTO: 26th Annual International Cryptology Conference*, pages 142–159, 2006.

[18] O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic applications*. Cambridge University Press, 2004.

[19] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game – a completeness theorem for protocols with honest majority. In *19th STOC*, 1987.

[20] Vipul Goyal, Payman Mohassel, and Adam Smith. Efficient two party and multi party computation against covert adversaries. In *EUROCRYPT*, pages 289–306, 2008.

[21] C. Hazay and Y. Lindell. Efficient oblivious transfer with simulation-based security. 2009.

[22] Stanislaw Jarecki and Vitaly Shmatikov. Efficient two-party secure computation on committed inputs. In *EUROCRYPT*, pages 97–114, 2007.

[23] Somesh Jha, Louis Kruger, and Vitaly Shmatikov. Towards practical privacy for genomic computation. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 216–230, Washington, DC, USA, 2008. IEEE Computer Society.

[24] Jonathan Katz, Rafail Ostrovsky, and Moti Yung. Efficient password-authenticated key exchange using human-memorable passwords. In *EUROCRYPT '01: Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques*, pages 475–494, London, UK, 2001. Springer-Verlag.

[25] Jonathan Katz, Rafail Ostrovsky, and Moti Yung. Forward secrecy in password-only key exchange protocols. In *SCN*, pages 29–44, 2002.

[26] Leslie Lamport. Password authentication with insecure communication. *Commun. ACM*, 24(11):770–772, 1981.

[27] Y. Lindell and B. Pinkas. A proof of security of yao's protocol for two-party computation. *Journal of Cryptology*, 22(2), 2009.

[28] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*, pages 52–78, 2007.

[29] Philip D. MacKenzie, Sarvar Patel, and Ram Swaminathan. Password-authenticated key exchange based on RSA. In *ASIACRYPT*, pages 599–613, 2000.

[30] Philip D. MacKenzie, Thomas Shrimpton, and Markus Jakobsson. Threshold password-authenticated key exchange. In *CRYPTO: 22nd Annual International Cryptology Conference*, pages 385–400, 2002.

[31] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *SIAM Symposium on Discrete Algorithms (SODA)*, 2001.

[32] MSDN security account manager protocol specification. `http://msdn.microsoft.com/en-us/library/cc245506(PROT.13).aspx`.

[33] The OpenSC project. `http://www.opensc-project.org/`.

[34] RedHat enterprise linux 5.4 deployment guide – shadow passwords. `http://www.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/5.4/html/Deployment_Guide/s1-users-groups-shadow-utilities.html`.

[35] Sean Smith and John Marchenisi. *The Craft of System Security*. Addison Wesley, 2008.

[36] Ubuntu manual entry on shadow passwords. `http://www.opensc-project.org`.

[37] David P. Woodruff. Revisiting the efficiency of malicious two-party computation. In *EUROCRYPT*, pages 79–96, 2007.

[38] Wen-Her Yang and Shiuh-Pyng Shieh. Password authentication schemes with smart cards. *Computers & Security*, 18(8):727–733, 1999.

[39] Andrew C. Yao. Protocols for secure computations. In *FOCS*, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

[40] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.

[41] T. Ylonen and C. Lonvick. RFC 4251: The secure shell (SSH) protocol architecture, January 2006.

[42] T. Ylonen and C. Lonvick. RFC 4252: The secure shell SSH authentication protocol, January 2006.

[43] Muxiang Zhang. New approaches to password authenticated key exchange based on rsa. In *ASIACRYPT*, pages 230–244, 2004.

Appendix

EKE

Encrypted Key Exchange (EKE) was developed by Bellovin and Merritt to allow two parties to communicate using a weak secret, such as a low-entropy password [7]. Suppose A and B share such a secret, P, and that A wishes to securely communicate message M to B. Naïvely, A could use a symmetric cipher E with P as the key, and send $E_P(M)$ to B; however, because P is weak, this is subject to brute-force attack by an adversary who wishes to learn M. Instead, EKE uses a temporary asymmetric key pair to exchange a stronger shared symmetric key to use for the duration of the session. The steps of the protocol, simplified for clarity of presentation, are as follows:

1. A generates a random asymmetric key pair (E_A, D_A) and sends the public key encrypted with the weak password, $E_P(E_A)$, to B.

2. B generates a random symmetric key R, and sends $E_P(E_A(R))$ to A.

3. A uses S and D_A to decrypt R, which is used as a strong session key from this point on.

In this protocol, the weak secret P is used only to encrypt E_A and $E_A(R)$. Thus, it is critical that E_A and $E_A(R)$ are essentially random, to thwart brute-force attacks on the keyspace of R.

Central to the correct execution of this protocol is that both A and B have knowledge of P, and can use it to perform symmetric encryption. However, this violates our first and second assumptions, as current servers do not typically store user passwords in clear text for a number of reasons [26].

AEKE

Augmented Encrypted Key Exchange (AEKE) was developed by Bellovin and Merritt to relax the constraint in EKE that the server possess the shared secret P in clear text [8]. Rather, in AEKE, it is assumed that the server possesses knowledge of a secure hash of the password, $H(P)$. The first part of AEKE proceeds exactly as in EKE, with the exception that all uses of P in the protocol are replaced with uses of $H(P)$. However, B must be able to verify that A possesses the true shared secret P, and not merely its hash. To allow this, they conceptualize a new one-way function $F(P, R)$, where R is the strong session key as in EKE, as well as a predicate $T(H(P), F(P, R), R)$. By requiring that T evaluates to *true* if and only if $H(P)$ and $F(P, K)$ are computed using the same shared secret P, they provide a mechanism for B to verify that A knows P. Thus, two additional steps are needed:

4. A sends $R(F(P, R))$ to B. This proves A's ownership of the original shared secret P, as $H(P)$ cannot be used to calculate $F(P, R)$.

5. B decrypts A's message to obtain $F(P, R)$ and accepts the identity of A only if the T predicate evaluates to *true*.

Bellovin and Merritt proposed one practical scheme for instantiating H, F, and T on real systems. By using the public key in a digital signature scheme to define $H(P)$, A can compute $F(P, R)$ by signing R with the corresponding private key. Evaluating T then amounts to verifying the signature sent by A with $H(P)$. However, it is improbable that the hashing methods in use on current servers can be used in this manner, so this algorithm fails to meet our first requirement.

Ω-Method

The Ω-method was developed by Gentry *et al.* as a way of hardening arbitrary password-based key exchange (PAKE) protocols against the event that the server is compromised [17]. In other words, the Ω-method allows any PAKE protocol $\mathcal{P}$ to operate under the assumption that the server stores only a hash of the user's password, $H(P)$. It works as follows:

1. A and B run $\mathcal{P}$ as they would normally, substituting every occurrence of P with $H(P)$. After running $\mathcal{P}$, both parties possess a shared secret key R.

2. B derives a second session key K' from K, and uses it to send an encrypted secret key D_A from an asymmetric key pair (D_A, F_A), to A: $E_{K'}(E_P(D_A))$

3. A receives $E_{K'}(E_P(D_A))$, derives K' from K in a manner identical to B, and decrypts using K' and P to obtain D_A. A then signs a transcript of the protocol using the secret key, and sends it to B: $\mathsf{Sign}_{D_A}(\mathsf{transcript})$. The client then derives the final session key K'' from K, to be used for the remainder of the session.

4. B receives $\sigma = \mathsf{Sign}_{D_A}(\mathsf{transcript})$, and uses F_A to verify it: $\mathsf{Verify}_{F_A}(\mathsf{transcript}, \sigma)$. If it checks out, then B knows that A possesses P, and derives the shared session key K'' to continue.

The basic idea of the Ω-method can be summarized as follows: $\mathcal{P}$ is run as normal, substituting $H(P)$ for P as necessary, but afterward, A must prove knowledge of P such that $H(P)$ is in agreement with the B's authentication records. However, the manner in which the method performs this requires the server to store a non-trivial amount of additional information in the form of an asymmetric key pair, so it fails to meet our first requirement.

Multiple-Server

Ford and Kaliski [12] also consider the problem of using a weak password to bootstrap a secure means of communication, and the risk associated with storing some form of the password database on the server. Their approach consists of protecting passwords from server compromise by distributing trust across multiple servers, thereby eliminating the chance of total password compromise by infiltration of a single server. In their scheme, the client's weak password is used to derive a strong secret by means of multiple *hardening servers*, and combines the result of the individual interactions into a strong secret that can be used for further communication. It works as follows:

1. For each of one or more servers $B_1, \ldots, B_n$, A runs a *hardening protocol* to obtain a hardened password R_i based on the original weak password P. None of the servers learns either the original password, or the hardened password.

2. Using the set of hardened passwords $R_i, 0 \leq i \leq n$, A derives a strong secret K_i that is used to authenticate with B_i. Additional strong secrets $K_{n+1}, \ldots, K_m$ can be derived in a similar fashion, and used for future communications.

Ford and Kaliski point out that *all* of the servers must collaborate to determine whether A has given the true password, and no strict subset of the servers alone can determine the additional strong secrets $K_{n+1}, \ldots, K_m$. They presented a hardening protocol based on discrete logarithm cryptography, and claim that it can be extended to elliptic curve cryptography [12].

Because of their reliance on multiple servers for basic security, it is clear that the approach of Ford and Kaliski [12], as well as later approaches that are similar in nature [10, 30], fail to meet our first requirement of straightforward backwards compatibility with existing infrastructures.