Analysis of the SSH Key Exchange Protocol

Stephen C. Williams

Dept. Computer Science,
University of Bristol,
Woodland Road,
Bristol, BS8 1UB, United Kingdom,
williams@cs.bris.ac.uk

Abstract. We provide an analysis of the widely deployed SSH protocol's key exchange mechanism. We exploit the design of the SSH key exchange to perform our analysis in a modular manner. First, a *shared secret key* is obtained via a Diffie-Hellman key exchange. Next, a transform is applied to obtain the *application keys* used by later stages of SSH. We define models, following well-established paradigms, that clarify the security provided by each type of key. Previously, there has been no formal analysis of the SSH key exchange protocol. We provide a modular proof of security for the SSH shared secret and application keys. We show that although the shared secret key exchanged by SSH is not indistinguishable, the transformation then applied yields indistinguishable application keys. Our proofs use random oracles to model the hash function used within SSH.

Keywords: SSH; key exchange; security proof

1 Introduction

The SSH protocol is one of the most widely deployed secure network protocols, alongside TLS and IPSec. It was originally designed to replace remote login protocols which sent unprotected information over the network, *e.g.* Telnet. Since then, SSH has become a general purpose tool for securing Internet traffic. Currently, SSH Version 2 is the most recent specification for SSH, defined in a collection of RFCs [22–25], and the version we consider throughout this paper.

The SSH key exchange protocol, specified in [24], is the component of SSH responsible for parties agreeing upon the keys used by the various primitives later in the SSH protocol. The key exchange stage is the first part of the SSH algorithm to run, and occurs in the clear, as no session keys have yet been established to secure communication. Before describing our results we first recall the structure of the SSH key exchange protocol (Figure 1). The protocol proceeds in three stages. Stage 1 is the "Hello" phase, where first an identification string, I, is sent to initiate the protocol, followed by a list of supported algorithms, X. This algorithm list details the supported Diffie–Hellman key groups, signature schemes and certificate formats used by the key exchange, along with the symmetric encryption schemes, MAC and compression functions used once keys are established. Next, Stage 2 runs and the parties agree upon a *shared secret key*, s. This is done via a Diffie–Hellman based key exchange. The server confirms its identity to the client by sending its public key, pk_S, verified by some certificate authority, $\mathsf{sig}_{CA}(pk_S)$, and a signature of the digest, h, of the message transcript and shared secret key. The session identifier sid is set to be the digest, h. Notice as the shared secret key is required when computing the digest, the protocol provides explicit key confirmation to the client. Hence, the protocol is not a vanilla (signed) Diffie–Hellman key exchange. Finally in Stage 3, the shared secret key, session identifier and digest are used to generate six application keys, computed via $k_i = H_i(s||h||\mathsf{sid})$. For SSH, H_i is the SHA-1 hash function, and for each value of i, a different ASCII character is inserted at a well defined point within the plaintext. The application keys are used as an initial IV, encryption key and integrity key, with two sets for each; one for the client to server, and one for the server to client. Analysis on the use, but not establishment, of the application keys has been previously undertaken [4, 20], and is not part of our work.

Note, we assume the existence of some PKI which authenticates public keys during Stage 2 of the key exchange. Thus we do not consider PKI attacks.

	Client		Server
1. Hello Phase		$\xrightarrow{I_C}$	
		$\xleftarrow{I_S}$	
	$r_C \overset{\$}{\leftarrow} \{0,1\}^{128}$		
	$X_C = (r_C\|\|\text{algs}_C)$	$\xrightarrow{X_C}$	
			$r_S \overset{\$}{\leftarrow} \{0,1\}^{128}$
		$\xleftarrow{X_S}$	$X_S = (r_s\|\|\text{algs}_S)$
2. Shared Secret Exchange	$a \overset{\$}{\leftarrow} \{2,\dots,q-1\}$		
	$A \leftarrow g^a$		
		$\xrightarrow{A}$	
			$b \overset{\$}{\leftarrow} \{1,\dots,q-1\}$
			$B \leftarrow g^b$
			$s \leftarrow A^b$
			$h \leftarrow H_0(I_C\|\|I_S\|\|X_C\|\|X_S\|\|$ $pk_S\|\|A\|\|B\|\|s)$
			$\sigma \leftarrow \mathsf{sig}_{sk_S}(h)$
			$D = (pk_S, \mathsf{sig}_{CA}(pk_S), B, \sigma)$
		$\xleftarrow{D}$	
	$s \leftarrow B^a$		
	$h \leftarrow H_0(I_C\|\|I_S\|\|X_C\|\|X_S\|\|$ $pk_S\|\|A\|\|B\|\|s)$		
	If $\mathsf{ver}_{pk_S}(h,\sigma) = \mathsf{false}$ then abort		
	Set $\mathsf{sid} = h$		Set $\mathsf{sid} = h$
3. Application Key Generation	For $i = \{1,\dots,6\}$		For $i = \{1,\dots,6\}$
	$k_i \leftarrow H_i(s\|\|h\|\|sid)$		$k_i \leftarrow H_i(s\|\|h\|\|sid)$

Fig. 1. The SSH key exchange protocol. The connection string of the client (resp. server) is given by I_C (resp. I_S). The algorithms supported the client (resp. server) are denoted by algs_C (resp. algs_S). Hash functions are denoted H_j for $j = \{0,\dots,6\}$, and modelled as random oracles.

The model we use to analyse the SSH key exchange closely follows that of Morrissey, Smart and Warinschi (MSW) [18], and we use a similar methodology. We analyse the protocol in a modular manner, by first giving a model for the security of the shared secret key. The shared secret key is clearly not indistinguishable due to the signature sent during Stage 2. Therefore, we only require security of the shared secret in the one-way sense, *i.e.* the adversary has to recover the entire key. This is based on a weakened version of the Blake-Wilson, Johnson and Menezes (BJM) model [8] and is used by MSW including an adaptation to reflect the situation where only the server possesses a certified public key. Thus, the server is authenticated to the client; however the converse is not true or required. Furthermore, SSH bases partnering around session identifiers established during a run of the protocol. Therefore, unlike the work of MSW, we do not use notions of matching conversations [6] and instead base our notions of partnering around the session identifiers, using techniques of Bellare, Pointcheval and Rogaway (BPR) [5]. The combination of these various approaches allows us to give a model which more closely mimics the real world than any previous model.

The key exchange of SSH encodes its information (excluding the identification strings, I_C and I_S) using the *Binary Packet Protocol (BPP)*, defined in [24]. A BPP packet is a concatenation of packet length, padding length, the payload, random padding and a MAC. In Figure 1 we detail only information contained within the payload field. During the key exchange, the MAC is always omitted, and all data is sent unencrypted. Once keys are established, the packet (excluding the MAC) are encrypted. If one were to define partnering directly around matching conversations, then a trivial attack exists. An adversary is clearly able to alter the random padding; the padding is never checked at the key exchange stage, and such alterations would go unnoticed by the parties. However, the padding is never used by the protocol, so this form of tampering does not affect security of the keys. Again, by using session identifiers to define partnering, this trivial attack is irrelevant.

The model we give for application key security closely follows a combination of the BJM and MSW models. We now require key indistinguishability as opposed to just one-way key security. Here, the adversary must decide whether it is given the real exchanged session key, or a random session key. Again, we use the previously described adaptations to the BJM model, whereby partnering is based on the adaptations of BPR, and we are only concerned with an authenticated server as introduced by MSW.

Our results reduce the security of the shared secret key to that of the decisional Diffie–Hellman (DDH) problem. This is a novel proof technique to many given in the literature, who tend to reduce to a gap-DH assumption, whereby the adversary is given access to a DDH oracle [17, 18]. Such gap-DH style proofs are unable to cope with the shared secret key being part of the signed digest sent from the server to the client. The modular nature of our proof shows that any protocol providing one-way key security, along with appropriate server authentication, may replace the shared secret key exchange stage of the SSH protocol. The transform given to generate the application keys will then yield indistinguishable keys.

Our proofs use random oracles to model the hash function used as part of the SSH key exchange protocol. As such, the standard caveats of using the random oracle model apply.

1.1 Related Work

Surprisingly, there has been no formal security analysis of the key exchange stage of the SSH protocol. Instead, research has focused on the later stages of the protocol where keys are already established. We call this later stage of SSH the "application layer". The first formal security analysis of the application layer of SSH was undertaken by Bellare, Kohno and Namprempre (BKN) [4]. Their work focused on the Binary Packet Protocol (BPP) of SSH and assumed keys had already been securely established. The authors particularly focused on details such as the mode of operation and use of initialization vectors. Several possible weaknesses in the BPP were found, including one first reported by Dai [14]. The proposed fixes to the BPP were shown to be provably secure.

However, Albrecht *et al.* exposed various plaintext-recovering attacks against SSH [2]. These attacks were possible because the original security analysis of BKN neglected to consider the underlying data structure used by the SSH BPP in their security proofs. Later, Paterson and Watson provided a formal security treatment of the application layer of SSH [20], focusing specifically on improving the model of BKN to avoid the pitfalls exposed by Albrecht *et al.* Although the key exchange of SSH does use the SSH BPP to format messages sent over the network, the key exchange is undertaken in the clear, so all elements of the SSH packet (length fields, payload, padding) are known to an adversary. This makes the plaintext-recovery attacks of Albrecht *et al.* not applicable in our setting.

Interestingly, Diffie *et al.* provided several different attacks against various Diffie–Hellman based key exchange schemes [15]. The shared secret key of SSH is computed via Diffie–Hellman key exchange. However, it does not succumb the attacks of [15]. This is largely due to the server signing the (digest of the) message transcript *and* the computed Diffie–Hellman key. Including the key also provides the client with explicit key confirmation.

The seminal work of Bellare and Rogaway (BR) gave the first model for key exchange [6, 7]. This work was in part driven by results such as those of Diffie *et al.* [15]. Following this initial work, there have been a number of other models proposed for key exchange in various applications and environments [1, 3, 5, 8–12, 17–19, 21]. These models largely fall within two groups: Those based upon simulation [3, 12, 21], and those based upon games [5, 8, 9, 11, 17, 18]. Typically the security goal is that of key indistinguishability, where an adversary must determine if it is given the real exchanged session key, or a random key. Our model falls into the category of game based models, which, it has been argued, have certain drawbacks [11], but also several benefits over simulation based techniques. An analysis of SSH in a simulation based setting where one achieves composition [12] may prove beneficial. However, one must take care on the use of UC session identifiers which must be unique and predetermined [11, 13]. The SSH protocol actually defines these session identifiers during the course of the protocol, so whether the UC framework could be adapted to reflect this is unknown.

We use the techniques of various other game based models in our work. In [5] Bellare *et al.* introduce a notion of partnering based upon *session identifiers.* A session identifier is computed according to the protocol algorithm being run. If two sessions share the same session identifier, they are considered to be partners. An alternative to this would be to re-use the original definitions of [6], where the notion of matching conversations is formally defined. However, the SSH protocol explicitly defines a session identifier as part of its execution, so we take this more natural approach for our models. By slightly tweaking the results presented here it is simple to show for SSH, if two sessions have agreed upon the same session identifier then a matching conversation has occurred. This assumes one ignores the previously discussed issues, and *only* considers the payload field of the SSH BPP.

The execution and security models presented here are close to those defined by Morrissey, Smart and Warinschi (MSW) [18] for TLS/SSL. They model the setting where only one party involved in the execution (namely the server) has a certified public key. The MSW model uses matching conversations, so we adapt their techniques to use session identifiers. The execution model of MSW, and indeed our work, is strongly related to work by Blake-Wilson, Johnson and Menezes (BJM) and similar models [8, 9, 17]. The BJM model of [8] extended the original work of Bellare and Rogaway [6] to the public key setting for authenticated key agreement (AK) and authenticated key agreement with key confirmation (AKC). In [17], Kudla and Paterson give a modular proof technique to show the security of a key agreement protocol relative to some gap assumption. Moreover, they propose how one can transform a one-way security definition into an indistinguishability definition via a generic transform [16, 17]. This technique is adapted by MSW [18] and we adopt a similar method to these.

A model closely related to those discussed is that of Canetti and Krawczyk [11]. This model contains an additional corruption capability, whereby the adversary may receive the entire internal session state of a session. In particular, the adversary may be given the ephemeral secrets used in sessions. This query is the only essential difference to the work of Bellare and Rogaway as shown by Choo *et al.* [13]. The shared secret key of SSH can be shown to be insecure when given an adversary with such powerful corruption capabilities. It may be possible to show security of the shared secret key by assuming secure erasures as done for similar protocols [11, 12].

2 Basic Security Model

The security models used in this paper are based on the earlier work of Bellare *et al.* [3, 6, 7], refined by BJM [8] and adapted by BPR [5] and MSW [18].

We now give a general description of the common elements of the models required. Later we specialise this general model for the different tasks required of the various keys used in the SSH protocol.

Registered and unregistered users. We model the setting where there are two kinds of users. Registered users (with identities in some set $\mathcal{U}$) and unregistered users (with identities in some set $\mathcal{U}'$). Each user in $U \in \mathcal{U}$ has associated to it some public key pk_U and the corresponding secret key sk_U. The set $\mathcal{U}$ is used to model the set of servers of the SSH protocol, and the set $\mathcal{U}'$ of clients which do not have long term asymmetric keys.

Execution Model. We consider only two party interactive protocols, that is programs in which an initiator and responder communicate over some insecure communication channel. Each party runs some program, where messages are received from the communication channel, a response is computed which is output onto the communication channel. We call one execution of the program of the initiator (resp. responder) the initiator session (resp. responder session). Each party may engage in multiple, concurrent, initiator and responder sessions.

As standard we assume the adversary is an arbitrary probabilistic polynomial time algorithm with complete control of the communication channel. That is, the adversary intercepts all messages and can respond with any message of its choice. We model this by providing the adversary with access to oracles corresponding to some (initiator or responder) sessions of the protocol.

Oracle state. Each oracle maintains internal state consisting of the variables of the protocol, along with additional data required to model various security requirements required. We generally ignore the details of the exact local variables required and overlook the exact details of execution. For all our models we require the following variables within the oracle state:

- $\tau_{\mathcal{O}} \in \{\perp\} \cup \{0,1\}^*$ – The transcript of all messages sent and received by the oracle $\mathcal{O}$. Additional (non-secret) information related to the execution may also be added to the transcript.
- $role_{\mathcal{O}} \in \{\perp, initiator, responder\}$ – Stores whether this session is an initiator session, or responder session.
- $\mathsf{pid}_{\mathcal{O}} \in \{\perp\} \cup \mathcal{U} \cup \mathcal{U}'$ – The identity of the intended partner session.
- $\mathsf{sid}_{\mathcal{O}} \in \{\perp\} \cup \{0,1\}^*$ – The session identifier is used to define the "partner" session.
- $s_{\mathcal{O}} \in \mathcal{S}_{\text{ss}} \cup \{\perp\}$ – The shared secret key derived by the protocol. For SSH the set $\mathcal{S}_{\text{ss}}$ is a group [24] and we require that $|\mathcal{S}_{\text{ss}}| \geq 2^{\eta}$. Whenever $\gamma_{\mathcal{O}} \neq \mathsf{accepted}$ we assume that $s_{\mathcal{O}} = \perp$.
- $\gamma_{\mathcal{O}} \in \{\mathsf{running}, \mathsf{accepted}, \mathsf{rejected}\}$ – Details whether the execution for oracle $\mathcal{O}$ has finished successfully or not. If $\gamma_{\mathcal{O}} = \mathsf{running}$ then we assume execution has not yet finished, otherwise if $\gamma_{\mathcal{O}} = \mathsf{accepted}$ then execution is assumed to have finished correctly or $\gamma_{\mathcal{O}} = \mathsf{rejected}$ if execution finishes unsuccessfully. We assume if $\gamma_{\mathcal{O}} = \mathsf{accepted}$ then $\mathsf{sid}_{\mathcal{O}} \neq \perp$.
- $\delta_{\mathcal{O}} \in \{\mathsf{uncorrupted}, \mathsf{corrupted}\}$ – This flag shows if this session has been corrupted by the adversary.
- $\omega_{\mathcal{O}} \in \{\mathsf{unrevealed}, \mathsf{revealed}\}$ – Denote whether the shared secret key has been revealed by the adversary.

When execution of a game begins, an initialisation phase is run. This generates all long term keys for identities in $\mathcal{U}$, and the above variables are initialised so that $\tau_{\mathcal{O}}$, $role_{\mathcal{O}}$, $\mathsf{pid}_{\mathcal{O}}$, $\mathsf{sid}_{\mathcal{O}}$ and $s_{\mathcal{O}}$ are set to be undefined, $\gamma_{\mathcal{O}} \leftarrow \mathsf{running}$, $\delta_{\mathcal{O}} \leftarrow \mathsf{uncorrupted}$ and $\omega_{\mathcal{O}} \leftarrow \mathsf{unrevealed}$.

Queries. Following the initialisation of the game, the adversary takes control of the execution and begins issuing queries defined as follows:

- $\mathsf{NewSession}(U, role)$ – This establishes a new session of the protocol for the user $U \in \mathcal{U} \cup \mathcal{U}'$ where $role \in \{initiator, responder\}$. We write Π_U^i for the i-th session of user U.
- $\mathsf{Send}(\mathcal{O}, \mathbf{msg})$ – Simulate the message $\mathbf{msg}$ being sent to an oracle $\mathcal{O}$. This query returns a response computed according to the session maintained by oracle $\mathcal{O}$.
- $\mathsf{Corrupt}(U)$ – This corrupts an identity where $U \in \mathcal{U}$. The long term secret sk_U is returned and the following actions are taken. For all sessions $\mathcal{O} = \Pi_U^i$ with $i \in \mathbb{Z}$, where $\gamma_{\mathcal{O}} = \mathsf{running}$, the value of $\delta_{\mathcal{O}}$ is set to $\mathsf{corrupted}$. No further action between these oracles and the adversary may take place. Furthermore the adversary may not make any $\mathsf{NewSession}(U, role)$ queries. We say that the party U has been corrupted.
- $\mathsf{Reveal}(\mathcal{O})$ – If $\gamma_{\mathcal{O}} = \mathsf{accepted}$, the adversary receives the shared secret $s_{\mathcal{O}}$ of the oracle $\mathcal{O}$. The value $\omega_{\mathcal{O}}$ is set to $\mathsf{revealed}$. We say that the session has been revealed and the adversary is unable to make further queries to this session. If $\gamma_{\mathcal{O}} \neq \mathsf{accepted}$ then this query does nothing.
- $\mathsf{Check}(\mathcal{O}, s)$ – Allows the adversary to check whether it knows the shared secret held by the oracle $\mathcal{O}$. For $\mathcal{O} = \Pi_U^i$ with $i \in \mathbb{Z}$ and $s \in \mathcal{S}_{\text{ss}}$, if $\gamma_{\mathcal{O}} = \mathsf{accepted}$ and $s = s_{\mathcal{O}}$ then true is returned. Otherwise false is returned to the adversary.

At the end of a session's execution the role, partner ID, and oracle state (excluding the shared secret key) are added to the transcript. The execution of the game halts whenever the adversary decides to terminate execution.

3 Shared Secret Exchange

In this section we specialise the basic security model to analyse the security of the shared secret key of the SSH protocol. The exchange of this shared secret key is shown in Step 2 of Figure 1. The shared secret key is used to generate the application keys.

We require only weak security for the shared secret key. Specifically, we require that an adversary be unable to fully recover the shared secret key for a pair of "honest" sessions. The adversary is able to adaptively corrupt parties, reveal the shared secret key of sessions and may check to see if some string s is equal to the shared secret key exchanged for that session.

Only a certain type of oracle may be considered as a valid target for the adversary, to avoid trivial breaks of the scheme. Such oracles are called "fresh", and these are uncorrupted, unrevealed oracles which have successfully completed their execution whose partners have not been corrupted or revealed.

Definition 1. *Fresh Shared Secret Key Oracle A shared secret key oracle $\mathcal{O}$ is said to be fresh is all the following conditions hold:*

(1) $\delta_{\mathcal{O}} = \mathsf{uncorrupted}$, (2) $\omega_{\mathcal{O}} = \mathsf{unrevealed}$, (3) $\gamma_{\mathcal{O}} = \mathsf{accepted}$, (4) $\exists V \in \mathcal{U}$ such that V is uncorrupted and $\mathsf{pid}_{\mathcal{O}} = V$, and (5) no revealed oracle $\mathcal{O}^* = \Pi_V^i$ has session identifier $\mathsf{sid}_{\mathcal{O}^*} = \mathsf{sid}_{\mathcal{O}}$.

Recall, for SSH only the server is authenticated to the client. Hence, in condition (4) notice that the definition specifies an oracle is only considered to be fresh if its partner session is a registered user (*i.e.* a server). Therefore, we do not examine the security of any session where an adversary has caused two unregistered users to share a key. We also note it is perfectly possible under this definition for two servers to engage in communication, as may occur in reality.

Security game for shared secret keys. We define the security of shared secret key protocol Π via the game $\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta)$ between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ as follows:

1. The challenger, $\mathcal{C}$, generates the long term public/private keys for each user $U \in \mathcal{U}$ by running the appropriate key generation algorithm on the security parameter η. The public keys are returned to $\mathcal{A}$.
2. The adversary, $\mathcal{A}$, is allowed to make as many NewSession, Send, Corrupt, Reveal and Check queries as it wishes.
3. At some point adversary $\mathcal{A}$ outputs a pair $(\mathcal{O}^*, s^*)$, where $\mathcal{O}^*$ is some shared secret oracle and $s^* \in \mathcal{S}_{\mathrm{ss}}$. Adversary $\mathcal{A}$ now terminates.

We say that the adversary wins if its output $(\mathcal{O}^*, s^*)$ is such that $\mathcal{O}^*$ is fresh and $s_{\mathcal{O}} = s^*$. In this case the output of $\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta)$ is set to 1. Otherwise the output of the game is set to 0. We give the advantage of $\mathcal{A}$ in winning the $\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}$ game by

$$\mathbf{Adv}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = \Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 1\right].$$

The probability is taken over all random coins used in the execution.

The following definition describes the situation when some party U (a client or server) has engaged in a session with some party V (a server only), which has successfully finished execution, but there is no session of V which is partnered with U. This is used to ensure a server is authenticated to the client.

Definition 2. *Let* $\mathsf{NoPartnering}_{\Pi,\mathcal{A}}(\eta)$ *be the event that at some point during the execution of* $\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta)$ *for two uncorrupted parties* $U \in \mathcal{U} \cup \mathcal{U}'$ *and* $V \in \mathcal{U}$ *there exists an oracle* $\mathcal{O} = \Pi^i_U$ *for* $i \in \mathbb{Z}$ *with* $\mathsf{pid}_{\mathcal{O}} = V$, $\gamma_{\mathcal{O}} = \mathsf{accepted}$, *and yet no oracle* $\mathcal{O}^* = \Pi^j_V$ *for* $j \in \mathbb{Z}$ *has session identifier* $\mathsf{sid}_{\mathcal{O}^*} = \mathsf{sid}_{\mathcal{O}}$ *and shared secret key* $s_{\mathcal{O}^*} = s_{\mathcal{O}}$.

The following definition tells us that a protocol is a secure shared secret key protocol. That is the shared secret key is secret in the one-way sense, and no honest party will incorrectly accept a key.

Definition 3. *Shared Secret Key Security A shared secret key protocol is secure if it satisfies all the following:*

- **Correctness:** *At the end of execution of a benign adversary, which correctly relays messages, any two oracles which share the same session identifier hold the same shared secret key, which is distributed uniformly at random over the key space* $\mathcal{S}_{\text{ss}}$.
- **Key secrecy:** *A shared secret key protocol* Π *satisfies the* OW-SS *key secrecy if for any p.p.t adversary* $\mathcal{A}$, *its advantage* $\mathbf{Adv}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta)$ *is a negligible function in the security parameter.*
- **No Partnering:** *For any p.p.t adversary* $\mathcal{A}$, *the probability the* $\mathsf{NoPartnering}_{\Pi,\mathcal{A}}(\eta)$ *event occurs is a negligible function in the security parameter.*

The security requirements given here are weaker than the more standard requirements of key exchange protocols [5, 6, 8], despite the adversary having similar powers to existing models. Since a signature is sent from the server to the client, as the final step of the shared secret key exchange, key indistinguishability of the shared secret key is not achieved.

Note, the conditions on correctness and no partnering on the shared secret key together imply implicit key confirmation. Furthermore, the definitions given provide security against man-in-the-middle attacks and unknown-key-share attacks.

The model here does not consider the secure establishment of a shared secret key between two unregistered parties. Morrissey *et al.* [18] give an adaptation to the standard definition of matching conversations [6], so only one party in the execution, *i.e.* the server, is required to hold a certified key. We use a similar technique to define the NoPartnering event. However, this is modified to reflect that we base partnering upon session identifiers, not matching conversations. Although the SSH protocol itself allows two unregistered parties to perform a key exchange, such a session would require the user to “trust” the long term key sent by the party acting as the server in such an exchange. Otherwise an adversary would be able to trivially impersonate this user. As such, we do not consider this situation.

We now discuss the security of the shared secret key exchange protocol used in SSH. Let FGps be a family of prime order groups and $\mathbf{Sig} = (\mathsf{gen}, \mathsf{sig}, \mathsf{ver})$ be a public key signature scheme. For $(\mathbb{G}, q, g) \leftarrow \mathbb{G}(\eta)$, where η is the security parameter, the shared secret key exchange of SSH is a Diffie–Hellman key g^{xy} for $x, y \in \mathbb{Z}_q^\times$ randomly chosen by the two participants. The set of shared secret keys is set to $\mathbb{G}$, and we write $\mathsf{SS}(\mathbf{Sig}, \mathsf{FGps})$ for the resulting protocol (Steps 1 and 2 of Figure 1).

It has previously been shown that if parties sign only their sent values, *i.e.* g^x or g^y, then this does not meet the requirements of an authenticated key agreement protocol [15]. However, the SSH protocol demands the server signs a digest of the entire message transcript *and* the shared secret g^{xy}. This authenticates the server to the client and provides the client with explicit key confirmation.

Theorem 1. *Let* FGps *be a family of groups of prime order for which the decisional Diffie–Hellman assumption holds, and let* $\mathbf{Sig}$ *be a secure digital signature scheme. Then* $\Pi = \mathsf{SS}(\mathbf{Sig}, \mathsf{FGps})$ *is a secure shared secret key protocol.*

Proof. That the protocol is correct in the presence of a benign adversary is clear. We prove that for any adversary $\mathcal{A}$ against Π, there exists an algorithm $\mathcal{B}$ for the decisional Diffie–Hellman problem and an adversary $\mathcal{C}$ against $\mathbf{Sig}$ such that

$$\mathbf{Adv}^{\mathsf{OW\text{-}SS}}_{\mathsf{SS}(\mathbf{Sig},\mathsf{FGps}),\mathcal{A}}(\eta) < \mathbf{Adv}^{\mathsf{DDH}}_{\mathsf{FGps},\mathcal{B}}(\eta) + n_P \cdot \mathbf{Adv}^{\mathsf{SEF\text{-}CMA}}_{\mathbf{Sig},\mathcal{C}}(\eta)$$

and

$$\Pr\left[\mathsf{NoPartnering}_{\mathsf{SS}(\mathbf{Sig},\mathsf{FGps}),\mathcal{A}}(\eta)\right] \leq n_P \cdot \mathbf{Adv}^{\mathsf{SEF\text{-}CMA}}_{\mathbf{Sig},\mathcal{C}}(\eta) + \epsilon(\eta),$$

where n_P denotes a bound on the number of participants in the set $\mathcal{U}$ and $\epsilon(\eta)$ is a negligible function in the security parameter.

Let $\mathcal{A}$ be an adversary against the $\mathsf{OW\text{-}SS}$ security of the shared secret key exchange protocol Π of the SSH protocol. We define E to be the event that at the end of execution of $\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta)$ the oracle $\mathcal{O}^* = \Pi^{i^*}_{U^*}$ that $\mathcal{A}$ outputs has on its transcript an incoming message $(pk_{\mathsf{pid}_{\mathcal{O}^*}}, \mathsf{sig}_{CA}(pk_{\mathsf{pid}_{\mathcal{O}^*}}), g^b, \mathsf{sig}(h))$ that was not output by any other honest oracle in the game. Recall $h = H_0(I_{U^*}||I_{\mathsf{pid}_{\mathcal{O}^*}}||X_{U^*}||X_{\mathsf{pid}_{\mathcal{O}^*}}||pk_{\mathsf{pid}_{\mathcal{O}^*}}||g^a|| g^b||g^{ab})$. Since $h = \mathsf{sid}_{\mathcal{O}^*}$, the event E occurs if and only if the event $\mathsf{NoPartnering}$ occurs.

We have that

$$\begin{aligned}\Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 1\right] &= \Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 1 \cap E\right] \\ &\quad + \Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 1 \cap \neg E\right] \\ &= \Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 1 | E\right] \cdot \Pr\left[E\right] \\ &\quad + \Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 1 | \neg E\right] \cdot \Pr\left[\neg E\right] \\ &< \Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 1 | E\right] \\ &\quad + \Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 1 | \neg E\right]\end{aligned}$$

We now construct two algorithms, $\mathcal{B}$ against the DDH problem in the family FGps and $\mathcal{C}$ against the $\mathsf{SEF\text{-}CMA}$ of the underlying signature scheme $\mathbf{Sig}$, according to whether or not the event E occurs or not.

First assume that the event E does not occur. Then we construct the algorithm $\mathcal{B}$ against the DDH problem in FGps as follows. Algorithm $\mathcal{B}$ is given as input the security parameter η and an instance of the decisional Diffie–Hellman problem $(\mathbb{G}, q, g, g^a, g^b, g^c)$ in the group $(\mathbb{G}, q, g) \leftarrow \mathbb{G}(\eta)$. Now $\mathcal{B}$ acts as the challenger to $\mathcal{A}$ in an $\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta)$ game. To do this $\mathcal{B}$ generates n_P identities $\mathcal{U}$ and n'_P identities $\mathcal{U}'$. Now $\mathcal{B}$ runs the key generation algorithm of the public key signature scheme **Sig** with security parameter η to obtain public/private key pairs of all elements in $\mathcal{U}$. Next algorithm $\mathcal{B}$ calls $\mathcal{A}$ using this data.

Algorithm $\mathcal{A}$ starts to make NewSession, Send, Check, Corrupt and Reveal queries which $\mathcal{B}$ answers in the following way:

– If $\mathcal{A}$ makes a $\mathsf{Corrupt}(U)$ query, $\mathcal{B}$ returns sk_U and $\mathcal{A}$ is no longer able to make any queries to oracles belonging to U.
– When a $\mathsf{Send}(\mathcal{O}, \mathbf{msg})$ query is made to an oracle $\mathcal{O} = \Pi^i_U$, if $\mathcal{O}$ is in the "Hello" phase, then $\mathcal{B}$ responds as in the actual protocol. Otherwise:
  - Algorithm $\mathcal{B}$ generates a random value $r_\mathcal{O} \in \{1, \dots, q-1\}$.
  - If U is an initiator and the last message sent by $\mathcal{O}$ was a X_U message in the "Hello" phase then $\mathcal{B}$ sets $x = (g^a)^{r_\mathcal{O}}$ and returns x to $\mathcal{A}$.
  - Else if U is an initiator, for each oracle belonging to $\mathsf{pid}_\mathcal{O}$, $\mathcal{B}$ checks to find $\mathcal{O}^*$ such that $\mathcal{O}^*$ output the message **msg**. If no such oracle exists, then set $\gamma_\mathcal{O} = \mathsf{rejected}$, since signature verification will fail (otherwise the event E has occurred) and return $\perp$. Otherwise retrieve the generated value $r_{\mathcal{O}^*}$ for oracle $\mathcal{O}^*$ and set $s = (g^c)^{r_\mathcal{O} r_{\mathcal{O}^*}}$. Verify that the signature received as part of **msg** is valid, and act appropriately (either accepting or rejecting s). Return $\perp$ to $\mathcal{A}$.
  - Otherwise U is a responder, so $\mathcal{B}$ sets $B = (g^b)^{r_\mathcal{O}}$. For each oracle belonging to $\mathsf{pid}_\mathcal{O}$, $\mathcal{B}$ checks to find $\mathcal{O}^*$ such that $\mathcal{O}^*$ output the message **msg**, now $\mathcal{B}$ can obtain the value $r_{\mathcal{O}^*}$ used to generate **msg**. Next $\mathcal{B}$ computes $s_\mathcal{O} = (g^c)^{r_\mathcal{O} r_{\mathcal{O}^*}}$, If no such $\mathcal{O}^*$ was found then set $s_\mathcal{O} = \mathbf{msg}^{r_\mathcal{O}}$. Compute $h = H_0(I_{\mathsf{pid}_\mathcal{O}} || I_U || X_{\mathsf{pid}_\mathcal{O}} || X_U ||\ pk_U || \mathbf{msg} || B || s_\mathcal{O})$, where $I_{\mathsf{pid}_\mathcal{O}}$ all other values are obtained from $\tau_\mathcal{O}$, and reply with $(pk_U, \mathsf{sig}_{CA}(pk_U), B, \mathsf{sig}_{sk_U}(h))$. Note that if no $\mathcal{O}^*$ was found and this response is delivered to any oracle, verification of the signature will fail (otherwise the event E has occurred).
– If a $\mathsf{Check}(\mathcal{O}, s)$ query is made then $\mathcal{B}$ checks if $\gamma_\mathcal{O} = \mathsf{accepted}$ and $s = s_\mathcal{O}$ then true is returned. Otherwise false is sent back to $\mathcal{A}$.
– If a $\mathsf{Reveal}(\mathcal{O})$ query is made and $\gamma_\mathcal{O} = \mathsf{accepted}$ then $\mathcal{B}$ returns $s_\mathcal{O}$ to $\mathcal{A}$ and sets $\omega_\mathcal{O}$ to $\mathsf{revealed}$.

In this way $\mathcal{B}$ is able to answer all the queries of $\mathcal{A}$ and hence simulates the environment of $\mathcal{A}$. Therefore $\mathcal{A}$ will eventually terminate and output a pair $(\mathcal{O}^\dagger, s^\dagger)$ with $\mathcal{O}^\dagger = \Pi^{i^\dagger}_{U^\dagger}$.

Since we have assumed that the event E does not occur, there will be an entry on the transcript $\tau_{\mathcal{O}^\dagger}$ of the form $(pk_{U^*}, \mathsf{sig}_{CA}(pk_{U^*}), (g^b)^{r_{\mathcal{O}^*}}, \mathsf{sig}_{sk_{U^*}}(h))$, with h constructed by some oracle $\mathcal{O}^*$, as previously described. This means the oracle $\mathcal{O}^\dagger$ holds the shared secret $s_{\mathcal{O}^\dagger} = (g^c)^{r_{\mathcal{O}^\dagger} r_{\mathcal{O}^*}}$. If $s^\dagger = s_{\mathcal{O}^\dagger}$ then the output of the game is set to 1, *i.e.* $\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}} = 1$, and algorithm $\mathcal{B}$ will guess that $g^c = g^{ab}$. Otherwise, the output of the game is set to 0 and $\mathcal{B}$ will guess that $g^c \neq g^{ab}$.

The advantage of the DDH problem is defined as

$$\mathbf{Adv}^{\mathsf{DDH}}_{\mathsf{FGps},\mathcal{B}}(\eta) = \left| \Pr\left[\mathsf{Exec}^{\mathsf{DDH},0}_{\mathsf{FGps},\mathcal{B}}(\eta) = 0\right] - \Pr\left[\mathsf{Exec}^{\mathsf{DDH},1}_{\mathsf{FGps},\mathcal{B}}(\eta) = 1\right] \right|,$$

where $\mathsf{Exec}^{\mathsf{DDH},0}_{\mathsf{FGps},\mathcal{B}}$ denotes the game where $g^c = g^{ab}$ and $\mathsf{Exec}^{\mathsf{DDH},1}_{\mathsf{FGps},\mathcal{B}}$ is the game where $g^c \neq g^{ab}$. Since $\mathcal{B}$ guesses $g^c = g^{ab}$ when $\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}} = 1$, this gives

$$\Pr\left[\mathsf{Exec}^{\mathsf{DDH},b}_{\mathsf{FGps},\mathcal{B}}(\eta) = b\right] = \max\Big\{0, \Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 0|\neg E\right] - \Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 1|\neg E\right]\Big\},$$

and it is clear that

$$\Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 0|\neg E\right] + \Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 1|\neg E\right] = 1.$$

If we assume that $g^{ab} \neq g^c$ then g^c is a uniformly random element of $\mathcal{S}_{\mathsf{SS}}$. Therefore $s_{\mathcal{O}^\dagger} = (g^c)^{r_{\mathcal{O}^\dagger} r_{\mathcal{O}^*}}$ is a uniformly random element of $\mathcal{S}_{\mathsf{SS}}$. All information seen by the adversary for oracle $\mathcal{O}^\dagger$ (excluding $\mathsf{sig}_{sk_{U^*}}(h)$) is independent to $s_{\mathcal{O}^\dagger}$. Furthermore, h is the output of a random oracle, so we are guaranteed that $\mathsf{sig}_{sk_{U^*}}(h)$ leaks no information about the secret $s_{\mathcal{O}^\dagger}$. Thus, the probability $s^\dagger = s_{\mathcal{O}^\dagger} = (g^c)^{r_{\mathcal{O}^\dagger} r_{\mathcal{O}^*}}$ when $g^c \neq g^{ab}$ is $\frac{1}{|\mathcal{S}_{\mathsf{SS}}|}$, *i.e.*

$$\Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 1|\neg E\right] \leq \epsilon(\eta).$$

Therefore,

$$\begin{aligned}\Pr\left[\mathsf{Exec}^{\mathsf{DDH},1}_{\mathsf{FGps},\mathcal{B}}(\eta) = 1\right] &= \left|\Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 0|\neg E\right] - \Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 1|\neg E\right]\right| \\ &= 1 - \epsilon(\eta).\end{aligned}$$

Since we have

$$\begin{aligned}\mathbf{Adv}^{\mathsf{DDH}}_{\mathsf{FGps},\mathcal{B}}(\eta) &= \left|\Pr\left[\mathsf{Exec}^{\mathsf{DDH},0}_{\mathsf{FGps},\mathcal{B}}(\eta) = 0\right] - \Pr\left[\mathsf{Exec}^{\mathsf{DDH},1}_{\mathsf{FGps},\mathcal{B}}(\eta) = 1\right]\right| \\ &= \left|\Pr\left[\mathsf{Exec}^{\mathsf{DDH},0}_{\mathsf{FGps},\mathcal{B}}(\eta) = 0\right] - 1\right|\end{aligned}$$

and since

$$\Pr\left[\mathsf{Exec}^{\mathsf{DDH},0}_{\mathsf{FGps},\mathcal{B}}(\eta) = 0\right] = \max\Big\{0, \Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 0|\neg E\right] - \Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 1|\neg E\right]\Big\}$$

it follows that when $g^c = g^{ab}$ (*i.e.* the shared secret computed in the $\mathsf{OW\text{-}SS}$ game is the real shared secret)

$$\Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 0|\neg E\right] \geq 1 - \mathbf{Adv}^{\mathsf{DDH}}_{\mathsf{FGps},\mathcal{B}}(\eta) + \epsilon(\eta)$$

and therefore

$$\Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 1|\neg E\right] \leq \mathbf{Adv}^{\mathsf{DDH}}_{\mathsf{FGps},\mathcal{B}}(\eta) + \epsilon(\eta).$$

where $\epsilon(\eta)$ is a negligible function in the security parameter.

We now consider the case when the event E does occur. There are two possibilities. Either there exist oracles $\mathcal{O}^* = \Pi^{i^*}_{U^*}$, with $U^* \in \mathcal{U} \cup \mathcal{U}'$ and $\mathcal{O}' = \Pi^{j'}_{V'}$, with $V' \in \mathcal{U}$, such that $\mathsf{sid}_{\mathcal{O}^*} = \mathsf{sid}_{\mathcal{O}'}$ and $s_{\mathcal{O}^*} \neq s_{\mathcal{O}'}$, or there exists oracle $\mathcal{O}^*$ such that no other oracle has session identifier $\mathsf{sid}_{\mathcal{O}^*}$. We first consider the former. Since $\mathsf{sid}_{\mathcal{O}^*} = \mathsf{sid}_{\mathcal{O}'}$, we have $H_0(I_{U^*}||I_{V'}||X_{U^*}||X_{V'}||pk_{V'}||A||g^b||A^b) = H_0(I_{U^*}||I^\dagger||X_{U^*}||X^\dagger||pk_{V'}||A||g^{c^\dagger}|| A^{c^\dagger})$, for some adversarially chosen $I^\dagger$, $X^\dagger$, $c^\dagger \neq b$. This gives $s_{\mathcal{O}^*} = A^{c^\dagger} \neq A^b = s_{\mathcal{O}'}$. However, since H_0 is a random oracle, it is clear the probability an adversary selects the values $I^\dagger$, $X^\dagger$ and $c^\dagger$ correctly is a negligible function in the security parameter.

We now consider the case where there exists an oracle $\mathcal{O}^* = \Pi^{i^*}_{U^*}$ for $U^* \in \mathcal{U} \cup \mathcal{U}'$, such that no other oracle has session identifier $\mathsf{sid}_{\mathcal{O}^*}$. We construct the algorithm $\mathcal{C}$ against the SEF-CMA of the signature scheme used as follows. Algorithm $\mathcal{C}$ is given the security parameter, η, a public verification key $pk^\dagger$ and the corresponding signature oracle $\mathcal{O}^{sk^\dagger}_{\mathsf{sig}}$.

Algorithm $\mathcal{C}$ acts as the challenger to $\mathcal{A}$ in the $\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}$ game. First $\mathcal{C}$ generates n_P identities $\mathcal{U}$ and n'_P identities $\mathcal{U}'$. Now $\mathcal{C}$ randomly selects an oracle $U^\dagger \in \mathcal{U}$ and sets $pk_{U^\dagger} = pk^\dagger$ and $sk_{U^\dagger} = \perp$. Algorithm $\mathcal{C}$ then runs the key generation algorithm of the public key signature scheme with security parameter η to obtain public/private keys for all identities $U \in \mathcal{U} \setminus \{U^\dagger\}$. Setup of the $\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}$ game is completed by passing the sets of identities and all public keys to adversary $\mathcal{A}$.

Adversary $\mathcal{A}$ now starts to make NewSession, Send, Check, Corrupt and Reveal queries, which $\mathcal{C}$ answers in the following way:

- If a Corrupt query is made and $U \neq U^\dagger$ then $\mathcal{C}$ returns sk_U, and $\mathcal{A}$ can no longer query any oracle belonging to identity U. If $U = U^\dagger$ then $\mathcal{C}$ aborts.
- When a $\mathsf{Send}(\mathcal{O}, \mathbf{msg})$ query is made for oracle $\mathcal{O} = \Pi^i_U$, if $U \neq U^\dagger$ or $\mathcal{O}$ is still in the "Hello" phase then $\mathcal{C}$ responds as in the correct execution of the protocol. Otherwise $\mathcal{C}$ selects $b \xleftarrow{\$} \{1, \ldots, q-1\}$, computes $B \leftarrow g^b$, $A \leftarrow \mathbf{msg}$, $s \leftarrow A^b$ and obtains $h = H_0(I_{\mathsf{pid}_{\mathcal{O}}}||I_{U^\dagger}||X_{\mathsf{pid}_{\mathcal{O}}}||X_{U^\dagger}||pk_{U^\dagger}||A||B||s)$, where $I_{\mathsf{pid}_{\mathcal{O}}}$ and all other values are obtained from the oracle transcript $\tau_{\mathcal{O}}$. Algorithm $\mathcal{C}$ obtains a signature on h using its signature oracle and sends $(pk_{U^\dagger}, \mathsf{sig}_{CA}(pk_{U^\dagger}), B, \mathcal{O}^{sk^\dagger}_{\mathsf{sig}}(h))$ as the response to $\mathcal{A}$.
- If a $\mathsf{Check}(\mathcal{O}, s)$ or $\mathsf{Reveal}(\mathcal{O})$ query is made then $\mathcal{C}$ knows the shared secret of the queried oracle so can answer these queries honestly.

If $\mathcal{C}$ does not abort, then the environment of $\mathcal{A}$ is simulated perfectly and so eventually $\mathcal{A}$ will output a pair $(\Pi^{i^*}_{U^*}, s^*)$.

Let $\mathcal{O}^* = \Pi^{i^*}_{U^*}$, if $\mathsf{pid}_{\mathcal{O}^*} = U^\dagger$ and $\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 1$, then there exists an entry of the form $(pk_{U^\dagger}, \mathsf{sig}_{CA}(pk_{U^\dagger}), B, \mathsf{sig}_{sk_{U^\dagger}}(h))$ on the transcript $\tau_{\mathcal{O}^*}$ of $\mathcal{O}^*$, where the signature $\mathsf{sig}_{sk_{U^\dagger}}(h)$ has correctly verified under $pk_{U^\dagger}$. Since the event E has occurred this entry did not come from $U^\dagger$, so was not a signature output by $\mathcal{O}^{sk^\dagger}_{\mathsf{sig}}$. Therefore $\mathcal{C}$ reconstructs h using the transcript $\tau_{\mathcal{O}^*}$ and outputs the pair $(h, \mathsf{sig}_{sk_{U^\dagger}}(h))$, which is a valid forgery in the SEF-CMA game. As U^* is randomly selected, independently from the choices of $\mathcal{A}$ we obtain

$$\mathbf{Adv}^{\mathsf{SEF\text{-}CMA}}_{\mathbf{Sig},\mathcal{C}}(\eta) \geq \frac{1}{n_p} \cdot \Pr\left[\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{A}}(\eta) = 1 | E\right],$$

and the result now follows.

4 Application Keys

In this section we extend the basic security model to analyse the security of the application keys generated from the shared secret key exchange stage of the SSH protocol.

As discussed in the introduction, we focus on protocols of a specific form: First a shared secret key is agreed by the parties via some shared secret key exchange protocol Π. Next the application keys are generated by some application key generation Σ. In SSH, the application keys are derived by applying some hash function to the shared secret key as discussed in Section 1. We refer to the application key, k, as the concatenation of the application keys k_i generated for SSH (Step 3, Figure 1). We write $\Pi; \Sigma$ for the full protocol where the shared secret key is exchanged using Π and application keys are generated by running Σ on the exchanged shared secret key.

We extend the definitions of Section 2 to model the requirements of the application keys. We assume that application keys belong to some set $\mathcal{S}_{\mathsf{AK}}$, where $|\mathsf{SS}_{\mathsf{AK}}| \geq 2^{\eta}$, where η is the security parameter. Application key oracles, $\mathcal{Q} = \Sigma^i_U$, maintain the variables previously described, namely $\tau_{\mathcal{Q}}$, $role_{\mathcal{Q}}$, $\mathsf{pid}_{\mathcal{Q}}$, $\mathsf{sid}_{\mathcal{Q}}$, $\gamma_{\mathcal{Q}}$, $\delta_{\mathcal{Q}}$ and $\omega_{\mathcal{Q}}$. The shared secret is stored in $s_{\mathcal{Q}}$. Additionally they have a new variable $k_{\mathcal{Q}} \in \mathcal{S}_{\mathsf{AK}}$, for the application key obtained in the session, and we introduce the variable $\psi_{\mathcal{Q}} \in \{\mathsf{uncompromised}, \mathsf{compromised}\}$ and explain below when this is set.

The adversary is granted additional powers to those previously described. The adversary may make the new query $\mathsf{Compromise}(\mathcal{Q})$. This query returns the application key $k_{\mathcal{Q}}$ to the adversary and sets $\psi_{\mathcal{Q}}$ to $\mathsf{compromised}$. Note that since the adversary may still make Reveal queries to obtain the shared secret key, we introduce the $\mathsf{Compromise}$ query under a different name for clarity.

To capture the security requirements of application keys we grant the adversary access to the new query Test. When a $\mathsf{Test}(\mathcal{Q})$ query is made, a bit $b \in \{0, 1\}$ is randomly selected. Next if $b = 0$ the key $k_{\mathcal{Q}}$ is returned to the adversary, otherwise a randomly selected element of $\mathcal{S}_{\mathsf{AK}}$ is returned to the adversary.

An application key oracle is a valid target for the Test query if it is considered "fresh", where the notion of freshness is similar to that of shared secret key exchange. Namely, the oracle is uncorrupted, unrevealed, uncompromised and has accepted some application key, where the partner session has also not been corrupted, revealed or compromised.

Definition 4. *Fresh Application Key Oracle Let $\mathcal{Q}$ be an application key oracle. Oracle $\mathcal{Q}$ is said to be fresh if all the following conditions hold:*

(1) $\delta_{\mathcal{Q}} = \mathsf{uncorrupted}$, (2) $\omega_{\mathcal{Q}} = \mathsf{unrevealed}$, (3) $\psi_{\mathcal{Q}} = \mathsf{uncompromised}$, (4) $\gamma_{\mathcal{Q}} = \mathsf{accepted}$, (5) $\exists V \in \mathcal{U}$ such that V is uncorrupted and $\mathsf{pid}_{\mathcal{Q}} = V$, and (6) no revealed or compromised oracle $\mathcal{Q}^*$ has session identifier $\mathsf{sid}_{\mathcal{Q}^*} = \mathsf{sid}_{\mathcal{Q}}$.

Security game for application key exchange protocols. We define the security of an application key protocol $\Pi; \Sigma$ via the game $\mathsf{Exec}^{\mathsf{IND\text{-}AK}}_{\Pi;\Sigma,\mathcal{A}}(\eta)$ between an adversary $\mathcal{A}$ and the challenger $\mathcal{C}$ as follows:

- Challenger $\mathcal{C}$ generates the long term public/private keys for each user $U \in \mathcal{U}$, and returns the public keys to $\mathcal{A}$.
- Adversary $\mathcal{A}$ is able to make as many $\mathsf{NewSession}$, Send, $\mathsf{Corrupt}$, Reveal, Check and $\mathsf{Compromise}$ queries as it wishes.

– At any point in the game, the adversary $\mathcal{A}$ may make a single $\mathsf{Test}(\mathcal{Q}^*)$ query.
– The adversary outputs a bit b' and terminates.

We say that the adversary $\mathcal{A}$ wins if $\mathcal{Q}^*$ is fresh at the end of the game and $b = b'$, where b is the bit selected when the Test query is made. If this occurs then the output of $\mathsf{Exec}^{\mathsf{IND\text{-}AK}}_{\Pi;\Sigma,\mathcal{A}}$ is set to 1. Otherwise the output is set to 0. We give the advantage of $\mathcal{A}$ in winning the $\mathsf{Exec}^{\mathsf{IND\text{-}AK}}_{\Pi;\Sigma,\mathcal{A}}$ game by

$$\mathbf{Adv}^{\mathsf{IND\text{-}AK}}_{\Pi;\Sigma,\mathcal{A}}(\eta) = \left|\Pr\left[\mathsf{Exec}^{\mathsf{IND\text{-}AK}}_{\Pi;\Sigma,\mathcal{A}}(\eta) = 1\right] - \frac{1}{2}\right|.$$

Using this we now define the security of an application key exchange protocol.

Definition 5. *Application Key Security An application key exchange protocol is secure if it satisfies the following conditions:*

– **Correctness:** *In the presence of a benign adversary which honestly relays messages, any two oracles which share the same session identifier, will, at the end of execution, hold the same application key, which is distributed uniformly at random over the application key space $\mathcal{S}_{\mathsf{AK}}$.*
– **Key secrecy:** *An application key exchange protocol $\Pi;\Sigma$ satisfies $\mathsf{IND\text{-}AK}$ key secrecy if for any p.p.t. adversary $\mathcal{A}$, its advantage $\mathbf{Adv}^{\mathsf{IND\text{-}AK}}_{\Pi;\Sigma,\mathcal{A}}(\eta)$ is a negligible function in the security parameter.*
– **No Partnering:** *The probability of the event $\mathsf{NoPartnering}_{\Pi;\Sigma,\mathcal{A}}(\eta)$ occurring, for any p.p.t. adversary $\mathcal{A}$, is negligible function in the security parameter.*

The model given here provides strong guarantees for the application keys used. The security requirements given provide standard key indistinguishability guarantees.

We now show that the application key exchange protocol obtained from any secure shared secret key exchange protocol and the application key generation protocol of SSH is secure.

For any shared secret key exchange protocol Π, and set of hash functions $H = \{H_1, \ldots H_6\}$ we write $(\Pi; \mathsf{AK}_{\mathsf{SSH}}(H))$ for the application key exchange protocol obtained by extending Π with the application key generation of SSH. The following theorem gives us that, starting with a shared secret key exchange protocol secure in the sense of Definition 3, the above transformation gives a secure application key protocol.

Theorem 2. *Let Π be a secure shared secret key exchange protocol and a set of random oracles be given by $H = \{H_1, \ldots H_6\}$. Then $(\Pi; \mathsf{AK}_{\mathsf{SSH}}(H))$ is a secure application key exchange protocol.*

Proof. It is clear that in the presence of a benign adversary the protocol is correct. Let $\mathcal{A}$ be an $\mathsf{IND\text{-}AK}$ adversary against $(\Pi; \mathsf{AK}_{\mathsf{SSH}}(H))$, where H is the set of random oracles $H = \{H_1, \ldots, H_6\}$.

We prove for any adversary $\mathcal{A}$, there exists an algorithm $\mathcal{B}$ against the $\mathsf{OW\text{-}SS}$ security of the shared secret and an adversary $\mathcal{C}$ against the $\mathsf{NoPartnering}$ event such that

$$\mathbf{Adv}^{\mathsf{IND\text{-}AK}}_{\Pi;\Sigma,\mathcal{A}}(\eta) < \mathbf{Adv}^{\mathsf{OW\text{-}SS}}_{\mathcal{B},\Pi}(\eta) + \Pr\left[\mathsf{NoPartnering}_{\Pi,\mathcal{C}}(\eta)\right].$$

Since the Σ stage of the protocol $\Pi;\Sigma$ does not send any messages, it is clear that if the event $\mathsf{NoPartnering}_{\Pi;\Sigma,\mathcal{A}}(\eta)$ occurs, then for an adversary $\mathcal{C}$ playing the game $\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{C}}(\eta)$ the event $\mathsf{NoPartnering}_{\Pi,\mathcal{C}}(\eta)$ would occur. The construction of $\mathcal{C}$ is in the obvious manner. Thus it follows that

$$\begin{aligned}\Pr\left[\mathsf{NoPartnering}_{\Pi;\Sigma,\mathcal{A}}(\eta)\right] &= \Pr\left[\mathsf{NoPartnering}_{\Pi,\mathcal{C}}(\eta)\right]\\ &\leq \epsilon(\eta),\end{aligned}$$

where $\epsilon(\eta)$ is a negligible function in the security parameter.

It now remains to consider when the event E does not occur. We construct algorithm $\mathcal{B}$ against the $\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{B}}(\eta)$ game, where $\mathcal{B}$ acts as the challenger in an $\mathsf{Exec}^{\mathsf{IND\text{-}AK}}_{\Pi;\Sigma,\mathcal{A}}(\eta)$ game against $\mathcal{A}$. The algorithm $\mathcal{B}$ simulates $H_1,\ldots,H_6$ by maintaining six lists H_1-list, ..., H_6-list of queries and responses to the oracles $H_1,\ldots,H_6$. The input to algorithm $\mathcal{B}$ as part of the $\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{B}}(\eta)$ game is used as the input to adversary $\mathcal{A}$.

Algorithm $\mathcal{B}$ answers $\mathcal{A}$'s Corrupt, Reveal and Check queries by forwarding these to the challenger of $\mathcal{B}$ and passing the response back to $\mathcal{A}$. When $\mathcal{A}$ makes the query $\mathsf{Send}(\mathcal{O},\mathbf{msg})$, algorithm $\mathcal{B}$ passes this to its challenger, and forwards the response to $\mathcal{A}$. Algorithm $\mathcal{B}$ records all such Send queries and their responses, so for each oracle $\mathcal{O}$, algorithm $\mathcal{B}$ has a copy of the transcript $\tau_{\mathcal{O}}$.

If $\mathcal{A}$ makes a $\mathsf{Compromise}(\mathcal{O})$ query, then $\mathcal{B}$ makes a query $\mathsf{Reveal}(\mathcal{O})$ to its challenger and uses the returned shared secret key, s, and the values on $\mathcal{B}$'s copy of the transcript $\tau_{\mathcal{O}}$ to construct $h = H_0(I_C||I_S||X_C||X_S||pk_S||A||B||s)$. Algorithm $\mathcal{B}$ calls its oracles $H_1,\ldots,H_6$ on input $(s||h||h)$ to obtain $k = k_1||\ldots||k_6$. The appropriate values are stored on H-lists, H_1-list, ..., H_6-list, and k is returned to $\mathcal{A}$.

At some point $\mathcal{A}$ will make a Test query. Algorithm $\mathcal{B}$ returns a random key from the key space $\mathcal{S}_{\mathsf{AK}}$ to adversary $\mathcal{A}$.

Eventually $\mathcal{A}$ will terminate and output its guess for the bit b. We find that if $\mathsf{Exec}^{\mathsf{IND\text{-}AK}}_{\Pi;\Sigma,\mathcal{A}}(\eta) = 1$, then since $H_1,\ldots,H_6$ are modelled as random oracles, adversary $\mathcal{A}$ must have queried one (or more) of the oracles $H_1,\ldots,H_6$ with inputs corresponding to the shared secret key of an application key oracle $\mathcal{O}^*$. In addition the oracle $\mathcal{O}^*$ must be fresh in the $\mathsf{Exec}^{\mathsf{OW\text{-}SS}}_{\Pi,\mathcal{B}}(\eta)$ game.

Algorithm $\mathcal{B}$ now scans each of the H-lists and checks whether s^* of the component $(s^*||h^*||h^*)$ from the H-lists corresponds to the shared secret key of oracle $\mathcal{O}^*$. This is done using the Check query. When the correct key is found, algorithm $\mathcal{B}$ outputs $(\mathcal{O}^*, s^*)$ and terminates.

The theorem now follows.

References

1. Abdalla, M., Chevassut, O., Pointcheval, D.: One-time verifier-based encrypted key exchange. In: Public Key Cryptography – PKC 2005. (2005) 47–64 Springer LNCS 3386.
2. Albrecht, M., Paterson, K., Watson, G.: Plaintext recovery attacks against SSH. In: IEEE Symposium on Security and Privacy. (2009) 16–26 IEEE Computer Society.
3. Bellare, M., Canetti, R., Krawczyk, H.: A modular approach to the design and analysis of authentication and key exchange protocols. In: Proceedings of the 13th Annual ACM Symposium on Theory of Computing. (1998) 419–428 ACM.
4. Bellare, M., Kohno, T., Namprempre, C.: Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the encode-then-encrypt-and-MAC paradigm. ACM Transactions on Information and Systems Security **7**(2) (2004) 206–241

5. Bellare, M., Pointcheval, D., Rogaway, P.: Authenticated key exchange secure against dictionary attacks. In: Advances in Cryptology – EUROCRYPT 2000. (2000) 139–155 Springer-Verlag LNCS 1807.
6. Bellare, M., Rogaway, P.: Entity authentication and key distribution. In: Advances in Cryptology – CRYPTO 1993. (1993) 232–249 Springer-Verlag LNCS 773.
7. Bellare, M., Rogaway, P.: Provably secure session key distribution: The three party case. In: 27th Symposium on Theory of Computing – STOC 1995. (1995) 57–66 ACM.
8. Blake-Wilson, S., Johnson, D., Menezes, A.: Key agreement protocols and their security analysis. In: IMA International Conference on Cryptography and Coding. (1997) 30–45 Springer-Verlag.
9. Blake-Wilson, S., Menezes, A.: Entity authentication and authenticated key transport protocols employing asymmetric techniques. In: Security Protocols Workshop 1997. (1998) 137–158 Springer LNCS 1361.
10. Bresson, E., Chevassut, O., Pointcheval, D.: Provably authenticated group Diffie–Hellman key exchange - the dynamic case. In: Advances in Cryptology – ASIACRYPT 2001. (2001) 290–309 Springer Verlag LNCS 2248.
11. Canetti, R., Krawczyk, H.: Analysis of key exchange protocols and their use for building secure channels. In: Advances in Cryptology – EUROCRYPT 2001. (2001) 453–474 Springer-Verlag LNCS 2045.
12. Canetti, R., Krawczyk, H.: Universally composable notions of key exchange and secure channels. In: Advances in Cryptology – EUROCRYPT 2002. (2002) 337–351 Springer-Verlag LNCS 2332.
13. Choo, K.K., Boyd, C., Hitchcock, Y.: Examining indistinguishability-based proof models for key establishment protocols. In: Advances in Cryptology – ASIACRYPT 2005. (2005) 585–604 Springer Verlag LNCS 3788.
14. Dai, W.: An attack against SSH2 protocol (6th Feb 2002) E-mail to the SECSH Working Group available from `ftp://ftp.ietf.org/ietf-mail-archive/secsh/2002-02.mail`.
15. Diffie, W., Oorschot, P.V., Wiener, M.: Authentication and authenticated key exchanges. Designs, Codes and Cryptography **2**(2) (1992) 107–125
16. Kudla, C.: Special signature schemes and key agreement protocols (2006) PhD Thesis, Royal Holloway University of London.
17. Kudla, C., Paterson, K.: Modular security proofs for key agreement protocols. In: Advances in Cryptology - ASIACRYPT 2005. (2005) 549–565 Springer Verlag LNCS 3788.
18. Morrissey, P., Smart, N., Warinschi, B.: The TLS handshake protocol: A modular analysis. Journal of Cryptology **23**(2) (2010) 187–223
19. Paterson, K., Stebila, D.: One-time-password-authenticated key exchange. In: ACISP 2010. (2010) 264–281 Springer Verlag LNCS 6168.
20. Paterson, K., Watson, G.: Plaintext-dependent decryption: A formal security treatment of SSH-CTR. In: Advances in Cryptology – EUROCRYPT 2010. (2010) 345–361 Springer-Verlag LNCS 6110.
21. Shoup, V.: On formal models for secure key exchange (version 4) (Preprint, 1999)
22. Ylonen, T., Lonvick, C.: The secure shell (SSH) protocol architecture (2006) RFC 4251.
23. Ylonen, T., Lonvick, C.: The secure shell (SSH) authentication protocol (2006) RFC 4252.
24. Ylonen, T., Lonvick, C.: The secure shell (SSH) transport layer protocol (2006) RFC 4253.
25. Ylonen, T., Lonvick, C.: The secure shell (SSH) connection protocol (2006) RFC 4254.