Verified Contributive Channel Bindings for Compound Authentication

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Alfredo Pironti
INRIA Paris-Rocquencourt
karthikeyan.bhargavan,antoine.delignat-lavaud,alfredo.pironti@inria.fr

***Abstract*—Compound authentication protocols, such as EAP in IKEv2 or SASL over TLS, bind application-level authentication to a transport-level authenticated channel in order to obtain strong composite authentication under weak trust assumptions. Despite their wide deployment, these protocols remain poorly understood, leading to several credential forwarding man-in-the-middle attacks. We present formal models for several compound authentication protocols, and analyze them against a rich threat model that includes compromised certificates, leaked session keys, and Diffie-Hellman small subgroup confinement. Our analysis uncovers new compound authentication attacks on TLS renegotiation, SSH re-exchange, IKEv2 resumption, and a number of other channel binding proposals. We propose new channel bindings and formally evaluate their effectiveness using the automated symbolic cryptographic protocol verifier, ProVerif. In particular, we present the first formal models that can reconstruct the recently published triple handshake attacks on TLS, and the first automated analysis of its proposed countermeasure.**

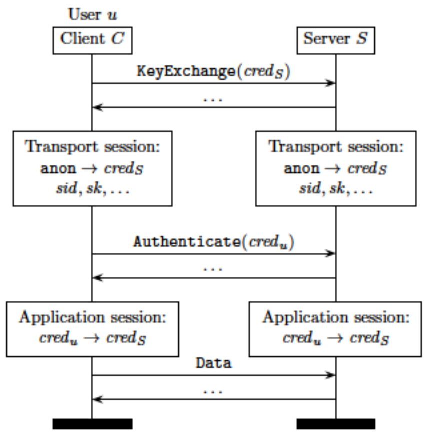

User u
Client C
Server S
KeyExchange($cred_S$)
...
Transport session: anon → $cred_S$ $sid, sk, \ldots$
Transport session: anon → $cred_S$ $sid, sk, \ldots$
Authenticate($cred_u$)
...
Application session: $cred_u \rightarrow cred_S$
Application session: $cred_u \rightarrow cred_S$
Data
...

Fig. 1. A compound authentication protocol combining a server-authentication transport protocol with application-level user authentication.

I. Compound Authentication

Mutual authentication of clients and servers is an important security goal of any distributed system architecture. To this end, popular cryptographic protocols such as Transport Layer Security (TLS), Secure Shell (SSH), and Internet Protocol Security (IPsec) implement well-studied cryptographic constructions called Authenticated Key Exchanges (AKEs) that can establish secure transport channels between clients and servers and at the same time authenticate them to each other.

However, a common deployment scenario for these protocols, as depicted in Figure 1, does not use mutual authentication. Instead the transport-level protocol authenticates only the server and establishes a unilaterally-authenticated secure channel where the client is anonymous. The client (or user) is authenticated by a subsequent application-level authentication protocol that is tunneled within the transport channel. The composition of these two protocols aims to provide *compound authentication*: a guarantee that the same two participants engaged in both protocols, and hence that both agree upon the identities of each other (and other session parameters).

Examples of such compound authentication protocols are widespread, and we list here some that use TLS as the transport-level protocol. TLS servers almost universally use only server authentication, relying on various application-level user authentication protocols within the TLS channel: HTTPS websites use cookies or HTTP authentication, wireless networks use the Extended Authentication Protocol (EAP), mail and chat servers uses the Simple Authentication and Security Layer (SASL), windows servers use the Generic Security Service Application Program Interface (GSSAPI). Even within the TLS protocol, clients and servers can re-authenticate each other via a second key exchange (called a *renegotiation*) tunneled within the first. For example, a server-authenticated TLS key exchange may be followed by a mutually-authenticated renegotiation key exchange that takes the place of the application-level protocol in Figure 1.

Similar layered compound authentication protocols have been built using SSH and IPsec. More generally, compound authentication protocols may compose any sequence of authentication and key (re-)exchange protocols, in each of which one or both participants may be anonymous. In this paper, we mainly consider protocols that use TLS, SSH, and IPsec to create transport channels, followed by some strong authentication protocol based either on public-key cryptography, or on challenge-response password verification.

Man-in-the-middle attacks: Even if two protocols are independently secure, their composition may fail to protect against man-in-the-middle (MitM) attack, such as the one depicted in Figure 2. Suppose a client C sets up a transport channel with a malicious server M and then authenticates the user u at M. Further assume that the credential $cred_u$ that C uses for u (e.g. an X.509 public-key certificate) is also accepted by an honest

NDSS '15, 8-11 February 2015, San Diego, CA, USA
Copyright 2015 Internet Society, ISBN 1-891562-38-X
http://dx.doi.org/10.14722/ndss.2015.23277

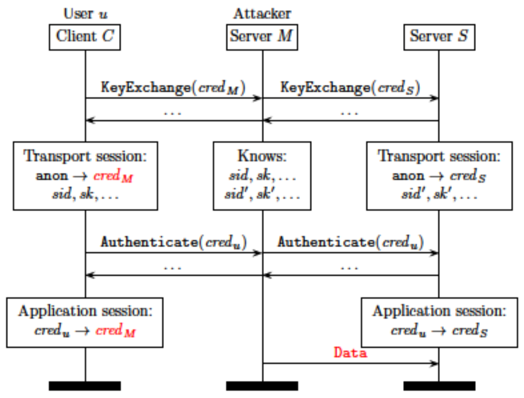

User u
Client C
Attacker
Server M
Server S
KeyExchange($cred_M$)
KeyExchange($cred_S$)
Transport session: anon → $cred_M$ $sid, sk, \ldots$
Knows: $sid, sk, \ldots$ $sid', sk', \ldots$
Transport session: anon → $cred_S$ $sid', sk', \ldots$
Authenticate($cred_u$)
Application session: $cred_u \rightarrow cred_M$
Application session: $cred_u \rightarrow cred_S$
Data

Fig. 2. Man-in-the-Middle (MitM) credential forwarding attack.

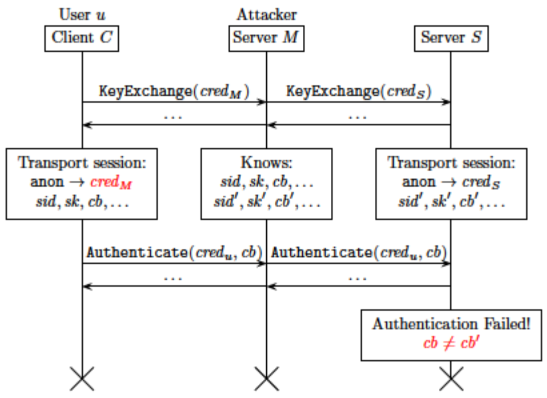

User u
Client C
Attacker
Server M
Server S
KeyExchange($cred_M$)
KeyExchange($cred_S$)
Transport session: anon → $cred_M$ $sid, sk, cb, \ldots$
Knows: $sid, sk, cb, \ldots$ $sid', sk', cb', \ldots$
Transport session: anon → $cred_S$ $sid', sk', cb', \ldots$
Authenticate($cred_u$, cb)
Authentication Failed! $cb \neq cb'$

Fig. 3. Channel binding to prevent MitM attacks.

server S. Then, M can set up a separate transport channel to S and forward all messages of the user-authentication protocol back and forth between C and S. At the end of the protocol, M has managed to authenticate as u on S, even though it does not have access to u's private keys.

This generic credential forwarding MitM attack on layered authentication protocols was first noted by Asokan et. al. [1] in the context of EAP authentication in various wireless network authentication protocols, and the attack motivated their statement of the compound authentication problem [2]. The attack applies to any scenario where the same user credential may be used with two different servers, one of which may be malicious. It also applies when the user authentication protocol may be used both within and outside a transport protocol. Another instance of the attack is the TLS renegotiation vulnerability discovered by Ray and Dispensa [3] and independently by Rex [4]. Other similar MitM attacks on HTTP authentication over TLS are noted in [5], [6].

Channel binding countermeasures: In response to these various attacks, new countermeasures were proposed and implemented in various protocols. The key idea behind these countermeasures is depicted in Figure 3. The user authentication protocol additionally authenticates a *channel binding* value derived from the transport-level session. Since the MitM M is managing two different sessions, one with C and one with S, the two channels should have different channel bindings (cb, cb'). In the user authentication protocol, C binds the user's credential $cred_u$ to its channel cb with M. When S receives the credential, it expects it to be bound to its channel cb' with M, and refuses the credential otherwise. Hence, the channel binding prevents credential forwarding.

Channel-bound compound authentication protocols differ mainly on what the transport-level channel binding value cb should be, and how it should be bound to the application-level user authentication protocol. Tunneled EAP methods use a technique called *cryptographic binding* which effectively uses the outer session key sk as a channel binding and uses a key derived from it to complete the user authentication protocol [7], [8], [9]. Application-level authentication frameworks such as SASL can use any channel binding that satisfies the definition in [10]. Three channel bindings for TLS are defined in [11]. To fix the TLS renegotiation attack, all TLS implementations implement a mandatory protocol extension [12] that binds each key exchange to (a hash of) the transcript of the previous exchange, effectively a channel binding that is similar to the definition of `tls-unique` in [11]. Extended Protection for Authentication on Windows servers binds user authentication to TLS to prevent credential forwarding [13]. Other countermeasures bind the user authentication protocol to the client or server certificates in the underlying TLS session [14], [5], [6].

Channel synchronization attacks: Despite the widespread implementation of channel binding countermeasures in compound authentication protocols, few of these have been formally evaluated. Indeed, even the original MitM attacks were discovered by hand, rather than with the help of formal tools. In the absence of systematic analyses against a variety of threat models, how can we be sure that these countermeasures work?

Of the various countermeasures, TLS renegotiation has received the most formal attention. In particular, a proof of compound authentication for a sequence of TLS-DHE handshakes appears in [15]. However, the recent *triple handshake attacks* [16] showed that this countermeasure fails if TLS session resumption is also enabled. More generally, the triple handshake attacks demonstrated that most commonly-used TLS channel bindings were ineffective; they still allowed an MitM to mount credential forwarding attacks.

The problem is that channel binding countermeasures only work if the attacker cannot *synchronize* the channel bindings on the two channels. In Figure 3, if M can ensure that cb = cb', then the countermeasure no longer works. Specifically, the triple handshake attacks rely on synchronizing TLS channel bindings (such as the renegotiation countermeasure [12]) across two different connections that each use a TLS-RSA or TLS-DHE key exchange followed by session resumption.

In this paper, we show that such channel synchronization attacks apply also to channel bindings proposed for other key exchanges such as the Internet Key Exchange (IKEv2, used in IPsec), Secure Remote Password (SRP), and Elliptic Curve Diffie-Hellman (ECDHE) using Curve25519. In each of these cases, we show that the existing channel bindings provided by these protocols are inadequate for compound authentication. We show a variation of the triple handshake attack on IKEv2, using IKEv2 resumption and IKEv2 re-authentication. We also show a triple exchange vulnerability in SSH key re-exchange where a client and server can be confused about the sequence of exchanges on the connection. All these attacks apply to mainstream implementations of these protocols.

A formal analysis of channel bindings: To systematically evaluate various channel binding proposals and discover new attacks automatically, we model a series of compound authentication protocols in the applied pi calculus [17] and analyze them with the protocol analyzer ProVerif [18].

We formalize the general security goals of compound authentication, propose a powerful threat model, and analyze various protocols built using TLS and SSH. Our formal analysis automatically finds many of the new attacks presented in this paper and also rediscovers older attacks. In particular, our models of TLS resumptions and renegotiation are the first to automatically reconstruct the triple handshake attack and other MitM attacks on TLS-based compound authentication.

We propose a new security requirement for key exchange protocols that enables them to be used for compound authentication. They must provide agreement on a channel binding value that is *contributive*, that is, it cannot be determined solely by one of the two participants. We propose new contributive channel bindings for IKEv2, SSH, and SRP. We analyze our new SSH channel bindings as well as the TLS session hash countermeasure [19] for the triple handshake attacks. We show that within our threat model and under the limitations of our symbolic cryptographic abstractions, these contributive channel bindings prevent channel synchronization attacks.

Outline: Section II presents general notations and formal definitions for the protocol model used in the paper as well as detailed examples of several compound authentication protocols. Section III presents old and new channel synchronization attacks on some compound authentication protocols. Section IV proposes new contributive channel bindings to prevent these attacks. Section V describes our ProVerif models that encode the formal definitions of Section II; it then shows how we can discover some of the attacks of Section III and analyze the countermeasures of Section IV. Section VI briefly discusses related work. Section VII concludes.

II. Formal Protocol Model

We consider a family of *two-party authentication protocols*. Each protocol session is executed by a pair of *principals* over an untrusted network. Each principal (written p, a, b) has access to a set of public *credentials* (written $c_1, c_2, \ldots$), and each credential has an associated secret (written $s_1, s_2, \ldots$) that may be used to create a *proof of possession* for the credential. Credentials and their secrets may be shared by two or more principals. A credential may be *compromised*, in which case its secret is revealed to the adversary.

The adversary is treated as a distinguished principal with access to a set of compromised credentials. At run-time, the adversary may trigger any number of instances of each authentication protocol. Each instance has a protocol role: it is either a *initiator* or a *responder* and this role is played by a principal. By the end of the protocol, each instance assigns the following variables:

- p: the principal executing this instance
- l: a fresh locally unique identifier for the instance at the principal p
- $role$: initiator or responder
- $params$: public session parameters, with the following distinguished fields, any of which may potentially be left unassigned ($\bot$)
  - c_i: the credential of the initiator
  - c_r: the credential of the responder
  - sid: a global session identifier
  - cb: a channel binding value computed for the current protocol instance
  - cb_{in}: a channel binding value for the underlying (previous, outer) protocol instance (if any)
- $secrets$: session-specific secrets, with the following distinguished field, potentially unassigned ($\bot$):
  - sk: an authentication (MAC or authenticated encryption) key created during the protocol
- $complete$: a flag ($\in \{0, 1\}$) that indicates whether the instance has completed its role in the protocol or not.

The principal name (p) and local identifier (l) are abstract values that do not appear in the protocol; we use them to state security properties about our protocol models. The protocol itself may assign one or both credentials (c_i, c_r), and may generate a global session identifier (sid) for use at both initiator and responder. It may generate a channel binding value (cb), and if the protocol is being run within an authenticated channel, it may also exchange a channel binding value (cb_{in}) for the outer channel.

When the initiator and responder credentials are both unassigned ($c_i = c_r = \bot$), the protocol instance is said to be *anonymous*; if only one of them is unassigned, the instance is called *unilateral*; otherwise the instance is said to be *mutually authenticated*. If the instance key is assigned ($sk \neq \bot$), then the instance is said to be *key generating*.

A. Threat Model

We consider a standard symbolic attacker model in the style of Dolev and Yao [20], as is commonly used in the formal analysis of cryptographic protocols, using tools like ProVerif [18]. The attacker controls the network and hence is able to read, modify, and inject any unencrypted message.

In addition, the attacker has access to a set of compromised credentials, marked by an event $\texttt{Compromise}(c)$, which may be used both by the attacker and by honest principals (who may not know that their credential has been compromised). In any given protocol, we say that the initiator or responder credential is *honest* if it is defined ($\neq \bot$) and has not been compromised. The attacker may also selectively compromise short-term session secrets, such as the session key sk; we mark the theft of a secret s by an event $\texttt{Leaked}(s)$.

Conversely, we assume that these compromise events are the only way the attacker can obtain any long-term or short-term secret; he cannot, for example, guess the value of a secret, even if it is a short password. Moreover, following Dolev and Yao, we assume that the underlying cryptography is perfect: we model each cryptographic primitive as an abstract symbolic function with strong properties. For example hash functions are irreversible (one-way) whereas encrypted values can only be reversed (decrypted) with the correct key.

For protocols that use a Diffie-Hellman (DH) key exchange, the attacker may try to either use a bad DH group (e.g. one with small subgroups) or may send an invalid public key (one that does not belong to the right group.) This attack vector is usually not considered in typical protocol analyses, but as we will see in Section III-B, it is practical for many protocols and often leads to serious attacks on compound authentication. In Section V, we show how to encode this more general Diffie-Hellman threat model in ProVerif. We treat Elliptic Curve Diffie Hellman (ECDH) protocols analogously.

Credential compromise ($\texttt{Compromise}(c)$) is a standard feature of formal protocol analyses but, to practitioners, it may seem unrealistic to try to protect against. The attacks in this paper do not rely on this capability. However, it is an important threat to consider when evaluating countermeasures, since it can commonly occur in in real-world scenarios. Consider the example of TLS server certificates. The attacker can always obtain certificates under his own name. The challenge is to obtain a certificate that may be used to impersonate an honest server. One way is to steal a server's private key. In practice, private key theft is difficult to achieve, however there are several simpler forms of compromise that achieve the same goal. For example, the client may fail to validate server certificates correctly (e.g. see [21]), or the user may click-through certificate warnings [22]. In these cases, the attacker may be able to use his own certificate to impersonate an honest server. Alternatively, the attacker may be able to exploit a badly-configured certification authority to obtain a *mis-issued certificate* under the honest server's name [6], [23], [24].

B. Security Goals

For each individual authentication protocol, the goal is agreement on (some subset of) both the public protocol parameters and the session secrets. While the precise definition of agreement depends on the protocol being considered, it can be informally stated as follows:

Definition 1 (Agreement): If a principal a completes protocol instance l, and if the peer's credential in l is honest, and if the session secrets of l have not been leaked, then there exists a principal b with a protocol instance l' in the dual role that agrees with l on the contents of $params$ and any shared session secrets (most importantly sk).

In particular, l and l' must typically agree on each other's credentials, the session identifier sid and channel binding cb, and any negotiated cryptographic parameters. We do not explicitly state the confidentiality goal for $secrets$, but many derived authentication properties such as compound authentication implicitly depend on the generated sk being confidential.

When composing a set of protocols, besides getting individual agreement on each protocol's parameters, we also require joint agreement on all the protocols. Informally:

Definition 2 (Compound Authentication): If a principal a completes a compound authentication protocol consisting of protocol instances $\{l_1, \ldots, l_n\}$, such that some instance l_i has an honest peer credential and the session secrets of l_i have not been leaked, then there exists a principal b with protocol instances $\{l'_1, \ldots, l'_n\}$ such that each l'_j has the dual role to l_j and agrees with l_j on $params_j$ and sk_j.

In other words, a compound authentication protocol composes a set of individual authentication protocols in a way that guarantees that the same peer principal participated in all the protocols. The strength of the definition is that it requires this guarantee even if all but one of the peer credentials were compromised (or anonymous). In particular, compound authentication protects against a form of *key compromise impersonation*: even if a server's transport-level credential is compromised, the attacker cannot impersonate an honest user at the application level.

Other weaker variations of this definition may be more appropriate for a particular compound authentication protocol. For example, the definition of security for TLS renegotiation [15] states that if the peer credential in the last protocol instance l_n is honest then there must be agreement on all previous protocol instances. Conversely, as we shall see, compound authentication for SSH re-exchange requires that the session key sk_1 of the first protocol instance l_1 is never leaked. Furthermore, some protocols guarantee joint agreement only on certain elements of $params_i$, such as the peer credentials, not on their full contents.

C. Compound Authentication Protocol Examples

We now discuss several examples of compound authentication protocols (and their variations) and show how they fit in our formal model. Formalizing these varied protocols in a uniform setting allows us to compare their security guarantees and serves as the basis for the ProVerif models of Section V.

1) TLS-RSA+SCRAM: Our first example uses the TLS protocol to establish a transport channel and then runs a SASL user authentication protocol called Salted Challenge Response Authentication Mechanism (SCRAM) [25]. For compound authentication, SCRAM relies on the `tls-unique` channel binding defined in [11].

TLS supports different key exchange mechanisms; we refer to the RSA encryption based key exchange as TLS-RSA. In TLS-RSA, the server credential (c_r) is an X.509 certificate containing an RSA public key used for encryption. The client can optionally authenticate via an X.509 certificate for signing; here we assume that it remains anonymous ($c_i = \bot$).

Figure 4 depicts the full protocol flow. The client and server first exchange their local identifiers, (nonces cr, sr) and the server sets a session id sid. At this stage, protocol version and cipher suite ($nego$) are also negotiated. The server then sends its certificate $cert_S$ which is verified by the client. The client follows by sampling a random pre-master secret pms which is encrypted under pk_S and sent to the server. The client and server then compute a shared master secret $ms = kdf_1^{TLS}(pms, cr, sr)$ and a session key $sk = kdf_2^{TLS}(ms, cr, sr)$. After the client and server finished messages are exchanged and their content checked by each peer, both instances complete and create a new TLS session with the following assignments:

$$\begin{aligned}
&params = (c_i = \bot, c_r = cert_s, cr, sr, nego)\\
&secrets = (pms, ms, sk)\\
&s_r = privkey(cert_s)\\
&cb = H(log_1)
\end{aligned}$$

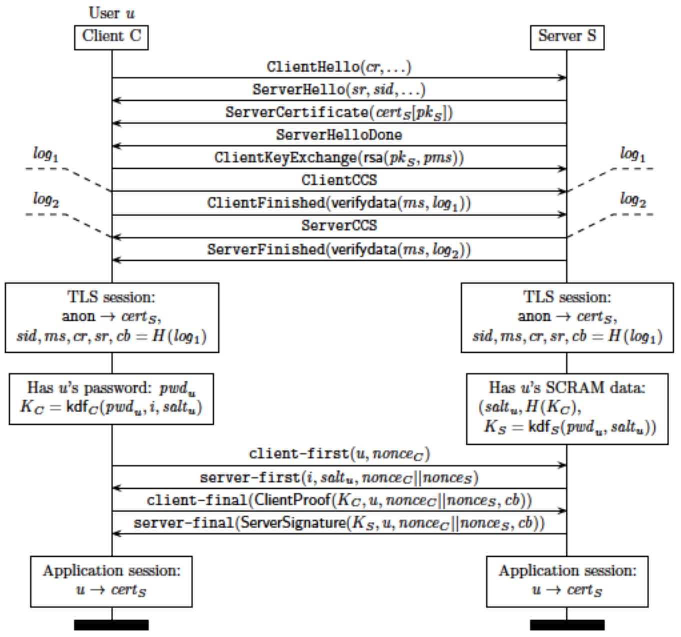

User u
Client C
Server S
ClientHello($cr, \ldots$)
ServerHello($sr, sid, \ldots$)
ServerCertificate($cert_S[pk_S]$)
ServerHelloDone
log_1
ClientKeyExchange(rsa(pk_S, pms))
log_1
ClientCCS
log_2
ClientFinished(verifydata(ms, log_1))
log_2
ServerCCS
ServerFinished(verifydata(ms, log_2))
TLS session: anon $\rightarrow cert_S$, $sid, ms, cr, sr, cb = H(log_1)$
TLS session: anon $\rightarrow cert_S$, $sid, ms, cr, sr, cb = H(log_1)$
Has u's password: pwd_u $K_C = \mathsf{kdf}_C(pwd_u, i, salt_u)$
Has u's SCRAM data: $(salt_u, H(K_C), K_S = \mathsf{kdf}_S(pwd_u, salt_u))$
client-first($u, nonce_C$)
server-first($i, salt_u, nonce_C \| nonce_S$)
client-final(ClientProof($K_C, u, nonce_C \| nonce_S, cb$))
server-final(ServerSignature($K_S, u, nonce_C \| nonce_S, cb$))
Application session: $u \rightarrow cert_S$
Application session: $u \rightarrow cert_S$

Fig. 4. The TLS-RSA+SCRAM compound authentication protocol.

According to the `tls-unique` specification, a channel binding cb is set to a hash of the transcript of all messages before the `ClientCCS` message.

The SCRAM protocol then runs on top of the TLS connection, performing password-based user authentication. Before the protocol runs, as part of the user registration process at S, S generates a random salt $salt_u$ and an iteration count i and asks u to derive two keys K_C and K_S from its password pwd_u. The server key K_S and a hash of the client key ($H(K_C)$) are stored at S, but the raw password or client key are not.

In the first message of SCRAM, the client sends its username u and a fresh $nonce_C$; the server responds with its own fresh $nonce_S$, the iteration count i, and the salt $salt_u$ from which the client key K_C can be recomputed. The client then sends a message that proves its possession of K_C and binds the key to the username, nonces, and the TLS channel binding. The server terminates the protocol sending a similar message, showing it knows the server key K_S. By the end of the protocol, we have agreement on:

$$\begin{aligned}
&cb_{in} = H(log_1)\\
&params' = (c_i = u, c_r = \bot, nonce_C, nonce_S, cb_{in})\\
&s_i = pwd_u, \quad s_r = (H(K_C), K_S)
\end{aligned}$$

The compound authentication goal for the composite TLS-RSA+SCRAM protocol is in two parts (one for each direction):

- If the server credential $cert_S$ is honest, and if a client principal a completes TLS-RSA+SCRAM, then there exists a server principal b running TLS-RSA+SCRAM, which has the same TLS $params$ and sk and the same SCRAM $params'$ as a.
- If the user's credential u is honest (pwd_u is secret), and if a server principal b completes TLS-RSA+SCRAM, then there is a client principal a running TLS-RSA+SCRAM, which has the same TLS $params$ and sk and the same SCRAM $params'$ as b.

Notably, the first goal holds even if the user's password (and therefore, the keys K_C, K_S) is compromised, and the second goal holds even if the server's certificate $cert_S$ is compromised. That is, user credential forwarding and server key compromise impersonation are both prevented.

Other TLS key exchange variants: There are various other key exchanges supported by TLS which may be used in place of TLS-RSA in the above protocol. In all these protocols, the computation of cb, ms, and sk remains the same. The main differences are the computation of pms and the choice of client and server credentials. The definition of compound authentication remains the same (adapted to the appropriate notion of credential compromise).

In TLS-DHE, the server and optional client credentials are both X.509 certificates used for signing. The pms is obtained using a Diffie-Hellman agreement between the client and server, over a prime order group whose parameters (prime π, generator g) are chosen and signed by the server.

$$\begin{aligned}
&params = ([c_i = cert_c], c_r = cert_s, cr, sr, nego, cb,\\
&\qquad\qquad \pi, g, g^x \ mod\ \pi, g^y \ mod\ \pi)\\
&secrets_i = (x, pms = g^{xy} \ mod\ \pi, ms, sk)\\
&secrets_r = (y, pms = g^{xy} \ mod\ \pi, ms, sk)\\
&s_i = privkey(cert_c), \quad s_r = privkey(cert_s)
\end{aligned}$$

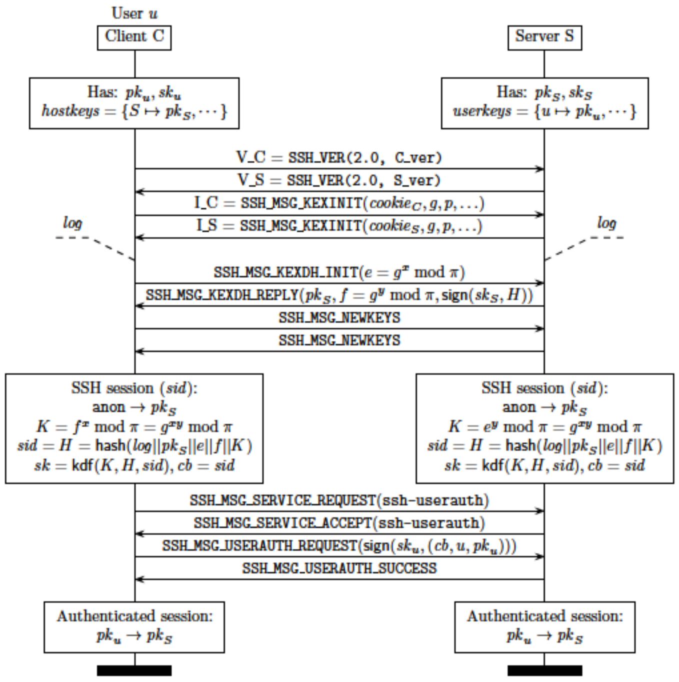
In TLS-ECDHE, the exchange is similar to TLS-DHE, except that the Diffie-Hellman group is represented by a named elliptic curve n and public keys are represented by points on the curve. TLS supports several elliptic curves, and more are being considered for standardization.

TLS-SRP uses the Secure Remote Password (SRP) protocol to authenticate the user with a password while protecting the exchange from offline dictionary attacks. The protocol relies on a fixed Diffie-Hellman group (π, g). The client credential refers to a username (u) and salted password (x_u) and the server credential refers to a password verifier value ($v_u = g^{x_u} \ mod\ \pi$). The pms is calculated using the SRP protocol.

$$\begin{aligned}
&params = (c_i = u, c_r = \bot, cr, sr, nego, cb,\\
&\qquad\qquad \pi, g, A = g^a \ mod\ \pi, B = (g^b + kv_u) \ mod\ \pi,\\
&\qquad\qquad h = hash(A\|B))\\
&secrets_i = (a, pms = g^{b(a+hx_u)} \ mod\ \pi, ms, sk)\\
&secrets_r = (b, pms = g^{b(a+hx_u)} \ mod\ \pi, ms, sk)\\
&s_i = x_u, \quad s_r = v_u
\end{aligned}$$

2) SSH User Authentication: A session of the SSH protocol consists of a key exchange protocol composed with a user authentication protocol, as depicted in Figure 5.

In the SSH key exchange protocol, the initiating principal is a user and the responding principal is a host that the user wishes to log on to. The two principals first exchange nonces n_i, n_r (called *cookies* in SSH), Diffie-Hellman public keys $g^x \, mod\, \pi, g^y \, mod\, \pi$ in some group (π, g), and other negotiation parameters $nego$. The host is authenticated with a public key $c_r = pk_S$ that is assumed to be known to the client. In the key exchange, the user is unauthenticated ($c_i = \bot$). At the end of the protocol, each instance produces an exchange hash H. Over a single connection, the SSH key exchange protocol can be run several times, each time generating a



User u
Client C
Server S
Has: pk_u, sk_u
$hostkeys = \{S \mapsto pk_S, \cdots\}$
Has: pk_S, sk_S
$userkeys = \{u \mapsto pk_u, \cdots\}$
V_C = SSH_VER(2.0, C_ver)
V_S = SSH_VER(2.0, S_ver)
I_C = SSH_MSG_KEXINIT($cookie_C, g, p, \ldots$)
I_S = SSH_MSG_KEXINIT($cookie_S, g, p, \ldots$)
log
log
SSH_MSG_KEXDH_INIT($e = g^x \bmod \pi$)
SSH_MSG_KEXDH_REPLY($pk_S, f = g^y \bmod \pi$, sign(sk_S, H))
SSH_MSG_NEWKEYS
SSH_MSG_NEWKEYS
SSH session (sid):
anon $\rightarrow pk_S$
$K = f^x \bmod \pi = g^{xy} \bmod \pi$
$sid = H = \mathsf{hash}(log \| pk_S \| e \| f \| K)$
$sk = \mathsf{kdf}(K, H, sid)$, $cb = sid$
SSH session (sid):
anon $\rightarrow pk_S$
$K = e^y \bmod \pi = g^{xy} \bmod \pi$
$sid = H = \mathsf{hash}(log \| pk_S \| e \| f \| K)$
$sk = \mathsf{kdf}(K, H, sid)$, $cb = sid$
SSH_MSG_SERVICE_REQUEST(ssh-userauth)
SSH_MSG_SERVICE_ACCEPT(ssh-userauth)
SSH_MSG_USERAUTH_REQUEST(sign($sk_u, (cb, u, pk_u)$))
SSH_MSG_USERAUTH_SUCCESS
Authenticated session:
$pk_u \rightarrow pk_S$
Authenticated session:
$pk_u \rightarrow pk_S$

Fig. 5. The SSH user authentication protocol.

fresh exchange hash. The exchange hash of the first key exchange happening over a connection is called the *session id* (sid), and it remains constant over the life of a connection. The authenticated encryption key for the current instance is computed as $sk = kdf^{SSH}(g^{xy} \ mod \ \pi, H, sid)$.

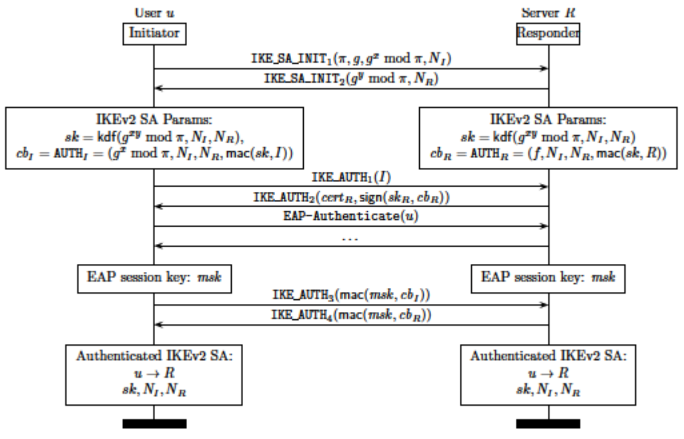
$$\begin{aligned}
&params = (c_i = \bot, c_r = pk_S, n_i, n_r, nego,\\
&\qquad \pi, g, g^x \ mod \ \pi, g^y \ mod \ \pi, H, sid)\\
&secrets_i = (x, g^{xy} \ mod \ \pi, H, sid, sk)\\
&secrets_r = (y, g^{xy} \ mod \ \pi, H, sid, sk)\\
&s_i = \bot, s_r = sk_S\\
&cb = sid = H
\end{aligned}$$

The SSH user authentication protocol is layered above the key exchange protocol. Figure 5 depicts the certificate-based user authentication protocol, where the client signs a block containing the username u and the sid with a private key sk_u assigned to the user (whose public key pk_u is known to the server). No new secrets are generated.

$$\begin{aligned}
&params' = (c_i = pk_u, c_r = \bot, sid)\\
&s_i = sk_u, s_r = \bot
\end{aligned}$$

The compound authentication goal for SSH user authentication can be written out very similarly to TLS-RSA+SCRAM. The user and host obtain a mutual authentication guarantee: if the peer's credential is honest, then both principals agree on the SSH key exchange $params$ as well as the user credential.

3) IKEv2+EAP: IKEv2 offers several authentication modes for the initiator and responder. They may authenticate each other with pre-shared keys, or with certificates, or the responder may use a certificate while the initiator uses an Extensible Authentication Protocol (EAP). In all these cases, the two instances first engage in the `IKE_SA_INIT` anonymous Diffie-Hellman key exchange protocol and then perform a the `IKE_AUTH` protocol within the established channel. Figure 6 depicts the EAP variant.



User u
Initiator
Server R
Responder
IKE_SA_INIT$_1$($\pi, g, g^x \bmod \pi, N_I$)
IKE_SA_INIT$_2$($g^y \bmod \pi, N_R$)
IKEv2 SA Params:
$sk = \mathsf{kdf}(g^{xy} \bmod \pi, N_I, N_R)$,
cb_I = AUTH$_I$ = ($g^x \bmod \pi, N_I, N_R$, mac(sk, I))
IKEv2 SA Params:
$sk = \mathsf{kdf}(g^{xy} \bmod \pi, N_I, N_R)$
cb_R = AUTH$_R$ = (f, N_I, N_R, mac(sk, R))
IKE_AUTH$_1$(I)
IKE_AUTH$_2$($cert_R$, sign(sk_R, cb_R))
EAP-Authenticate(u)
...
EAP session key: msk
EAP session key: msk
IKE_AUTH$_3$(mac(msk, cb_I))
IKE_AUTH$_4$(mac(msk, cb_R))
Authenticated IKEv2 SA:
$u \rightarrow R$
sk, N_I, N_R
Authenticated IKEv2 SA:
$u \rightarrow R$
sk, N_I, N_R

Fig. 6. The IKEv2+EAP compound authentication protocol.

In the first two messages, the initiator and responder exchange nonces (n_i, n_r), Diffie-Hellman parameters ((π, g)) and public keys ($g^x \ mod \ \pi$, $g^y \ mod \ \pi$), along with other protocol specific negotiation parameters ($nego$). The Diffie-Hellman shared secret is used to protect the subsequent mutual authentication protocol and create an authenticated encryption key $sk = kdf^{IKEv2}(g^{xy} \ mod \ \pi, n_i, n_r)$.

$$\begin{aligned}
&params = (c_i = \bot, c_r = cert_R, n_i, n_r, nego,\\
&\qquad \pi, g, g^x \ mod \ \pi, g^y \ mod \ \pi, AUTH_i, AUTH_r)\\
&AUTH_I = (g^x \ mod \ \pi, n_i, n_r, \mathsf{mac}(g^{xy} \ mod \ \pi, I))\\
&AUTH_R = (g^y \ mod \ \pi, n_i, n_r, \mathsf{mac}(g^{xy} \ mod \ \pi, R))\\
&secrets_i = (x, g^{xy} \ mod \ \pi, sk), secrets_r = (y, g^{xy} \ mod \ \pi, sk)\\
&s_i = \bot, s_r = sk_R
\end{aligned}$$

IKEv2 does not explicitly define a global session identifier, but its authentication protocol relies on two values $AUTH_I$ and $AUTH_R$, as defined above, that are used as channel bindings for the subsequent `IKE_AUTH` protocol.

In the EAP case depicted in Figure 6, in the first two messages of `IKE_AUTH`, the initiator sends its identity I but not its certificate, whereas the responder sends it certificate $cert_R$ and a signature over $AUTH_R$ with its private key sk_R. Then the initiator and responder begin a sequence of EAP request and response messages [7] in order to authenticate the user u (and potentially re-authenticate the server R). EAP embeds many authentication methods, ranging from weak password-based protocols like MSChapv2 and fully-fledged authenticated key-exchange protocols like TLS and IKEv2. At the end of the EAP exchange, the responder R has authenticated u and has generated a new EAP master session key msk.

To complete the `IKE_AUTH` protocol, the initiator and responder exchange MACs over $AUTH_I$ and $AUTH_R$ respectively, keyed with msk.

$$\begin{aligned}
&params = (c_i = u, c_r = \bot, cb_{in} = (AUTH_i, AUTH_r))\\
&secrets_i = (sk = msk), secrets_r = (sk = msk)\\
&s_i = cred_u, s_r = \bot
\end{aligned}$$

The final two messages cryptographically bind the `IKE_AUTH` authentication protocol to the `IKE_SA_INIT` key exchange to obtain the usual compound authentication guarantee.

4) Other Bindings: EAP, `tls-server-end-point`*:* The three previously described compound authentication protocols are only a few of the many possible combinations between transport protocols and application-level authentication.

Many protocols compose TLS with EAP methods [7], [8], [9] and in response to previous man-in-the-middle attacks [1] on such protocols, many EAP methods have been extended with a form of channel binding called *cryptographic binding* [2]. The idea is to use the master secret and random values of the TLS protocol (ms, cr, sr) as a channel binding and to derive a key by mixing it with the master session key msk and nonces $nonce_C, nonce_S$ generated by the EAP method. The resulting *compound MAC key* (cmk) is then used to cryptographically bind the EAP method to the TLS channel, by using it to create two MACs `B1_MAC` and `B2_MAC` that are exchanged in the final messages of the EAP exchange:

$$\begin{aligned}&cmk = prf^{EAP}(ms, cr, sr, msk, nonce_C, nonce_S)\\&\mathtt{B1\_MAC} = \mathsf{mac}(cmk, nonce_S)\\&\mathtt{B2\_MAC} = \mathsf{mac}(cmk, nonce_C)\end{aligned}$$

Some channel bindings have more modest compound authentication goals. For example, the `tls-server-end-point` channel binding [11] only aims to ensure that the application level protocol instances agree on the transport-level server certificate. In this case, the channel binding cb for TLS consists of the hash of the TLS server certificate $(H(cert_S))$. This binding is used, for example, when binding SAML assertions to the underlying TLS channel [26], so that a SAML assertion generated for use at one server may not be used at another, unless the two servers share the server certificate.

Re-keying and resumption: Many of the authentication protocols described above also offer a re-keying protocol, by which the session key sk generated by the protocol can be refreshed without the need for full re-authentication of the client and the server. Re-keying is mainly useful on connections where a lot of data is exchanged, so that the compromise of a session key is of limited benefit to the attacker. For example, SSH recommends that keys be refreshed every hour, or for every gigabyte of data.

Re-keying protocols may also be used to perform fast *session resumption*. If an initiator and responder already have a channel between them with a session key sk, they may reuse the session key to start a new channel without the need to repeat the full key exchange. Such session resumption protocols are included within TLS, and are available as extensions to IKEv2 [27], SSH [28], and EAP [29]. Session resumption can have a major impact on the performance of a client or a server since it skips many of the expensive public-key operations of a full key exchange. For example, the vast majority of TLS connections between web browsers and major websites like Google perform session resumptions rather than full key exchanges.

A full key exchange followed by re-keying or resumption can be treated as a compound authentication protocol, except that the re-keying protocol does not change the client or server credentials. Instead, it simply performs a key confirmation of the previous session key sk and generates a new session key sk'. For example, in TLS resumption, the new key is computed from the old master secret plus the new random nonces generated by the client and the server:

$$\begin{aligned}&params' = (c_i = c_r = \bot, cr', sr', sid, nego)\\&secrets' = sk' = kdf_2^{TLS}(ms, cr', sr')\end{aligned}$$

The compound authentication goal for re-keying is that if the session secrets and peer credentials in the original session are not compromised, then the two principals agree upon both the old and new session parameters $(params, params')$ and session keys (sk, sk').

Re-exchange and re-authentication: In addition to re-keying, many key exchange protocols also allow the initiator and responder to perform a second key-exchange to re-authenticate each other. In TLS, this is called renegotiation while in SSH it is called re-exchange. For IKEv2, there is a proposed extension that allows re-authentication in the style of TLS [30].

The TLS renegotiation is a full key exchange and both the client and server may authenticate themselves using credentials that differ from the previous exchange. This feature was famously subject to a man-in-the-middle attack [3], [4] and in response to this attack all TLS libraries implement a mandatory channel binding countermeasure [12] that binds the renegotiation key exchange to the the transcript of the previous handshake. More precisely, each TLS handshake generates a channel binding of the form:

$$cb = (\mathsf{verifydata}(log_1, ms), \mathsf{verifydata}(log_2, ms))$$

The subsequent handshake agrees on this channel binding value, and by including it in the key exchange, the chain of channel bindings on a connection guarantees agreement on the full sequence of protocol assignments on a connection [15].

The SSH re-exchange is also a full server-authenticated key exchange where the server's host key and other parameters may be different from the previous exchange. Unlike TLS, however, SSH uses the sid, that is the hash H of the first exchange on the connection, as a channel binding for all subsequent key exchanges on the connection. In particular, during the second SSH key exchange, a new set of parameters and secrets are generated, but the session id does not change. Hence, the new session key is computed as

$$sk = kdf^{SSH}(g^{xy}\ mod\ \pi, H', sid)$$

where H' is the hash computed during the new exchange the $sid$is still the hash computed in the first exchange.

The proposed re-authentication protocol for IKEv2 [30] is inspired by TLS renegotiation and treats the $AUTH_I$ and $AUTH_R$ payloads as channel bindings for re-authentication. It runs a new `IKE_SA_INIT` protocol and within this protocol and a new `IKE_AUTH` protocol that binds the initiator and responder credentials to the $AUTH_I$ and $AUTH_R$ payloads of the previous IKEv2 session.

III. Channel Synchronization Attacks

In the previous section, we described a number of compound authentication protocols that implement the channel binding pattern of Figure 3 in order to prevent man-in-the-middle attacks like the one in Figure 2. Now we will evaluate a number of these channel binding mechanisms to see if they succeed in preventing such attacks.

A channel binding countermeasure only works if the channel binding values for independent protocol sessions are different. Hence, we observe that if the man-in-the-middle attacker manages to *synchronize* the channel bindings on its protocol sessions to two different principals, it can re-enable the credential forwarding attack. We call such attacks *channel synchronization attacks*. More generally, if two principals engage in a sequence of protocols, we say that they are subject to a channel synchronization attack if the channel binding generated by the final protocol is the same at both principals and each principal used an honest credential to authenticate itself (somewhere in the protocool sequence), but the two principals do not agree on some protocol parameter.

A channel synchronization attack typically leads to an impersonation attack on compound authentication after one more protocol, since agreement on the final channel binding no longer guarantees agreement on all previous protocol instances. It may be easier to understand such attacks by example, and we shall see several concrete examples below.

A. Triple Handshake Attacks on TLS

The triple handshake attacks [16] show that a number of TLS channel bindings fail to prevent man-in-the-middle attacks. For the full details of the attacks we refer the reader to the original paper and to our ProVerif models. Here, we summarize their impact and identify their general principles.

First, they show that the master secrets ms in TLS-RSA and TLS-DHE can be synchronized across two different TLS connections. This means that the EAP cryptographic binding (based on (ms, cr, sr)) does not provide compound authentication and still leads to man-in-the-middle attacks on protocols like PEAP, EAP-TTLS, and EAP-FAST. The fact that RSA-based key transport protocols like TLS-RSA allow key synchronization was well-known (see e.g. the famous attack on the Needham Schroeder public-key protocol [31]). However, the impact of key synchronization on compound authentication was identified only in [16]. The key synchronization attack on TLS-DHE is more surprising since Diffie-Hellman key exchanges are expected to be *contributive*: both parties contribute to the established key. But TLS allows servers to choose arbitrary DH groups, even ones with non-prime orders, and clients do not validate the group, enabling the attack.

Second, they show that the handshake transcripts (log_1, log_2) can be synchronized across two different TLS connections that use session resumption after an initial TLS-RSA or TLS-DHE key exchange. In particular, this means that the `tls-unique` channel binding (derived from the transcript) can be synchronized after session resumption. Hence, if we run SCRAM after resuming a TLS-RSA session, a credential forwarding attack on SCRAM becomes possible, despite its use of channel bindings. Moreover, the channel binding used by TLS renegotiation $(\mathsf{verifydata}(log_1, ms), \mathsf{verifydata}(log_2, ms))$ can also be synchronized and hence mutually authenticated TLS renegotiation after session resumption is also subject to a MitM impersonation attack. Since the transcripts of the two connections have been synchronized, it means that countermeasures such as [6], [14], [5] are also broken after session resumption.

The triple handshake attacks had a strong impact: fixes to major TLS libraries and web browsers and a new protocol-level countermeasure that is being standardized and implemented as a protocol extension [19]. In the rest of this section, we investigate whether such synchronization attacks apply to other key exchanges used within TLS, IKEv2 and SSH.

B. Key Synchronization via Small Subgroup Confinement

Diffie-Hellman key exchange protocols are based on prime-order groups, typically written (π, q, g) where q is a prime less than π and g generates a q-order subgroup of $[1..p-1]$. All participants are expected to choose private keys in the range $[1..q-1]$. However, such protocols are known to be vulnerable to various attacks when the group has small subgroups (see e.g. [32]). In particular, we show that small subgroups can be exploited for key synchronization.

For all π, there is at least two subgroups of size 1 ($\{0\}, \{1\}$) and one subgroup of size 2 ($\{1, p-1\}$). So, if one of the participants chooses a Diffie-Hellman public key of 0, no matter what exponent y the other participant chooses, the resulting shared secret will be $0^x \ mod \ \pi = 0$. Similarly, by choosing 1 or $p-1$ as a public key, one of the participants of the key exchange can force the shared secret to be a fixed value, no matter what the other participant chose. This is called a *small subgroup confinement* attack: rather than honestly choosing a public key in the q-order subgroup, a malicious participant can force its peer to compute in a smaller subgroup where the resulting shared secrets are predictable (or at least guessable from a small set of values).

We advocate that, in order to eradicate such attacks, both participants should validate the groups and public keys they receive, say using the rules in [33]. The tests ensure that the public key is in the q-order subgroup and is not equal to 1. Still many protocol implementations do not perform these checks: either because the protocol itself does not provide enough information (e.g. a TLS server provides the generator g and the prime π, but not the order q); or for efficiency (the checks require an exponentiation by q); or because it is commonly believed that small subgroup confinement attacks only matter when keys are reused [34]. We show that these attacks can break compound authentication even if keys are never reused.

1) Key Synchronization in IKEv2: IKEv2 can be used with a number of well-known MODP groups including the groups 22-24 that have many small subgroups [35]. However, the specification for IKEv2 public-key validation [34] only requires implementations to check for 0, 1 and $p-1$, but does not require it to check that the public key is in the q-order subgroup, as long as it does not reuse private exponents. Indeed, a number of open source IKEv2 implementations that implement these groups skip the q-order check. This leads to the following key synchronization attack.

Suppose an initiator I connects to a malicious responder M, which then in turn connects to an honest responder R. During the `IKE_SA_INIT` key exchange, M forwards messages between I and R but it uses its own Diffie-Hellman public key. M chooses as its public key a generator g' of a small k-order subgroup and sends it to both I and R. Consequently the resulting Diffie-Hellman shared secrets on both connections is in the k-order subgroup and there is a $1/k$ chance of both secrets being the same.

Since M has also synchronized the nonces N_I and N_R, the session key sk on both connections also has a $1/k$ chance of being the same. So any compound authentication protocol that relies on a channel binding derived from (sk, N_I, N_R) (as proposed in [36]) is vulnerable to a man-in-the-middle attack.

2) Key Synchronization in ECDHE with Curve25519: The named elliptic curves used with TLS and other protocols typically do not have any small subgroups, but there are many new proposals and prototype implementations that use Curve25519 [37], because its implementations are faster and because it does not require any public key validation (all 32-byte strings are said to be valid public keys). However, Curve25519 has a subgroup of size 8, and hence there are 12 points that fall in small subgroups. Yet, implementations of the curve typically do not forbid these values, trusting that "these exclusions are unnecessary for Diffie-Hellman".[1]

Hence, if a client C and server S both allow Curve25519 public keys in the 8-order subgroup, a man-in-the-middle M can mount a key-synchronization attack to obtain the same key on two connections with probability at least $1/8$. Consequently, TLS-ECDHE with Curve25519 also becomes vulnerable to the first stage of the triple handshake attacks.

More generally, checking that a public key point lies on a curve is quite efficient (one scalar multiplication) and we advocate that this check should always be performed, otherwise a similar attack becomes possible on any curve.

3) Key Synchronization in SRP: The SRP protocol uses a Sophie-Germain prime π that has only the usual small subgroup values $0, 1, p-1$. The initiator and responder exchange two values $A = g^a \ mod \ \pi$ and $B = (g^b + kv_u) \ mod \ \pi$ where $v_u = g^{x_u} \ mod \pi$ is the password verifier. The SRP specification says that A and B must not be 0 but does not otherwise require any public key validation. Indeed the OpenSSL implementation of TLS-SRP does not perform any additional checks on A and B. This leads to a key synchronization attack.

Suppose a malicious server M registers its own username and password at S and suppose it chooses $x_u = 0$; that is, the verifier $v_M = 1$. Now, suppose the client C connects to M using SRP. M chooses $B = 1 + kv_u$ (i.e. $b = 0$) so that the resulting session key $sk = g^{b(a+hx_u)} = 1$. Meanwhile, suppose M separately connects to S using its own credential x_M, and chooses $A = 1$ ($a = 0$). Again, on this connection the resulting session key $sk = g^{b(a+hx_u)} = 1$. The two connections have different client and server credentials, but the resulting session key is the same. Consequently, using TLS-SRP in the initial handshake also leads to the triple handshake attacks.

C. Transcript Synchronization via Session Resumption

A number of compound authentication protocols use the transcript of the previous (outer) authentication protocol as a channel binding. For example, both TLS renegotiation and the `tls-unique` binding use a channel binding derived from the TLS handshake log. IKEv2 authentication and re-authentication both use $AUTH$ payloads derived from the preceding `IKE_SA_INIT` transcript as a channel binding. In contrast, SSH only uses the transcript of the first exchange on the connection, not the most recent exchange.

Protocols that rely on transcript for channel bindings must be wary of session resumption, since the transcript of a resumption (or re-keying) handshake is necessarily abbreviated and does not authenticate all the session parameters. For example, the transcripts of both TLS and IKEv2 resumption only guarantee agreement on the previous session keys sk, but not on other parameters. Consequently, like TLS resumption, IKEv2 resumption leads to a transcript synchronization attack.

Suppose a man-in-the-middle M has managed to implement a key synchronization attack across two connections as described above, one from C to M and the other from M to S. At the end of this key exchange, the values (sk, N_I, N_R) on the two connections are the same. Now suppose C resumes its session with M and M resumes its session with S. M can simply forward the `IKE_SA_INIT` and `IKE_AUTH` messages of session resumption between C and S since the original session keys are the same. M will not know the new session keys, but at the end of the resumption exchange, the two authentication payloads (channel bindings) $AUTH_I$ and $AUTH_R$ are the same (even though the identities and credentials used in the original key exchange were different.) Consequently, if this channel binding is used in a subsequent user authentication protocol or by IKEv2 re-authentication, it will lead to a man-in-the-middle credential forwarding attack.

In other words, we have reconstructed a variant of the TLS triple handshake attack on the composition of IKEv2, IKEv2 session resumption and IKEv2 re-authentication. The impact of this attack is not as strong as the TLS attack since both IKEv2 re-authentication and IKEv2 channel bindings are not yet widely implemented or used.

D. Breaking Compound Authentication for SSH Re-Exchange

The SSH re-exchange protocol uses the session id sid as a channel binding, where sid is derived from the transcript of the first key exchange on the connection. Consequently, each exchange on an SSH connection is bound to the first exchange; however, these subsequent exchanges are not bound to each other. This is in contrast to the TLS renegotiation countermeasure [12] which chains together the whole sequence of key exchanges on a given connection.

We show that a sequence of three SSH exchanges may break compound authentication, if the attacker succeeds in compromising the session secrets of the first exchange.

The protocol flow that exhibits the vulnerability is depicted in Figure 7. Suppose a client C executes an SSH key exchange and user authentication with a server S. Now suppose a malicious server M compromises the session key sk and session id sid (by exploiting a bug at the client or at the server, for example.) Suppose C initiates a second key exchange. Since M knows the session key, it can intercept this key exchange and return its own host key (SSH allows a change of host keys during re-exchange). At the end of the second key exchange, the session keys and other parameters at C and at S are now different, but the session id remains the same. Now, suppose C begins a third key exchange; M can

[1] http://cr.yp.to/ecdh.html

re-encrypt all messages sent by C with the previous session key sk still used by S and vice versa. At the end of this third exchange, C and S have the same keys, session parameters, and session id, and they have not detected that there was a completely different exchange injected at C in between. Since the number of protocol instances at C and S differ, our compound authentication goal is violated.

Since the attack requires session key compromise, which is difficult to mount in practice, we consider it largely a theoretical vulnerability. However, it serves to illustrate the difference between the channel bindings used by TLS renegotiation and SSH re-exchange. Furthermore, it clarifies the dangers of session key compromise in SSH. SSH session keys are supposed to be refreshed every hour, presumably since there is some danger that they may be compromised. The above attack shows that if an SSH session key is compromised when it is still in use, the attacker can exploit it for much longer than an hour; he can use any number of SSH re-exchanges to create new keys and keep the session alive at both the client and the server. Then, at any point, the attacker may step out of the middle and the client and server will continue to talk to each other without detecting any wrongdoing.

E. Summary of Attacks

In this section we recalled the triple handshake attacks on TLS and described a series of new channel synchronization attacks on compound authentication in TLS, IPsec, and SSH:

- TLS-ECDHE (Curve25519) is vulnerable to key synchronization, and hence to triple handshake attacks;
- TLS-SRP is vulnerable to key synchronization, and hence to triple handshake attacks;
- IKEv2 with groups 22-24 is vulnerable to key synchronization, and hence its unique channel binding [36] is vulnerable to channel synchronization;
- IKEv2 session resumption is vulnerable to transcript synchronization, and hence IKEv2 resumption followed by IKEv2 re-authentication is vulnerable to a MitM impersonation attack;
- SSH re-exchange is vulnerable to a triple-exchange vulnerability, if session keys may be compromised.

Not all these attacks have a practical impact, but in sum, they show that channel synchronization is an important and widespread problem for compound authentication, one deserving of formal analysis and robust countermeasures.

IV. Contributive Channel Bindings

Protocol implementations can prevent many of the key key synchronization attacks in the previous section by fully validating DH public keys [33] and by forbidding unknown DH groups and elliptic curves. Other re-authentication attacks may be prevented by forbidding the change of the peer's credential during key re-exchange. While such countermeasures may be sufficient, they do not address the core weaknesses of the channel bindings used in these protocols.

We propose a new requirement for the channel bindings generated by composite authentication protocols. We advocate

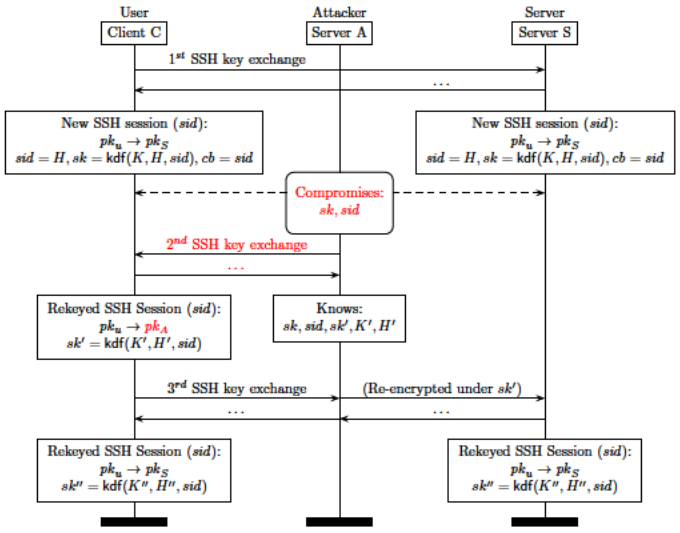

User
Client C
Attacker
Server A
Server
Server S
1st SSH key exchange
...
New SSH session (sid): $pk_u \to pk_S$ $sid = H, sk = \mathsf{kdf}(K, H, sid), cb = sid$
New SSH session (sid): $pk_u \to pk_S$ $sid = H, sk = \mathsf{kdf}(K, H, sid), cb = sid$
Compromises: sk, sid
2nd SSH key exchange
...
Rekeyed SSH Session (sid): $pk_u \to pk_A$ $sk' = \mathsf{kdf}(K', H', sid)$
Knows: sk, sid, sk', K', H'
3rd SSH key exchange
(Re-encrypted under sk')
...
...
Rekeyed SSH Session (sid): $pk_u \to pk_S$ $sk'' = \mathsf{kdf}(K'', H'', sid)$
Rekeyed SSH Session (sid): $pk_u \to pk_S$ $sk'' = \mathsf{kdf}(K'', H'', sid)$

Fig. 7. Triple Exchange Vulnerability in SSH

that the channel binding must be *contributive*, that is, it must contain contributions from each participant of the protocol. In particular, if a compound authentication protocol consists on n protocol instances $\{l_1, \ldots, l_n\}$, the channel binding of l_n must be bound to the parameters and session secrets of all n instances $\{params_1, sk_1, \ldots, params_n, sk_n\}$, so that agreement on the channel binding guarantees compound authentication for the composite protocol.

A. TLS Session Hash and Extended Master Secret

In response to the triple handshake attacks, [16] proposed a new protocol extension called the `tls-session-hash` [19] that fixes `tls-unique` and the TLS renegotiation channel binding, so that they guarantee compound authentication even when session resumption is enabled.

The idea of the session hash is inspired by the SSH session hash: for each TLS handshake, the session hash contains a hash of the transcript and is used within the key derivation function that generates the master secret:

$$h = H(log_1)$$
$$ms = kdf_1^{TLS}(pms, h)$$

Consequently, the master secret is bound to all the session parameters negotiated in the handshake and key synchronization attacks is no longer possible. Furthermore, since session resumption authenticates the ms, it also implicitly authenticates all the session parameters. We formally evaluate the effectiveness of this countermeasure in the Section V.

B. SSH Cumulative Session Hash

The SSH session id sid is a good channel binding for SSH user authentication, but it fails to provide strong compound authentication guarantees for SSH re-exchange. To address the triple-exchange vulnerability of the previous section, we propose a new contributive channel binding, inspired by the TLS renegotiation countermeasure. In the terminology of [10], we aim to define a `unique` channel binding for SSH channels that identifies the innermost SSH exchange.

The *SSH cumulative session hash* is computed as the incremental hash of the sequence of exchange hashes. Each SSH exchange includes the hash of the previous exchange H_{i-1} in the hash for the current exchange H_i. The initial exchange treats the previous exchange hash (H_0) as empty. Now, when generating the session key, we no longer need to mix in the session id, since the cumulative session hash is bound to all previous exchanges, including the first one.

$$\begin{aligned} H_0 &= \epsilon \\ H_i &= hash(log||pk_S||e||f||K||H_{i-1}) \\ sk_i &= kdf^{SSH}(K, H_i) \end{aligned}$$

In the next section, we show that this cumulative hash prevents the triple-exchange vulnerability.

C. IKEv2 Extended Session Keys

IKEv2 key derivation suffers from the same weakness as TLS, leading to similar key synchronization attacks. While the $AUTH$ payloads provide a good channel binding for EAP authentication, they are not suitable for IKEv2 resumption or re-authentication. Consequently, we propose an extended session key derivation for the `IKE_SA_INIT` protocol that derives the session key from the Diffie-Hellman shared secret, the nonces, and the public keys:

$$sk = kdf^{IKEv2}(g^{xy} \ mod \ \pi, g^{x} \ mod \ \pi, g^{y} \ mod \ \pi, N_I, N_R)$$

Much like the TLS session hash, this modification ensures that the IKEv2 session key is context bound to all the `IKE_SA_INIT` parameters, and hence prevents key synchronization attacks, prevents transcript synchronization during resumption, and fixes the unique channel binding [36].

V. Formal Analysis with ProVerif

A. Presentation of the Model

We write our protocol models in the input language of ProVerif [18] and we refer to its manual for the full syntax. Here, we only describe the salient features of our models.

Cryptographic library: Asymmetric-key encryption and digital signature primitives are modeled in the standard symbolic (Dolev-Yao) style. The terms aenc(pk(s),p) and adec(s,c) represent asymmetric encryption and decryption, where s is a private key, pk(s) its public part and p the plaintext. Their behavior is defined by the single equation adec(s,aenc(pk(s),p)) = p. Hence, a plaintext encrypted with public key pk(s) can be recovered only if the private key s is available. Similarly, signatures are written sign(s,d) and they can be verified by using the equation check(pk(s),d,sign(s,d)) = true. This model implicitly excludes collisions between different function symbols, so an asymmetric encryption and a signature cannot return the same value, even if the same key-pair is used for both operations.

In many protocols, authenticated encryption is obtained by composing symmetric-key encryption with a message authentication scheme. In our model, we abstract over these compositions and model a perfect authenticated encryption scheme via the equation ad(k, ae(k,p)) = p where ae(k,p) and ad(k,c) are the authenticated encryption and decryption functions respectively and k is a symmetric key and p is a plaintext.

One way functions such as hashes and key derivation functions are modeled as terms hash(x), kdf(k,x) without additional equations. In particular, they cannot be inverted.

As indicated in our threat model of Section II, we define DH key agreement in the presence of bad groups and keys. We start by defining a standard core DH model that only handles good keys and one static good group. The following equation captures the core DH property

```
E(E(G,x),y) = E(E(G,y),x)
```

where E(e,x) represents the DH modular exponentiation function, G is the static good DH group, and x,y are honestly generated keys. This simple equation was adequate to analyze our models and find the attacks we were interested in, but for more precise analyses of DH protocols one would need to use more elaborate encodings for exponentiation [38], or tools that provide specialized DH support (e.g. [39]).

We extend this core DH model by wrapping it within a DHExp(elt,x) function that handles multiple good groups, bad groups, and bad elements (public keys) as follows:

```
1: DHExp(goodDHElt(goodDHGroup(id),x),y) =
     goodDHElt(goodDHGroup(id),E(x,y))
2: DHExp(goodDHElt(badDHGroup,x),y) = badDHElt(badDHGroup)
3: DHExp(badDHElt(gr),y) = badDHElt(gr).
```

The equation at line 1 handles the case where good groups and elements are used. In this case, the good group has an identifier id, and exponentiation in this group behaves like exponentiation over the core group G. The equations at lines 2 and 3 state that, whenever DHExp is computed for a bad group or bad element, a constant bad element for that group is obtained. The adversary knows the term badDHGroup and can always apply the badDHElt(gr) function to obtain bad elements. Hence, our model over-approximates small subgroup confinement, in that the small subgroup has always size 1, and hence the attacker can guess the computed subgroup value with probability 1.

Overall process structure: Given a two-party authentication protocol, we model one process per role, initiator() and responder() respectively. If one of the role needs to authenticate itself, the corresponding process takes a credential (and its secret) as an input parameter. A top level process sets up credentials and runs an unlimited number of instances of each role. For example, the top-level process for a key-exchange protocol where the responder authenticates (using a public key) to an anonymous initiator is written as:

```
process
  (* Responder credential generation *)
  new rsec:privkey; let rpub = pk(rsec) in out(net,rpub);
  (!initiator() | !responder(rpub,rsec))
```

When a process successfully ends a protocol instance, it stores the local identifier l, the authenticated credentials c_i, c_r, the instance parameters $params$ and the secret sk into a table, which acts as a session database. Initiators and responders use disjoint tables, named idb and rdb respectively.

For protocols that allow re-keying, session renegotiation or resumption, the initiator process has the following structure:

```
let initiator() =
  ... (* Model of initial key−exchange *)
  insert idb(l,ci,cr,params,sk)
| get idb(l,ci,cr,params,sk);
  ... (* Model of subsequent key−exchange *)
  insert idb(l',ci',cr',params',sk')
| ... (* Model of other subsequent key−exchange *)
```

That is, a process non-deterministically either runs the standard (initial) key exchange, or picks a session from the database and starts some subsequent key exchange method like re-keying or resumption. Responder processes have the same pattern.

In our model, a principal process accepts any credential from the other principal, as long as proof of possession of its associated secret can be provided. Hence, a session can be successfully completed either with an honest principal, or with the attacker who is using a compromised credential.

Honest principals only use honestly generated credentials and associated secrets; the attacker can generate any number of compromised credentials and use them in protocol instances. Hence, our model captures static credential compromise, but does not fully handle dynamic credential or session secret compromise, where some honest credentials or session secret are later leaked to the attacker, or where some compromised secrets are used by honest principals. Nevertheless, we can handle specific dynamic compromise scenarios by adapting the model of honest principals to intentionally leak credentials or session secrets after a certain step of a protocol instance.

We define several security properties as ProVerif queries and verify them against this attacker model, as we below.

B. Channel Synchronization

Channel synchronization over a channel binding parameter cb occurs when the following proposition is violated:

> Whenever an initiator and responder each complete a protocol instance with the same channel binding cb, all other parameters $(params, sk)$ at these two instances must be the same.

We encode such proposition in ProVerif by defining an auxiliary oracle() process, that tries to get from both the initiator and responder tables an entry having the same channel binding parameter cb, but different keys or credentials. If this succeeds, the oracle() process emits an event(Session_sync()). The query event(Session_sync()) checks for the reachability of this event; hence, if ProVerif can prove that event(Session_sync()) is unreachable, it means there is no channel synchronization attack for cb on the analyzed protocol.

1) TLS Initial Handshake: We begin by modeling TLS-RSA and using the master secret ms as a channel binding. As described in [16], synchronizing the master secret ms on TLS-RSA is not complicated: since $ms = kdf(pms, n_c, n_s)$, it is enough to synchronize the values used for its computation in order to mount the attack. ProVerif is able to find an attack where the attacker poses as a malicious responder to the honest initiator and as a malicious initiator to an honest responder. The honest participants end up with the same master secret even though their session parameters do not match: they have different server credentials. Adding further elements to the channel binding such as the TLS session id does not help, but using the session hash as channel binding prevents the attack.

We also model TLS-DHE and ProVerif finds a master secret synchronization attack by relying on bad groups (as in [16]). If both client and server check that good DH groups and keys are being used, ProVerif cannot find an attack.

2) SSH Key Exchange and Re-Keying: By comparison, we analyze encryption key synchronization attacks for the SSH key exchange protocol by using the session key as a channel binding. ProVerif can prove that the event(Session_sync()) is unreachable even in the presence of bad DH groups and keys, both for the first key exchange and for re-keying. Indeed, SSH encryption keys are computed as $sk = kdf(K, H, sid)$, where K is the potentially bad DH shared secret, but crucially H is the exchange hash capturing unique information about the ongoing instance, notably including local unique identifiers and the value of the credential being authenticated.

C. Agreement at Initiator

Agreement for a single protocol (Definition 1) is modeled as an authentication query as follows:

```
query inj−event InitiatorEnd(pk(s),params,sk) =>
        inj−event ResponderBegin(pk(s),params,sk) || attacker(s)
```

where s is the secret associated with credential pk(s), and params and sk are the instance parameters and shared secret respectively. That is, if the initiator completes the protocol, either the responder has completed with the same parameters and keys, or the responder's credential is compromised.

1) TLS with Renegotiation and Resumption: ProVerif can prove agreement at initiator for all the three TLS modes, namely initial handshakes, renegotiation and resumption, even when session keys are dynamically compromised. We stress that this kind of agreement holds even if we do not model the renegotiation information (RI) extension [12], or any other channel binding mechanism, since they only apply to compound authentication, not to single protocol agreement.

2) SSH with Re-keying: According to our definition, we try to prove agreement on the shared secret sk and the parameters H, K, sid, pk_S. We model the SSH key exchange protocol, including re-keying. At the end of each key exchange we can only prove agreement on K, H and pk_S; but, crucially, right after the key exchange protocol has ended, agreement on sid and sk fails, and ProVerif hints at the following attack.

First, the attacker connects to a honest server b, obtaining $sk, K, H, sid = H$. Second, an honest client tries to connect to b; the attacker tunnels this key exchange through its current connection. At the end of the key exchange, client and server agree on the most recent exchange hash H' and DH shared secret K', but they have different session ids and encryption keys, namely $sid' = H', k' = kdf(K', H', sid')$ on the client and $k'' = kdf(K', H', sid)$ on the server.

As noted in [40, §6.3], the SSH key exchange protocol prescribes explicit confirmation only for K and H, via server digital signature. Confirmation of the encryption keys, and hence of sid, is implicitly done when receiving the first encrypted application message from the other party, in case decryption succeeds. Accordingly, if we add an explicit key confirmation message encrypted under the new keys at the end of the SSH key exchange, we can successfully prove agreement on encryption keys and all parameters. In other words, SSH re-keying does guarantee agreement, but only after the keys have been confirmed by a pair of additional (application) messages have been exchanged.

D. Agreement at Responder and Compound Authentication

Agreement at responder is defined symmetrically to agreement at initiator, as:

```
query inj−event ResponderEnd(pk(s),params,sk) =>
      inj−event InitiatorBegin(pk(s),params,sk) || attacker(s).
```

Following definition 2, we may want to write compound authentication as an authentication query over n protocols:

```
query inj−event Compound_ResponderEnd(pk(s),
          params_1,sk_1, ..., params_n,sk_n) =>
      inj−event Compound_InitiatorBegin(pk(s),
          params_1,sk_1, ..., params_n,sk_n) || attacker(s).
```

However, the number n of protocol instances is unbound, and hence this query cannot be practically written. We overcome this problem by defining a function log(params,pl) that takes the current instance parameters params and a previous log pl, and returns a new log that is the concatenation of the current parameters and the previous log. A constant emptyLog is defined to bootstrap. Each initiator and receiver session table is updated to additionally store the log; the first key exchange stores log(params,emptyLog) into its table, while any subsequent key exchange picks a previous log pl from the table, and at the end of a successful run stored the new log(params',pl).

Using log, we write compound authentication at the responder as the following authentication query:

```
query inj−event Compound_ResponderEnd(pk(s),p,sk,log) =>
   inj−event Compound_InitiatorBegin(pk(s),p,sk,log) || attacker(s).
```

The log is never used by the protocol, it only appears in the tables and in the security events. In the protocol, the channel binding cb must guarantee agreement on the log and hence on all prior protocol instances.

We note a difference between this query and the more general Definition 2, in that our query only proves agreement on previous sessions. We believe that agreement on subsequent sessions can be obtained as a corollary, since a honest participant will not authenticate attacker-provided parameters in successive protocol instances.

1) TLS-RSA+SCRAM with Renegotiation and Resumption: We model agreement at the responder by letting the user authenticate to the server via the password-based SCRAM protocol on top of a TLS connection. User authentication can be performed after any TLS handshake (initial, resumed or renegotiated) has taken place.

We model dynamic key compromise for all TLS sessions, by leaking the session keys to the attacker at the completion of each session. This means that, in practice, all SCRAM messages can be tampered with by the attacker, which accounts for a strong attacker model. Furthermore, we let the user use the same password with the attacker, under the condition that the attacker salt differs from the salt of the honest peers.

ProVerif can prove agreement at the responder at the end of each SCRAM instance, which shows that, in isolation, SCRAM provides user authentication, even when the same password is used with the attacker.

Compound authentication of TLS-RSA+SCRAM relies on the use of the `tls-unique` channel binding in SCRAM. However, we find that this goal fails when TLS session resumption is enabled. ProVerif finds an attack in accordance with the results of [16]: at the end of the second (resumption) handshake, the channel bindings for client and server are synchronized, hence the attacker can forward the SCRAM messages between server and client, with the result of authenticating as the user u to the server.

We patch the TLS model to implement the extended master secret derivation of Section IV-A. For this model, ProVerif is able to prove compound authentication. Indeed, the addition of the session hash into the master secret fixes `tls-unique` and makes it an adequate channel binding for SCRAM over TLS, thwarting the channel synchronization attack.

2) SSH-USERAUTH with Re-keying: We model the SSH user authentication protocol on top of the SSH key exchange protocol. In our model, the key exchange protocol can be run several times (for re-keying) but the user authentication protocol is run only once after the first key exchange: this is in conformance to the standard, which prescribes that any further user authentication request after the first successful one should be ignored. After each key exchange, the attacker may compromise the session and obtain its keys and exchange hash.

For this protocol, we are interested in two kinds of compound authentication: the first is about successive instances of the key exchange protocol itself; the second is between the key exchange protocol and the user authentication one.

As anticipated by the attack depicted in figure 7, SSH does not satisfy compound authentication for arbitrary sequences of key exchange if the first session keys and exchange hash are compromised. In this setting, ProVerif finds the attacks and reports the authentication property failure.

The cumulative hash we proposed in Section IV-B binds all parameters of the current protocol instance to the parameters of previous instances. In proposing this fix, we claim that: (i) keeping sid becomes unnecessary, as the cumulative hash provides a stronger binding; (ii) the extra key confirmation messages become unnecessary, since now all agreement information is contained within the cumulative hash, which is explicitly agreed upon. We implement our fix in the SSH ProVerif model, and obtain a proof of key exchange compound authentication, which formally validates our proposed fix.

With respect to compound authentication between key exchange and user authentication, ProVerif can prove that this property holds, even when the cumulative hash is not used. Restricting user authentication to happen after the first key exchange avoids the key exchange channel binding problem, and hence thwarts the attack.

TABLE I. VERIFICATION SUMMARY

Model (with session secret compromise)	Session Sync	Initiator agr.	Responder agr.	Compound auth.	Verification time
SSH-USERAUTH+Rekey	None	Yes[1]	Yes	No / Yes[2]	1.9s
SSH-USERAUTH+Rekey (cumulative hash)	None	Yes[3]	Yes	Yes / Yes[2]	0.6s
TLS-RSA+Renego+Resume	sid, ms, cr, sr	Yes	N/A	N/A	1.3s
TLS-RSA+Renego+Resume+SCRAM	sid, ms, cr, sr	Yes	Yes	No[4]	15.6s
TLS-RSA+Renego+Resume+SCRAM (session hash)	None	Yes	Yes	Yes	21.6s

[1]After explicit key confirmation [2]Key exchange / User authentication [3]With no need for explicit key confirmation [4]Triple handshake; SCRAM impersonation

E. Summary of Analyzed Models and Properties

Table I summarizes the 20 protocol variants and authentication properties examples that have been discussed and analyzed with ProVerif in this section. All reported models take into account static credential compromise and dynamic session secret compromise, by explicitly leaking the session secret to the attacker at the end of a successful protocol instance. The table reports, for each protocol model, a synthetic comment on the analyzed security properties and, in the last column, the ProVerif verification time on a 2.7 GHz Intel Core i7 machine with 8GB of RAM running a Unix operating system. All our ProVerif scripts are available online.[2]

In the first row, we find that the SSH key exchange with user authentication is not vulnerable to channel synchronization when known DH groups are used and public values are validated. The protocol has no initiator or responder agreement flaws, albeit we observe that an extra key confirmation step is necessary to get initiator agreement on the session secret. Moreover, while compound authentication of key exchange and user authentication is sound, ProVerif finds an attack on sequences of key exchanges, where an attacker compromising the first session secret can cause a mismatch between the key exchange histories at the user and host.

The second row shows that using the cumulative hash as a channel binding fixes compound authentication for sequences of key exchanges, and furthermore makes the extra key confirmation step superfluous.

TLS-RSA with session resumption and renegotiation is summarized at the third row. As discussed in [16], the protocol is vulnerable to channel synchronization on many relevant parameters, notably the shared secret. On this model we also analyze basic agreement at the initiator, which can be showed to hold even without the presence of the mandatory RI extension, as this agreement is a property local to the current handshake instance.

We move our analysis to the combination TLS-RSA+SCRAM (fourth row), where we find the same TLS-level issues such as channel synchronization, and where the analysis of compound authentication properties finds two instances of a family of attacks. The first instance is a triple handshake attack; the second instance involves two TLS handshakes followed by a run of the SCRAM protocol.

We formally evaluate the validity of the proposed session hash in the fifth row, where we observe that both channel synchronization and compound authentication flaws are fixed.

We emphasize that these results only hold for our abstract models and within the limits of our formal threat model. We do not capture, for example, dictionary attacks on SCRAM passwords, or padding oracle attacks on the TLS record protocol. Even when ProVerif finds no attacks, there may well be realistic attacks on the protocol outside our model.

[2]http://prosecco.inria.fr/projects/channelbindings

VI. RELATED WORK

Man-in-the-middle attacks that break authentication have been documented both against well-known academic security protocols such as Needham-Schroeder [31] and against widely used ones such as PEAP [1] and TLS renegotiation [3], [4], [16]. The work in this paper is closely related to and inspired by the triple handshake attacks on TLS [16]. However, most of these attacks were found by hand, whereas we aim to find them systematically by formal analysis.

Several works have performed rigorous analysis of widely used key exchange protocols, both in the symbolic setting (e.g. [41], [42] for TLS, [43], [38], [44] for SSH, [45] for IKEv2) and in the computational setting (e.g. [46], [47], [48], [49] for TLS, [50], [51] for SSH). We observe that none of the formal analysis works above takes into account the problem of compound authentication, neither by means of what channel bindings to expose to outer protocols, nor by means of the interaction between several instances and modes of the same protocol. Furthermore, with the exception of [46], due to the complexity of the analyzed protocols, no previous work performs a global analysis encompassing at the same time features such as re-keying, renegotiation and resumption, often necessary to mount the man-in-the-middle attacks discussed in this paper. In our work, we complement previous analysis results by providing a formal model for compound authentication that can be automatically verified in the symbolic setting.

A separate line of work concerns safe protocol composition [52], [53], [54], for instance, for protocol instances that are nested within each other or run in parallel. These works aim at ensuring that the individual security of each protocol is preserved even when it runs within or alongside other protocols. In contrast, these works do not consider the problem of obtaining stronger compound authentication properties by the composition of the protocols. We present the first formal protocol models and systematic analysis for such properties.

VII. CONCLUSIONS

Compound authentication protocols present a challenging but rewarding target for formal analysis. While it may be possible to analyze specific configurations of these protocols by hand, the complex multi-protocol attacks described in this paper show that automation is direly needed both to find new attacks and to evaluate their countermeasures against strong attackers. We have made a first attempt towards the automated analysis of such protocols. Our 20 models of various combinations of TLS, SSH, and SASL are detailed and precise and we are able to find both known and new man-in-the-middle attacks on various channel binding proposals, as well as evaluate the new proposals presented in this paper. Our models are far from complete, but they already indicate that this is a fruitful direction for future research.

Acknowledgments: We would like to thank Bruno Blanchet for his support and advice on the use of the ProVerif tool.

References

[1] N. Asokan, V. Niemi, and K. Nyberg, "Man-in-the-middle in tunnelled authentication protocols," in *Security Protocols*, 2005.

[2] J. Puthenkulam, V. Lortz, A. Palekar, D. Simon, and B. Aboba, "The compound authentication binding problem," IETF Draft v04, 2003.

[3] M. Ray and S. Dispensa, "Authentication gap in TLS renegotiation," 2009.

[4] M. Rex, "Mitm attack on delayed TLS-client auth through renegotiation," 2009, http://ietf.org/mail-archive/web/tls/current/msg03928.html.

[5] R. Oppliger, R. Hauser, and D. Basin, "SSL/TLS session-aware user authentication - or how to effectively thwart the man-in-the-middle," *Comput. Commun.*, vol. 29, no. 12, pp. 2238–2246, 2006.

[6] M. Dietz, A. Czeskis, D. Balfanz, and D. S. Wallach, "Origin-bound certificates: a fresh approach to strong client authentication for the web," in *USENIX Security*, 2012.

[7] B. Aboba, L. Blunk, J. Vollbrecht, J. Carlson, and H. Levkowetz, "Extensible Authentication Protocol (EAP)," IETF RFC 3748, 2004.

[8] A. Palekar, D. Simon, J. Salowey, H. Zhou, G. Zorn, and S. Josefsson, "Protected EAP protocol (PEAP) version 2," IETF Draft v10, 2004.

[9] P. Funk and S. Blake-Wilson, "EAP-TTLSv0: Extensible Authentication Protocol Tunneled Transport Layer Security Authenticated Protocol version 0," IETF RFC 5281, 2008.

[10] N. Williams, "On the use of channel bindings to secure channels," IETF RFC 5056, 2007.

[11] J. Altman, N. Williams, and L. Zhu, "Channel bindings for TLS," IETF RFC 5929, 2010.

[12] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov, "Transport Layer Security (TLS) Renegotiation Indication Extension," RFC 5746, 2010.

[13] Microsoft, "Extended protection for authentication in integrated Windows authentication," 2009, http://support.microsoft.com/kb/968389.

[14] D. Balfanz and R. Hamilton, "TLS Channel IDs," IETF Draft v01, 2013.

[15] F. Giesen, F. Kohlar, and D. Stebila, "On the security of TLS renegotiation," in *ACM CCS*, 2013.

[16] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P.-Y. Strub, "Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS," in *S&P*, 2014.

[17] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," *SIGPLAN Not.*, vol. 36, pp. 104–115, January 2001.

[18] B. Blanchet, "An efficient cryptographic protocol verifier based on Prolog rules," in *CSF*, 2001, pp. 82–96.

[19] K. Bhargavan, A. Delignat-Lavaud, A. Pironti, A. Langley, and M. Ray, "TLS session hash and extended master secret," IETF Draft, 2014.

[20] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on IT*, vol. IT–29, no. 2, pp. 198–208, 1983.

[21] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *ACM CCS*, 2012.

[22] D. Akhawe, B. Amann, M. Vallentin, and R. Sommer, "Here's my cert, so trust me, maybe?: understanding TLS errors on the Web," in *WWW*, 2013, pp. 59–70.

[23] A. Cassola, W. Robertson, E. Kirda, and G. Noubir, "A practical, targeted, and stealthy attack against WPA enterprise authentication," in *NDSS*, 2013.

[24] C. Soghoian and S. Stamm, "Certified lies: Detecting and defeating government interception attacks against SSL," in *FC*, 2012.

[25] A. Menon-Sen, N. Williams, A. Melnikov, and C. Newman, "Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms," IETF RFC 5802, 2010.

[26] T. Hardjono and N. Klingenstein, "SAML v2.0 channel binding extensions version 1.0," OASIS Committee Specification, 2013.

[27] Y. Sheffer and H. Tschofenig, "IKEv2 session resumption," IETF RFC 5723, 2010.

[28] J. Schonwalder, G. Chulkov, E. Asgarov, and M. Cretu, "Session resumption for the Secure Shell protocol," in *Integrated Network Management*, 2009, pp. 157–163.

[29] Z. Cao, B. He, Y. Shi, Q. Wu, and G. Zorn, "EAP extensions for the EAP re-authentication protocol (ERP)," IETF RFC 6696, 2012.

[30] K. Welter, "Reauthentication extension for IKEv2," IETF Draft, 2011.

[31] G. Lowe, "Breaking and fixing the Needham-Schroeder public-key protocol using FDR," in *TACAS*, 1996, pp. 147–166.

[32] R. Anderson and S. Vaudenay, "Minding your p's and q's," in *ASIACRYPT*, 1996.

[33] E. Barker, D. Johnson, and M. Smid, *NIST Special Publication 800-56A Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised)*, 2007.

[34] Y. Sheffer and S. Fluhrer, "Additional Diffie-Hellman tests for IKEv2," IETF RFC 6989, 2013.

[35] M. Lepinski and S. Kent, "Additional Diffie-Hellman groups for use with IETF standards," IETF RFC 5114, 2008.

[36] N. Williams, "Unique channel bindings for IPsec using IKEv2," IETF Draft, 2008.

[37] D. J. Bernstein, "Curve25519: new Diffie-Hellman speed records," in *Public Key Crypto*. Springer, 2006, pp. 207–228.

[38] R. Kusters and T. Truderung, "Using ProVerif to analyze protocols with Diffie-Hellman exponentiation," in *CSF*, 2009, pp. 157–171.

[39] B. Schmidt, S. Meier, C. Cremers, and D. Basin, "Automated analysis of Diffie-Hellman protocols and advanced security properties," in *CSF*, 2012, pp. 78–94.

[40] R. Gelashvili, "Attacks on re-keying and renegotiation in key exchange protocols," Master's thesis, ETH Zurich, 2012.

[41] K. Bhargavan, C. Fournet, R. Corin, and E. Zălinescu, "Verified cryptographic implementations for TLS," *TISSEC*, vol. 15, no. 1, p. 3, 2012.

[42] M. Avalle, A. Pironti, D. Pozza, and R. Sisto, "JavaSPI: A framework for security protocol implementation," *JSSE*, vol. 2, p. 34–48, 2011.

[43] E. Poll and A. Schubert, "Verifying an implementation of SSH," in *WITS*, 2007, pp. 164–177.

[44] A. Pironti, D. Pozza, and R. Sisto, "Formally based semi-automatic implementation of an open security protocol," *Journal of Systems and Software*, vol. 85, no. 4, pp. 835–849, 2012.

[45] C. Cremers, "Key exchange in IPsec revisited: Formal analysis of IKEv1 and IKEv2," in *ESORICS*, 2011, pp. 315–334.

[46] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub, "Implementing TLS with verified cryptographic security," in *S&P*, 2013.

[47] H. Krawczyk, K. G. Paterson, and H. Wee, "On the Security of the TLS Protocol: A Systematic Analysis," in *CRYPTO*, 2013.

[48] P. Morrissey, N. P. Smart, and B. Warinschi, "A modular security analysis of the TLS handshake protocol," in *ASIACRYPT*, 2008, pp. 55–73.

[49] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Z. Béguelin, "Proving the TLS handshake secure (as it is)," in *CRYPTO*, 2014, pp. 235–255.

[50] S. C. Williams, "Analysis of the SSH key exchange protocol," in *Cryptography and Coding*, 2011, pp. 356–374.

[51] D. Cadé and B. Blanchet, "From computationally-proved protocol specifications to implementations," in *ARES*, 2012, pp. 65–74.

[52] T. Groß and S. Modersheim, "Vertical protocol composition," in *CSF*, 2011, pp. 235–250.

[53] S. Gajek, M. Manulis, O. Pereira, A.-R. Sadeghi, and J. Schwenk, "Universally composable security analysis of TLS," in *Provable Security*, 2008, pp. 313–327.

[54] C. He, M. Sundararajan, A. Datta, A. Derek, and J. C. Mitchell, "A modular correctness proof of IEEE 802.11i and TLS," in *ACM CCS*, 2005, pp. 2–15.