Attack, solution and verification for shared authorisation data in TCG TPM

Liqun Chen and Mark Ryan

HP Labs, UK, and
University of Birmingham, UK

Abstract. The Trusted Platform Module (TPM) is a hardware chip designed to enable computers achieve greater security. Proof of possession of authorisation values known as *authdata* is required by user processes in order to use TPM keys. If a group of users are to be authorised to use a key, then the authdata for the key may be shared among them. We show that sharing authdata between users allows a TPM impersonation attack, which enables an attacker to completely usurp the secure storage of the TPM. The TPM has a notion of encrypted transport session, but it does not fully solve the problem we identify.
We propose a new authorisation protocol for the TPM, which we call Session Key Authorisation Protocol (SKAP). It generalises and replaces the existing authorisation protocols (OIAP and OSAP). It allows authdata to be shared without the possibility of the impersonation attack, and it solves some other problems associated with OIAP and OSAP. We analyse the old and the new protocols using ProVerif. Authentication and secrecy properties (which fail for the old protocols) are proved to hold of SKAP.

1 Introduction

The Trusted Platform Module (TPM) specification is an industry standard [**?**] and an ISO/IEC standard [**?**] coordinated by the Trusted Computing Group (TCG), for providing trusted computing concepts in commodity hardware. TPMs are chips that aim to enable computers to achieve greater levels of security than is possible in software alone. There are 100 million TPMs currently in existence, mostly in high-end laptops. Application software such as Microsoft's BitLocker and HP's HP ProtectTools use the TPM in order to guarantee security properties.

The TPM stores cryptographic keys and other sensitive information in shielded locations. Keys are organised in a tree hierarchy, with the *Storage Root Key* (SRK) at its root. Each key has associated with it some authorisation data, known as authdata. It may be thought of as a password to use the key. Processes running on the host platform or on other computers can use the TPM keys in certain controlled ways. To use a key, a user process has to prove knowledge of the relevant authdata. This is done by accompanying the command with an HMAC (a hash-function-based message authentication code, as specified in [**?**]),

keyed on the authdata or on a shared secret derived from the authdata. When a new key is created in the tree hierarchy, its authdata is chosen by the user process, and sent encrypted to the TPM. The encryption is done with a key that is derived from the parent key authdata. The TPM stores the new key's authdata along with the new key. Creating a new key involves using the parent key, and therefore an HMAC proving knowledge of the parent key's authdata has to be sent.

If a group of users are to be authorised to use a key, then the authdata for the key may be shared among them. In particular, the authdata for SRK (written srkAuth) is often assumed to be a widely known value, in order to permit anyone to create child keys of SRK. This is analogous to allowing several people to share a password to use a resource, such as a database.

We show that sharing authdata between users has some significant undesirable consequences. For example, an attacker that knows srkAuth can *fake all the storage capabilities* of the TPM, including key creation, sealing, unsealing and unbinding. Shared authdata completely breaks the security of the TPM storage functions. Some commentators to whom we have explained our attack have suggested the TPM's encrypted transport sessions as a way of mitigating the attack. We show that they are not able to do that satisfactorily (section 2.4).

We solve this problem by proposing a new authorisation protocol for the TPM, which we call Session Key Authorisation Protocol (SKAP). It generalises and replaces the existing authorisation protocols (OIAP and OSAP). In contrast with them, it does not allow an attacker that knows authdata to fake a response by the TPM. SKAP also fixes some other problems associated with OIAP and OSAP. To demonstrate its security, we analyse the old and the new protocols using the protocol analyser ProVerif [**?**,**?**], and prove authentication and secrecy properties of SKAP.

Related work Other attacks of a less significant nature have been found against the TPM. The TPM protocols expose weak authdata secrets to offline dictionary attacks [**?**]. To fix this, we proposed to modify the TPM protocols by using SPEKE (Simple Password Exponential Key Exchange [**?**]). However, the modifications proposed in [**?**] do not solve the problem of shared authdata.

An attacker can in some circumstances illegitimately obtain a certificate on a TPM key of his choice [**?**]. Also, an attacker can intercept a message, aiming to cause the legitimate user to issue another one, and then cause both to be received, resulting in the message being processed twice [**?**]. Some verification of certain aspects of the TPM is done in [**?**]. Also in [**?**], an attack on the delegation model of the TPM is described; however, experiments with real TPMs have shown that the attack is not possible [**?**].

Paper overview Section 2 describes the current authorisation protocols for the TPM, and in Sections 2.2 and 2.3 we demonstrate our attack. In Section 2.4 we explain why the TPM's encrypted transport sessions don't solve the problems. Section 3 describes our proposed protocol, SKAP, that replaces OIAP and OSAP.

In section 4, we use ProVerif to demonstrate the security of SKAP compared with OIAP and OSAP. Conclusions are in Section 5.

2 TPM authorisation

A TPM command that makes use of TPM keys requires the process issuing the command to be authorised. A process demonstrates its authorisation by proving knowledge of the relevant authdata. This is done by accompanying a TPM command with such an HMAC of the command parameters, keyed on the authdata or on a shared secret derived from the authdata. We note the result of the HMAC by $\mathsf{hmac}_{ad}(msg)$, where ad is the authdata, and msg is a concatenation of selected message parameters.

The response from the TPM to an authorised command is also accompanied by an HMAC of the response parameters, again keyed on the authdata or the shared secret. This is intended to authenticate the response to the calling process.

The TPM provides two kinds of authorisation sessions, called *object independent authorisation protocol* (OIAP) and *object specific authorisation protocol* (OSAP). OIAP allows multiple keys to be used within the same session, but it doesn't allow commands that introduce new authdata, and it doesn't allow authdata for an object to be cached for use over several commands. An OSAP session is restricted to a single object, but it does allow new authdata to be introduced and it creates a session secret to securely cache authorisation over several commands. If a command within an OSAP session introduces new authdata, then the OSAP session is terminated by the TPM (because the shared secret is contaminated by its use in XOR encryption).

In order to prevent replay attacks, each HMAC includes two nonces, respectively from the user process and TPM, as part of msg. The nonces created by the calling process are called "odd", denoted n_o, and the nonces created by the TPM are called "even", denoted by n_e. The nonces are sent in the clear, and also included in the msg part of the HMAC. Both the process and TPM use a fresh nonce in each HMAC computation, and they verify the incoming HMACs to check integrity and authorisation. For example, the process sends the first nonce odd n_{o1} to the TPM and receives the first nonce even n_{e1} along with $mac_1 = \mathsf{hmac}_{ad}(n_{o1}, n_{e1}, ...)$, and then sends $mac_2 = \mathsf{hmac}_{ad}(n_{e1}, n_{o2}, ...)$ with n_{o2} and receives n_{e2} along with $mac_3 = \mathsf{hmac}_{ad}(n_{o2}, n_{e2}, ...)$, and so on. This is sometimes called a rolling nonce protocol.

2.1 Authorisation example

In this subsection, we will take a look at an authorisation example, in which a user process first asks the TPM create a new key as part of the storage key tree, then loads this key into the TPM internal memory, and finally uses this key to encrypt some arbitrary data. These three functions are demonstrated in Figure 1 in three separated sessions.

Session 1 shows the exchange of messages between the user process and the TPM when a child key of another loaded key (called the parent key) is created using the TPM command TPM_CreateWrapKey. The TPM returns a blob, consisting of the newly created key and some other data, encrypted with the parent key. The user and TPM achieve this function by performing the following steps:

1. First, the user process sets up a OSAP session based on the currently loaded parent key. The parent key handle is pkh, and its authdata is $\mathsf{ad}(pkh)$. The TPM_OSAP command includes pkh and the nonce n_o^{osap}.
2. Upon receipt of the TPM_OSAP command, the TPM assigns a new session authorisation handle ah, generates two nonces n_e and n_e^{osap}, and sends these items back as the response.
3. The user process and the TPM each calculate the shared secret S derived from $\mathsf{ad}(pkh)$, and the two nonces for OSAP by using the HMAC algorithm.
4. Then, the user process calls TPM_CreateWrapKey, providing arguments including authdata $newauth$ for the key being created, some other parameters about the key, and the HMAC keyed on S demonstrating knowledge of SRK authdata. To protect the new authdata, it is XOR-encrypted with a key derived from $\mathsf{ad}(pkh)$ and n_e using the hash-function SHA1.
5. After receiving this command, the TPM checks the HMAC and creates the new key. The TPM returns a blob, $keyblob$, consisting of the public key and an encrypted package containing the private key and the new authdata. The returned message is authenticated by accompanying it with an HMAC with the two nonces keyed on S.
6. Because the shared secret S has been used as a basis for an authdata encryption key, the OSAP session is terminated by the TPM. Later commands will have to start a new session.

In order to be used, the newly created key must be loaded into the TPM. For this, an OIAP session may be used. Session 2 shows the messages exchanged between the user process and the TPM during the creation of the OIAP session and the TPM_LoadKey2 command. The following steps are performed:

1. The user process sends the TPM_OIAP command to the TPM.
2. The TPM assigns the session authorisation handle ah' and sends it back along with the newly created nonce n_e''. ,
3. The process calls TPM_LoadKey2, providing arguments including the parent key handle pkh and $keyblob$. The authorisation of this command is achieved using the authdata of the parent key, $\mathsf{ad}(pkh)$.
4. The TPM checks the HMAC, and if the check passes, decrypts $keyblob$ and loads the key into its internal memory. The TPM finally creates a key handle for the loaded key kh and a nonce n_e''' and sends them back together with an HMAC keyed on the authdata of the parent key $\mathsf{ad}(pkh)$.

After the key is loaded, it can be used to encrypt data using TPM_Seal. As well as encrypting the data, TPM_Seal binds the encrypted package to particular Platform Configuration Registers (PCRs) specified in the TPM_Seal command.

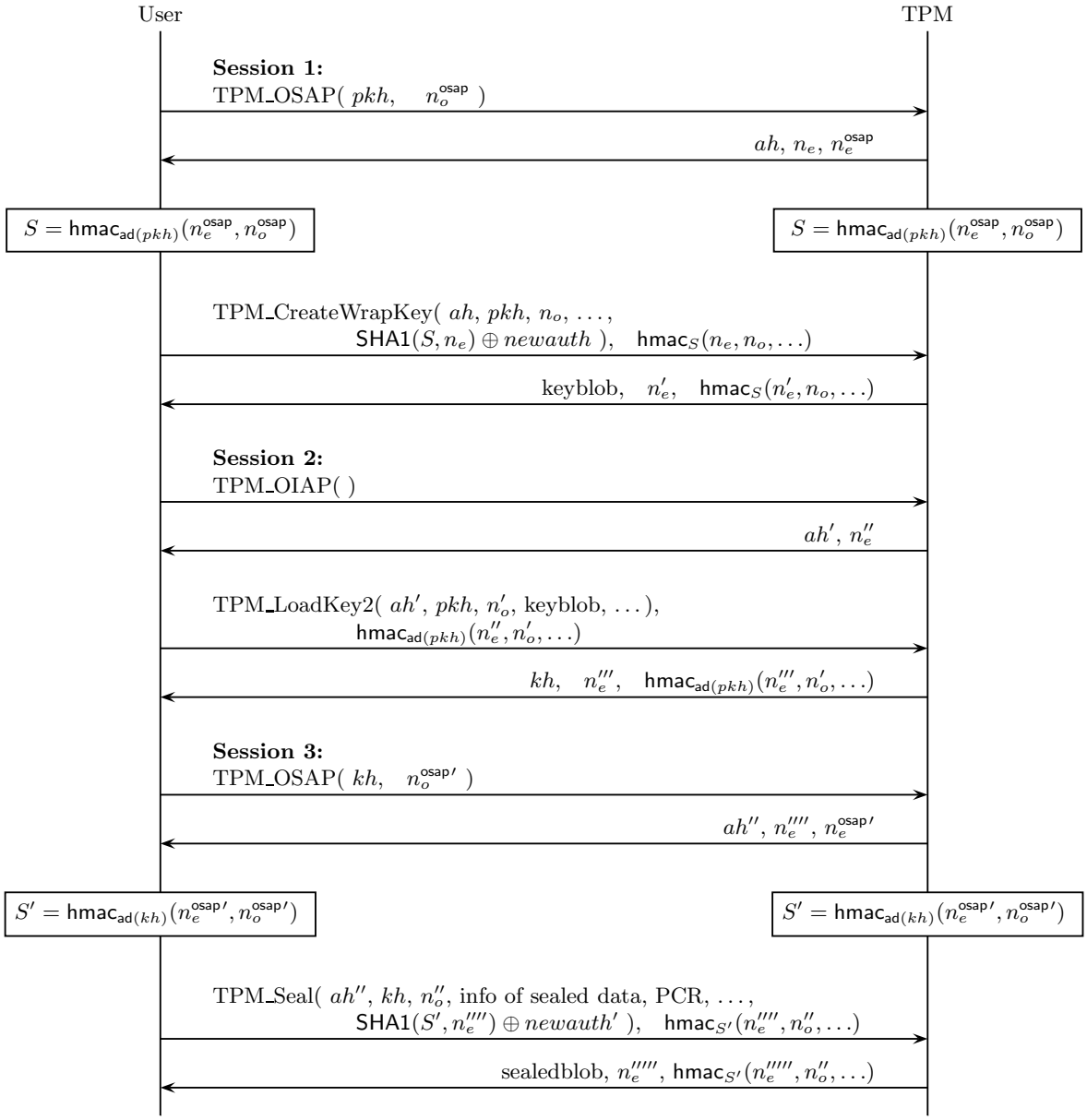

Fig. 1. Session 1: Creating a key on the TPM. TPM_OSAP creates an OSAP session and the shared secret S by both parties. TPM_CreateWrapKey requests the TPM to create a key. The command and the response are authenticated by the shared secret S. Session 2: Loading a key on the TPM. TPM_OIAP creates an OIAP session for the TPM_LoadKey2 command. Session 3: Using the key to seal data. TPM_OSAP creates an OSAP session and its corresponding shared secret S' for the TPM_Seal command. The seal command and the response are authenticated by S'.

The TPM will later unseal the data only if the platform is in a configuration matching those PCRs. TPM_Seal requires a new OSAP session based on the newly created key. The details are shown in Session 3, where the user process and TPM perform the following steps:

1. The first three steps are identical to Session 1, except they use the key handle kh and authdata $\mathsf{ad}(kh)$ belonging to the newly loaded key, instead of pkh and $\mathsf{ad}(pkh)$.
2. After setting up the OSAP session, the user process calls TPM_Seal, providing arguments including the information of the sealed data, a PCR and the new authdata for the corresponding unseal process. The new authdata is again XOR-encrypted with a key derived from the encryption key authdata. The message is authenticated by accompanying it with an HMAC keyed on the secret S'.
3. The TPM responds the command with a sealed blob *sealedblob*, which consists of an encrypted package containing the sealed data, the PCR value and the new authdata. Again the returned message is authenticated by accompanying it with an HMAC keyed on the secret S'.

2.2 The problem of shared authdata

If authdata is a secret shared only between the calling process and the TPM, then the HMACs serve to authorise the command and to authenticate the TPM response. However, as mentioned earlier, authdata may be shared between several users, in order to allow each of them to use the resource that the authdata protects. In particular, the authdata of SRK is often assumed to be a well-known value. E.g., in *Design Principles* of the TPM specification [**?**,**?**], sections 14.5, 14.6 refer to the possibility that SRK authdata is a well-known value, and sections 30.2, 30.8 refer to other authorisation data being well-known values.

The usage model is that a platform has a single TPM, and the TPM has a single SRK, which plays the role of the root of a trusted key hierarchy tree. If the platform has multiple users, each of them can build their own branches of the tree on the top of the same root. In order to let multiple users access SRK, the authdata of SRK is made available to all of them. The goal is that although these users share the same SRK and its authdata, they are only able to access their own key branches but not anyone else's. We will show how the idea of sharing SRK authdata fails to achieve the design principle of the protected storage functionality of the TPM.

Suppose one of these users who knows an authdata value is malicious and he can intercept a command from another user to the TPM (the TPM protocols involving encryption and HMACs are clearly designed on the assumption that such interception is possible). He can use knowledge of the authdata to decrypt any new authdata that the command is introducing; and he can fake the TPM response that is authenticated using the shared authdata.

It follows that an attacker that knows the authdata for SRK can fake the creation of child keys of SRK. Those keys are then keys made by the attacker

in software, and completely under his control. He can intercept requests to use those keys, and fake the response. Therefore, all keys intended to be descendants of SRK can be faked by the TPM. An attacker with knowledge of SRK authdata can completely usurp the storage functionality of the TPM, by creating all the keys in software under his own control, and faking all the responses by the TPM.

2.3 The attack in practice

We suppose that Alice is in possession of a laptop owned by her employer, that has an IT department which we call ITadmin. The TPM_TakeOwnership command has been performed by ITadmin when the laptop was first procured; thus, the TPM has created SRK and given its authdata to ITadmin. When Alice receives the laptop, she is also provided with SRK authdata so that she can use the storage functions of the TPM.

Alice now decides to create a key on the TPM with authdata of her own choosing, and wants to encrypt her data using that key. She invokes the commands of Figure 1 of Section 2.1. Unknown to her, ITadmin has configured the laptop so that commands intended to go to the TPM go instead to software under ITadmin's control. This software responds to all the commands that Alice sends. ITadmin's software creates the necessary nonces and fakes the response to TPM_OSAP. Next, it fakes the creation of the key and fakes all the responses to the user (again creating all the necessary nonces). In particular, in the case of TPM_CreateWrapKey, ITadmin's software

- is able to calculate the session secret S, since it is based on SRK authdata and other public values (namely, the OSAP nonces that are sent in the clear);
- is able to decrypt the new authdata, since it is XOR encrypted with a key based on SRK authdata and other public values (namely, the command nonces that are sent in the clear);
- is able to create an RSA key in software, according to the parameters specified in the command;
- is able to create the message returned to the user process. This involves encrypting the "secret" package with SRK, and creating the HMAC that "authenticates" the TPM.

Next, ITadmin's software fakes the response to TPM_LoadKey2 (using its knowledge of SRK authdata to create the necessary HMAC). Finally, it fakes the response to TPM_Seal (using its knowledge of the new key's authdata to create the necessary HMAC). Therefore, the ITadmin can successfully impersonate the TPM just because it knows the authdata of SRK.

The attack scenario given in this example, in which ITadmin is the attacker, is similar to that illustrating the TPM_CertifyKey attack in [**?**]. Many other scenarios are possible. For example, TPMs are now common in servers, and many interesting use cases involve remote clients accessing TPM functionality on a server (for instance, to achieve guarantees about the server behaviour). In that scenario, our attack means that the server is able to spoof all the responses

from the TPM. Another class of scenarios which illustrate this attack revolve around virtualisation; there too, independent virtual environments share a TPM and share knowledge of SRK authdata, allowing one such environment to spoof TPM replies to another.

2.4 Encrypted transport sessions

The OIAP and OSAP sessions are intended to provide message integrity, but not message confidentiality. The TPM has a notion of encrypted transport session [**?**,**?**] which is intended to provide message confidentiality. Encrypted transport sessions are initiated with the TPM_EstablishTransport command, which allows a session key to be established, using a public storage key of the TPM. Since the security of the session is anchored in a public key, and that public key can be certified, this does indeed defeat the TPM spoofing attack we have described above.

However, encrypted transport sessions are not an ideal solution to be used as an alternative of the OIAP and OSAP sessions for the purpose of providing robust TPM authorisation, because the encrypted transport sessions do not solve the problem of weak authdata, reported in [**?**]. In that paper, it is shown that the TPM protocols expose authdata to the possibility of offline guessing attacks. If authdata is based on a weak secret, then an attacker that tries to guess the value of the authdata is able to confirm his guess offline. Encrypted transport sessions do not resist against this attack because they do not encrypt the high-entropy values (the rolling nonces) that are used in the authorisation HMACs.

Therefore, changes to OIAP and OSAP are necessary, to avoid the attack of [**?**]. Unfortunately, the changes proposed in [**?**] do not solve the attack we have identified in this paper. The solution proposed in [**?**] is based on the SPEKE protocol, which relies on a secret being shared between the two participants, whereas shared authdata precisely invalidates that assumption.

Thus, there is no alternative to a thorough re-design of the authorisation protocols of the TPM.

3 A new TPM authorisation protocol

Our aim is to design an authorisation protocol that solves both the weak authdata problem of [**?**] and the shared authdata problem reported in this paper. Moreover, we aim to avoid the complexity and cost of the encrypted transport session. (We showed above that the encrypted transport session doesn't solve both attacks anyway.)

We propose Session Key Authorisation Protocol (SKAP), which has the following advantages over the existing OIAP and OSAP protocols:

- It generalises OIAP and OSAP, providing a session type that offers the advantages of both. In particular, it can cache a session secret to avoid repeatedly requesting the same authdata from a user (like OSAP), and it allows different objects within the same session (like OIAP).

– It is a long-lived session. In contrast with OSAP, it is not necessary to terminate the session when a command introduces new authdata.
– It allows authdata to be shared among users, without allowing users that know authdata to impersonate the TPM.
– In contrast with existing TPM authorisation, it does not expose low-entropy authdata to offline dictionary attacks [**?**].

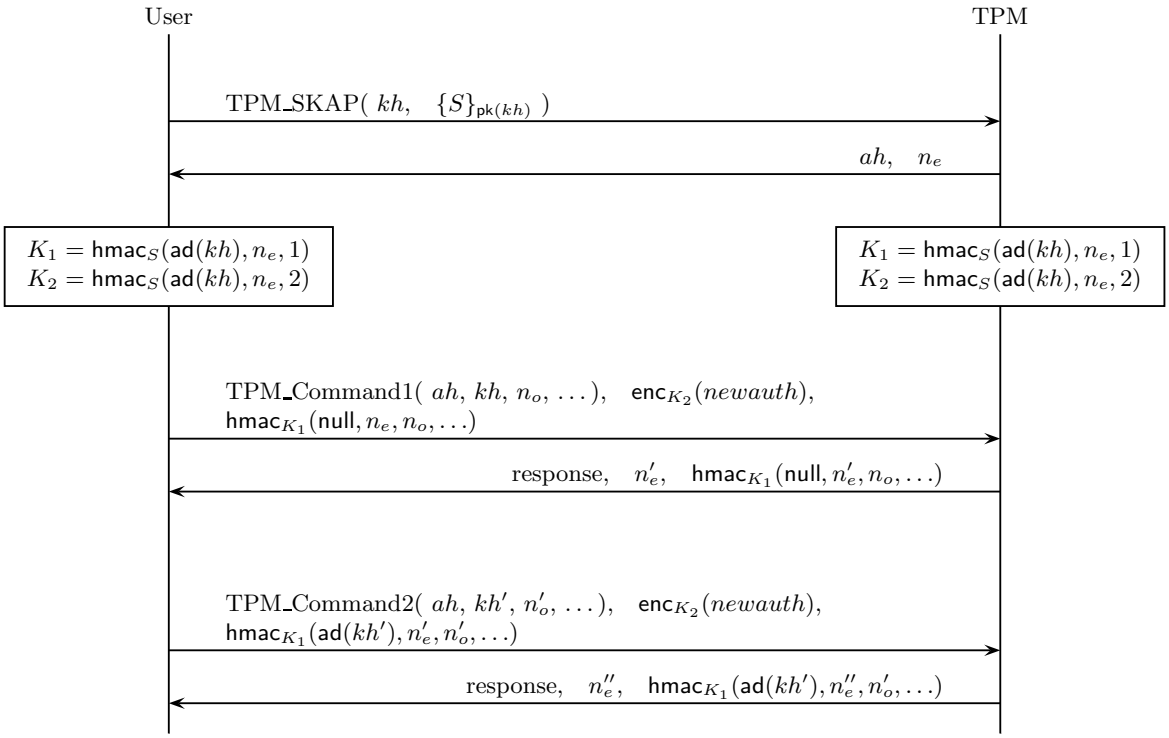

Fig. 2. Establishing a session using *Session Key Authorisation Protocol*, and executing two commands in the session. The session is established relative to a loaded key with handle kh. Command1 uses that key, and therefore does not need to cite authdata. Command2 uses a different key, and cites authdata in the body of the authorisation HMAC.

The message exchanges between a user process and the TPM in the SKAP protocol is illustrated in Figure 2. Similarly to OSAP, an SKAP session is established relative to a loaded key with handle (say) kh. The secret part of this

key $\mathsf{sk}(kh)$ is known to the TPM and the public part $\mathsf{pk}(kh)$ is known to all user processes which want to use the key. At the time the session is established, the user process generates a high-entropy session secret S, which could be created as a session random number, and sends the encryption $\{S\}_{\mathsf{pk}(kh)}$ of S with $\mathsf{pk}(kh)$ to the TPM. Theoretically any secure asymmetric encryption algorithm can be used for this purpose; in the TPM Specification uses RSA-OAEP [**?**] throughout, so we propose to use that too. The TPM responds with an authorisation handle ah and the first of the rolling nonces, n_e, as usual. Then each side computes two keys K_1, K_2 from S by using a MAC function keyed on S. The authdata $\mathsf{ad}(kh)$ for the key and the nonce n_e are cited in the body of the MAC. Any secure MAC function is suitable for our solution, but the TPM specification uses HMAC [**?**] for other purposes so we use that too.

Command1 in the illustrated session uses the key $(\mathsf{sk}(kh), \mathsf{pk}(kh))$ for which the session was established. The authorisation HMAC it sends is keyed on K_1, a secret known only to the user process and the TPM. In contrast with OSAP, this secret is not available to other users or processes that know the authdata for the key. Moreover, K_1 is high-entropy even if the underlying authdata is low entropy (thanks to the high-entropy session secret S). New authdata (written $newauth$) that Command1 introduces to the TPM is encrypted using K_2. In the figure, $enc_{K_2}(newauth)$ denotes the result of encrypting $newauth$ with a symmetric encryption algorithm using the secret key K_2. In general, any secure symmetric encryption scheme can be used in this solution. More specifically, in order to guarantee against not only eavesdropping but also unauthorised modification, we suggest using authenticated encryption as specified in [**?**]. One example is AES Key Wrap with AES block cipher [**?**].

In contrast with OSAP, SKAP sessions may use keys other than the one relative to which the session was established. Command2 in Figure 2 uses a different key, whose handle is kh'. Authdata for that key is cited in the body of the HMAC that is keyed on S.

3.1 The example revisited

We revisit the authorisation example described in Section 2.1, where the user wants to perform three commands, TPM_CreateWrapKey, TPM_LoadKey2 and TPM_Seal in a short period. We briefly demonstrate how these commands can be run in a single session (Figure 3). Suppose that the user starts from a parent key whose handle is pkh, and whose authdata $\mathsf{ad}(pkh)$ is well-known. (This parent key might be SRK, for example.) By following the SKAP protocol, the user first establishes a session for the parent key. To do this, he chooses a 160-bit random number as the session secret S, then encrypts S with the public part of the parent key and sends $\{S\}_{\mathsf{pk}(pkh)}$ to the TPM.

After that both sides compute two keys K_1 and K_2 based on the values S and $\mathsf{ad}(pkh)$. Then the user sends TPM_CreateWrapKey as TPM_Command1 in Figure 2 along with an encrypted new authorisation data for the requested key and HMAC for integrity check. The TPM responds the command with a key blob for the newly created key. When receiving any message which shows

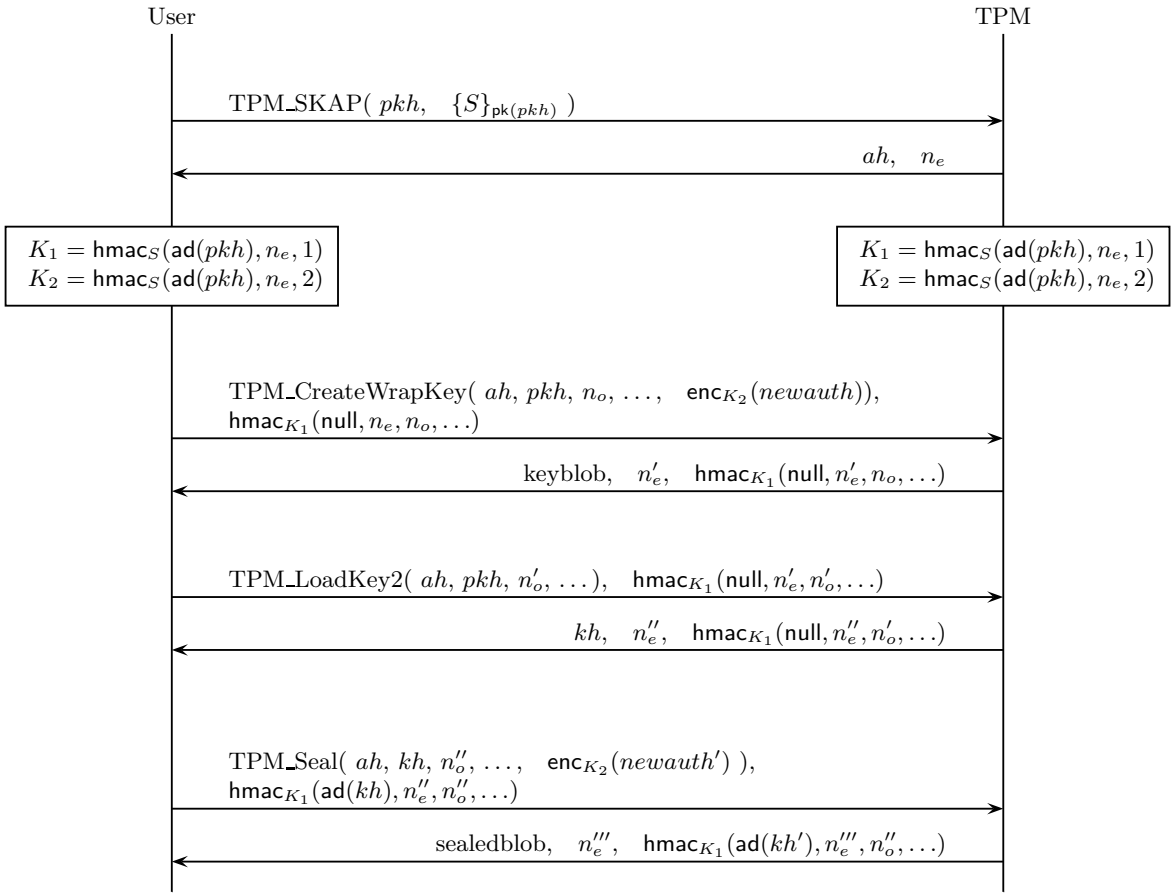

Fig. 3. An example of SKAP, showing creating a key, loading the key, and sealing with the key in a single SKAP session. Compare Figure 1.

either of these two keys K_1 and K_2 has been used, the user is convinced that he must be talking to the TPM and the TPM knows that its communication partner knows $\mathsf{ad}(pkh)$.

When the user wants to use this key (for example, for the sealing function), he sends the TPM the second command TPM_LoadKey2 in the same session. Since this also uses the parent key, it is again an example of Command1. The user and the TPM carry on using K_1 for authentication. Since TPM_LoadKey2 does not introduce new authdata, K_2 is not used. After the loading key process succeeds, the user sends the last command TPM_Seal. This command uses the newly created and loaded key, which is not the key for which the session is created. Therefore it is an example of Command2 in the figure, and the authdata for the key is required. The command uses the session keys K_1 and K_2 for authentication and protection of the sealed blob authdata, as before. So as we have seen that a single session of the SKAP protocol can handle multiple commands comfortably. The commands are shown in Figure 3. Comparison with Figure 1 shows a reduction from 12 to 8 messages, showing that that our protocol is more efficient as well as more secure.

4 Verification

We have modelled the current OSAP authorisation protocol using ProVerif [**?**,**?**]. ProVerif is a popular and widely-used tool that checks security properties of protocols. It uses the Dolev-Yao model; that is, it assumes the cryptography is perfect, and checks protocol errors against an active adversary that can capture and insert messages, and can perform cryptographic operations if it has the relevant keys. ProVerif is particularly good for secrecy and authentication properties, and is therefore ideal for our purpose. ProVerif is easily able to find the shared authdata attack of section 2.3. It shows both failure of secrecy and failure of authentication. We have also modelled the new proposed protocol SKAP, and ProVerif confirms the secrecy and authentication properties.

Our ProVerif code scripts for OSAP and SKAP are shown in Appendixes 1 and 2 respectively. In both models, there are two processes, representing the user process and the TPM. The user process requests to start a new session (respectively OSAP or SKAP) and then requests the execution of a command, such as TPM_CreateWrapKey to create a new key. The user process then checks the response from the TPM, and (in our first version) declares the event successU.

The TPM process provides the new session, executes the requested command (after checking correct authorisation), and provides the response to the calling user process. It declares the event successT.

The properties we verify are

- query attacker:newauth
- query ev:successU() ==> ev:successT()

The first one checks if *newauth* is available to the attacker. The second one stipulates that if the user declares success (i.e. the user considers that the command has executed correctly), then the TPM also declares success (i.e. it has

executed the command). If this property is violated, then potentially an attacker has found a means to impersonate the TPM.

We expect the secrecy property (first query) to fail for OSAP and succeed for SKAP, and this is indeed the case. The correspondence property (second query) is also expected to expected to fail OSAP and succeed for SKAP. Unfortunately the second query fails for both models, for the trivial reason that the TPM can complete the actions in its trace and then stop just before it declares success. To avoid this trivial reason, we extend the user process so it asks the TPM to prove knowledge of the new authdata introduced by the command, before it declares success. Now if the user declares success, the TPM should have passed the point at which it declares success too. If it has not, then an attacker has found a means to impersonate the responses of the TPM.

With this modification, we find an attack for each of the properties for OSAP, demonstrating the attack of section 2.3. ProVerif proves that SKAP satisfies both properties, demonstrating its security.

5 Conclusion

Sharing authorisation data between several users of a TPM key is a practice endorsed by the Trusted Computing Group [**?**,**?**, Design principles, §14.5, §14.6, §30.2, §30.8], but it makes the TPM vulnerable to impersonation attacks. An attacker in possession of the authorisation data for the storage root key (which is the authdata most likely to be shared among users) can completely usurp the secure storage functionality of the TPM.

The encrypted transport sessions of the TPM solve this problem, but they do not solve the related problem of guessing attacks (also known as dictionary attacks) on weak authdata, reported in [**?**]. The solution proposed for guessing attacks does not solve the problem of shared authdata. Therefore, a re-design of the TPM authorisation sessions is necessary.

We propose SKAP, a new authorisation session, to replace the existing authorisation sessions OIAP and OSAP. It generalises both of them and improves them in several ways, in particular by avoiding the TPM impersonation attack and the weak authdata attack.

We have analysed the old authorisation sessions and the new proposed one in ProVerif, the protocol analyser. The results show the vulnerability of the old sessions, and the security of the new one.

Appendix 1: ProVerif script for OSAP

```
free null, c, one, two.
fun enc/2. fun dec/2. fun senc/2. fun sdec/2.
fun hmac/2. fun pk/1. fun handle/1.

equation dec(sk, enc(pk(sk), m)) = m.
equation sdec(k, senc(k, m)) = m.

(* Queries. Uncomment one or the other. *)
(* query attacker:newauth. *)                        (* ATTACK FOUND *)
(* query ev:successU() ==> ev:successT(). *)         (* ATTACK FOUND *)

let User =
        (* request an OSAP session *)
        new no;
    new noOSAP;
    out(c, (kh, noOSAP));
    in(c, (ah, ne, neOSAP) );
    let K = hmac(authdata, (neOSAP, noOSAP)) in

    (* request execution of a command, e.g. TPM_CreateWrapKey *)
    new newauth;
    out(c,  no);
    out(c,  senc(K,newauth) );
        out(c,  hmac(K,(ne,no)) );

        (* receive the response from the TPM, and check it *)
    in(c, (r, hm) );
    if hm = hmac( K , r) then

        (* check that the TPM has newauth *)
    new n;
    out(c, n);
    in(c, hm2);
    if hm2=hmac(newauth,n) then
    event  successU(  ).

let TPM =
        (* handle the request for an OSAP session *)
    new ne;
    new neOSAP;
    in(c, noOSAP );
    out(c, (ne, neOSAP) );
    let K = hmac(authdata, (neOSAP, noOSAP)) in

    (* execute a command from the user, e.g. TPM_CreateWrapKey *)
    in(c, (no, encNewAuth, hm));
    if hm = hmac(K, (ne,no)) then
    let newauth = sdec(K, encNewAuth) in
```

```
        (* return a response to the user *)
    new response;
    out(c, ( response, hmac( K , response) ));
    event successT();

        (* if asked, prove knowledge of newauth *)
    in(c, n);
    out(c, hmac(newauth,n)).

process
        new skTPM; (* secret part of a TPM key *)
    let pkTPM = pk(skTPM) in  (* public part of a TPM key *)
    new authdata;   (* the shared authdata *)
    let kh = handle(pkTPM) in
    out(c, (pkTPM, authdata, kh) );

    ( !User | !TPM )
```

Appendix 2: ProVerif script for SKAP

```
free null, c, one, two.
fun enc/2. fun dec/2. fun senc/2. fun sdec/2.
fun hmac/2. fun pk/1. fun kdf/2. fun handle/1.

equation dec(sk, enc(pk(sk), m)) = m.
equation sdec(k, senc(k, m)) = m.

(* Queries. Uncomment one or the other. *)
(* query attacker:newauth. *)                   (* SECRECY HOLDS *)
(* query ev:successU() ==> ev:successT(). *)   (* CORRESPONDENCE HOLDS *)

let User =
        (* request an OSAP session *)
    new K;
        new no;
    out(c, (kh, enc(pkTPM, K)) );
    in(c, (ah, ne));
    let K1 = hmac(K, (authdata, ne, one)) in
    let K2 = hmac(K, (authdata, ne, two)) in

    (* request execution of a command, e.g. TPM_CreateWrapKey *)
    new newauth;
    out(c, ( no, senc(K2,(ne,no,newauth)), hmac(K1,(null,ne,no)) ) );

        (* receive the response from the TPM, and check it *)
    in(c, (response, hm) );
```

```
    if hm = hmac( kdf(K1,newauth), response) then

        (* check that the TPM has newauth *)
    new n;
    out(c, n);
    in(c, hm2);
    if hm2=hmac(newauth,n) then
    event  successU(  ).

let TPM =
        (* handle the request for an OSAP session *)
    new ne;
    in(c, encSessKey );
    let K = dec(skTPM, encSessKey) in
    out(c, ne);
    let K1 = hmac(K, (authdata, ne, one)) in
    let K2 = hmac(K, (authdata, ne, two)) in

    (* execute a command from the user, e.g. TPM_CreateWrapKey *)
    in(c, (no, encNewAuth, hm));
    if hm = hmac(K1, (null,ne,no)) then
    let (ne',no',newauth) = sdec(K2, encNewAuth) in
    if ne'=ne then
    if no'=no then

        (* return a response to the user *)
    new reponse;
    out(c, ( response, hmac( kdf(K1,newauth), response) ));
    event successT();

        (* if asked, prove knowledge of newauth *)
    in(c, n);
    out(c, hmac(newauth,n)).

process
        new skTPM; (* secret part of a TPM key *)
    let pkTPM = pk(skTPM) in  (* public part of a TPM key *)
    new authdata;   (* the shared authdata *)
    let kh = handle(pkTPM) in
    out(c, (pkTPM, authdata, kh) );
    ( !User | !TPM )
```