

How To Bind A TPM's Attestation Keys With Its Endorsement Key

Liqun Chen, Nada El Kassem and Christopher J.P. Newton

University of Surrey

*Corresponding author: nada.elkassem@surrey.ac.uk

Abstract

A trusted platform module is identified by its endorsement key, while it uses an attestation key to provide attestation services, for example, signing a set of platform configuration registers, providing a timestamp or certifying another of its keys. This paper addresses the problem of how a certificate authority binds the endorsement and attestation keys together. This is necessary for the authority to be able to reliably certify the attestation key. This key binding also enables the authority to revoke the attestation key should the endorsement key be compromised. We study all of the existing solutions and show that they either do not solve the problem or cannot be implemented with a real trusted platform module (or both). We propose a new solution which addresses this problem. We develop a security model for our solution and provide a rigorous security proof under this model. We have also implemented the solution using a real trusted platform module, and our implementation results show that this solution is feasible and efficient.

Keywords: key certificate and revocation, key binding, attestation key, endorsement key, trusted platform module

1. Introduction

A Trusted Platform Module (TPM) is a cryptographic coprocessor. It is a tamper-resistant device and enables trust in computing platforms in general. The specifications of TPMs [1, 2] have been developed by the Trusted Computing Group (TCG), an international industry standard body. As stated by the TCG, more than a billion devices use the TPM technology; virtually all enterprise PCs, many servers and embedded systems include a TPM [3]. Microsoft states [4] that 'Microsoft and other industry stakeholders continue to improve the global standards associated with TPM and find more and more applications that use it to provide tangible benefits to customers'. Since 2016 it has been a requirement that systems running Windows 10 should have a TPM 2.0 present and enabled [5]. The TPM is used for a range of functions [4] including several attestation services, e.g. measured boot and health attestation. The TPM serves as a Root-of-Trust on behalf of its host platform.

In a typical scenario, the TPM measures the platform software/firmware state, then provides a digitally signed measurement report to a verifier who remotely communicates with the platform. The correctness of the report is acceptable to the verifier because the TPM is tamper-resistant. The attestation service is not only software and firmware measurement, it can also be used to, for example, reliably provide a timestamp, a key certificate or an audit record of a TPM session. The TPM attestation service can be used in different applications [6–12]. In many cases, user privacy is important. A user may not want their transactions to be monitored when they use their TPM to confirm their access to a service.

The signature key used for the TPM attestation service is referred to as an **Attestation Key (AK)**. To preserve user privacy, the AK is not used as the TPM's identity. However, the external verifier wants to make sure that given attestation information was

created by a genuine TPM even if it is unidentified. Each TPM has an **Endorsement Key (EK)**, usually certified by the TPM manufacturer, that is used to represent its identity. To ensure the user's privacy, this key is an asymmetric encryption key rather than a signature key. An encryption key has the deniability property, meaning that after receiving a ciphertext and decrypting it to obtain the plaintext, the decrypter can deny what they have done, and the encrypter cannot convince a third party that the decrypter carried out the decryption. The encrypter can simulate all the plaintext and ciphertext and therefore, cannot prove that the decrypter has been involved. This is the opposite of a digital signature.

An **Attestation Certificate Authority Solution (ACAS)** provides a certificate confirming that an AK belongs to a TPM. To do this, the Certificate Authority (CA) is given a AK-EK pair and needs to confirm that they belong to the same TPM. In this case, the CA provides a **credential CRE for the AK**. Following the TPM 2.0 specification (section 24.2 of Part 1 [1]), the ACAS-in-TPM-2.0 protocol works as shown in Fig. 1, and the notation is listed in Table 1. This notation will be used throughout the whole paper.

The protocol involves a set of CAs, \mathcal{C} , a set of TPMs, \mathcal{T} , and a set of the hosts, \mathcal{H} . A single CA is denoted by $c_j \in \mathcal{C}$, as well as a TPM $t_i \in \mathcal{T}$ and a host $h_i \in \mathcal{H}$. Each host, h_i , corresponding to a TPM, t_i , c_j has a signature and verification key pair $ck_j = (cpk_j, cs_kj)$. t_i has a public and secret Endorsement Key pair, $ek_i = (epk_i, esk_i)$ used for an asymmetric encryption scheme. It is assumed that the CA has access to an authentic copy of epk_i and the host has access to an authentic copy of cpk_j and epk_i . By an authentic copy of a key we mean that the key is believed to be correctly generated and cryptographically bound with its owner; for example, this can be achieved using standard PKI (Public Key Infrastructure) digital certificates. Throughout the paper, we omit the presentation of certificates.

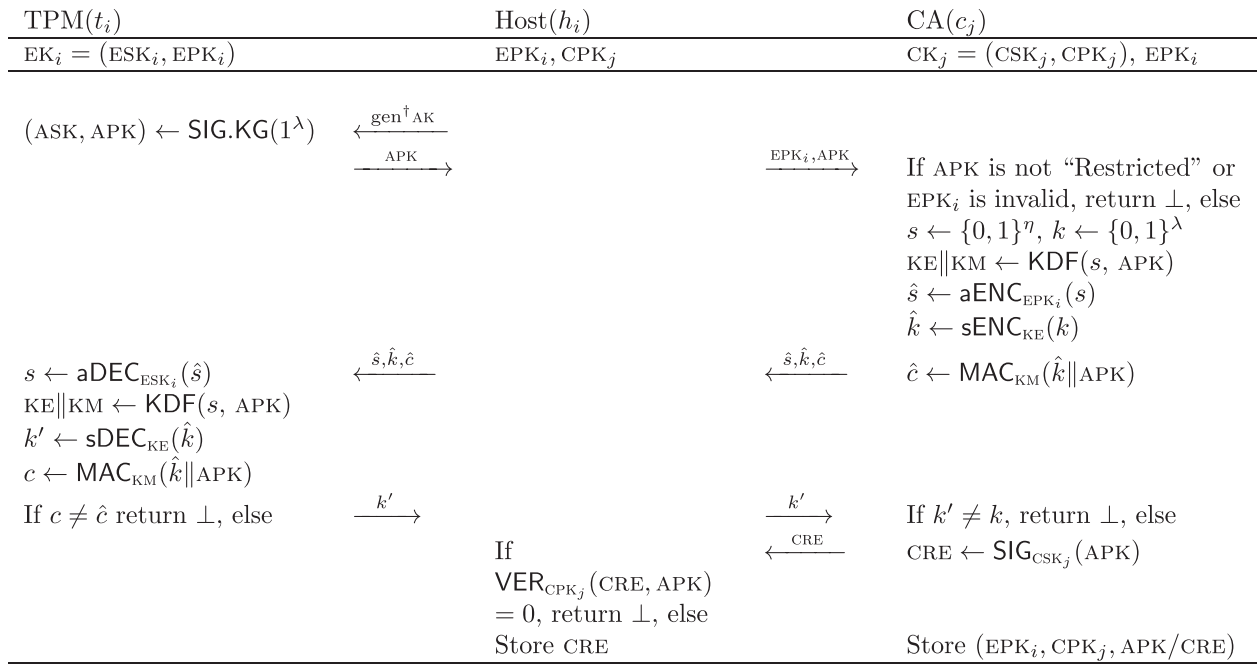


Figure 1. The Attestation Certificate Authority Solution in TPM 2.0 (ACAS-in-TPM-2.0) [†] Request generating a ‘Restricted’ signing key AK

Table 1. Notation and acronyms

Notation and acronym	Meanings
ACAS	Attestation Certificate Authority Solution
EK, EPK/ESK	endorsement key and its public/private portion
AK, APK/ASK	attestation key and its public/private portion
CK, CPK/CSK	CA’s key and its public/private portion
SCER	self-certificate for EK, i.e. a signature on EK under AK
CRE	credential for AK, i.e. a signature on AK under CK
KG	key generation algorithm
aENC/aDEC	asymmetric encryption/decryption operation
sENC/sDEC	symmetric encryption/decryption operation
SIG/VER	signing/verification operation
MAC	message authentication code operation
KDF	key derivation function
H	hash function
x y	concatenation of two data strings x and y

To support different requirements for user privacy, the AK can be used to provide conventional digital signatures or **Direct Anonymous Attestation (DAA)** signatures [13]. Given a conventional signature a verifier cannot trace it back to its signer but the CA who provided the AK credential can. For a DAA signature even the CA cannot find which TPM was used, and it can also meet more flexible requirements by allowing a user to choose pure anonymity (individual signatures are unlinkable) or pseudonymity (signatures are linkable but still untraceable) [14]. In the TPM 2.0 specifications [1], an ACAS is used to generate a credential for both types of AK. In the DAA literature, this solution is called the DAA Join protocol and the CA is called the DAA issuer. Throughout the paper, except for specific discussions, we do not distinguish whether an AK is a conventional signing key or a DAA key.

We now demonstrate that the ACAS-in-TPM-2.0 protocol suffers from a man-in-the-middle attack from a rogue and undetected TPM. Let the rogue TPM be denoted by t_A and

the honest one be t_B . t_A replaces the message (EPK_B, APK_B) with (EPK_A, APK_B) . The CA follows the protocol and returns the values \hat{s}_A (a ciphertext under EPK_A), and \hat{k} and \hat{c} that are computed using APK_B . t_A decrypts \hat{s}_A under ESK_A , recomputes \hat{s}_B using EPK_B , keeps everything else unchanged and sends them to t_B . Both the CA and t_B follow the protocol correctly, and they will not detect this attack. As a result, the CA will issue the credential CRE for APK_B and record $(EPK_A, CPK, APK_B/CRE)$. This is a mismatching key binding.

We, for the first time, propose that a secure ACAS protocol must satisfy the key binding property, i.e. after issuing the credential CRE for an AK, the CA has a record including $(EPK, APK/CRE)$, where the EK and AK belong to the same TPM.

1.1. Motivation of our proposal

Key binding is indispensable to build a revocation infrastructure. Consider the scenario that the CA is aware that the TPM’s EK is compromised after it issues the credential to the TPM’s AK. Key binding enables the CA to put the AK into a revocation list. If the AK is a traditional signature key, its public portion APK can be added in the key revocation list. If the AK is a DAA key, the CA can let the TPM provide a signature signed under the AK during the process of issuing the AK credential, and this signature can be added in the signature revocation list, e.g. using Intel’s Enhanced Privacy ID (EPID) scheme [15, 16]. EPID is widely used and Intel announced at the RSA Conference in 2016 that it had shipped over 2.4B EPID keys since 2008. Without the key binding, the CA cannot revoke any AKs belonging to this rogue TPM. As key revocation is necessary in the real-world, key binding is an important feature.

Key binding is also required if signatures need to be traceable. Let us see a rewarding service as a use case. It allows users to earn incentives in an anonymous way, i.e. a user can choose to link or unlink one transaction to their identity and/or other transactions. If a user reaches a reward point, they want to link their signature to the original credential generation protocol. If the key binding property is held, the CA can confirm which TPM should obtain the reward. If the CA has a mismatching record binding an honest

user's AK with a corrupted TPM's EK , the corrupted TPM will benefit from the honest TPM's contributions.

1.2. Challenge of building a correct ACAS

The difficulty of building the TPM attestation CA solution is not only because of the TPM's 'strong' privacy requirements (as discussed before) but also because of the TPM's 'weak' communication and authentication capabilities. A TPM is embedded inside its host platform. Any communication between the TPM and the outside environment is via the host and the TPM can only respond to its host's commands. The authentication between the TPM and CA is also via the host. A typical process is that the CA authenticates the TPM and the host authenticates the CA. In the security threat model of the trusted computing technology, the host can be an active attacker, who can read and modify messages sent from or to the TPM, block the communication between the TPM and external partner and can even coordinate some rogue and undetected TPMs.

We observe that a standard 2-party unilateral or mutual entity authentication mechanism cannot solve this problem. An asymmetric encryption based unilateral entity authentication mechanism cannot stop a rogue TPM to fool the CA. The details will be given when we analyse the existing solutions in Section 2. In the literature, there are papers discussing 2-party mutual authentication based on asymmetric encryption (e.g. [17, 18]), but they are not suitable for TPM use either, because the TPM is not able to authenticate the CA and the host is mistrusted by the CA. The TPM is a general purpose device, so it is impossible to include a particular CA's public key inside a TPM. This is not a design flaw but it does provide an interesting challenge for researchers.

1.3. Our Contributions

This work contributes in 4-fold:

- (i) Analysing all the existing ACAS. We observe: (i) following recommendations in the TPM specifications or (ii) using any of the existing 'concrete' DAA Join protocols, a CA could end the ACAS protocol with a wrong key binding, if a rogue and undetected TPM exists; and (iii) the 'generic' DAA Join protocols either in the DAA papers or in the standard documents (e.g. [19]) cannot be implemented with a real TPM.
- (ii) Proposing an enhanced ACAS protocol, denoted by eACAS. By enforcing the TPM's AK to hold necessary attributes, which are verified by the CA, our eACAS protocol guarantees the key binding between EK and AK .
- (iii) Formalising the security notions for eACAS. In the TPM 1.2 security model of an enhanced Privacy-CA Solution (ePCAS) [20], the key binding property is not discussed. We observe that the ePCAS protocol proved to be secure under this model suffers from a more powerful attack by colluding a dishonest Host and a rogue TPM, which leads to a mismatching key binding. The challenge is that the dishonest Host can fool its honest TPM. We add a new security experiment to explicitly define the key binding property, and modify the TPM and CA oracles in their model to capture the powerful attack and also to align with TPM 2.0 specification. We provide a rigorous proof of the key binding property of the proposed eACAS solution under this new security model.
- (iv) Implementing the eACAS protocol. We demonstrate how to implement this protocol using the existing TPM 2.0 commands with a real TPM 2.0 chip. This is the first key binding solution that has provable security and can be implemented

using a real TPM chip. Our implementation details show that this solution is feasible and efficient.

1.4. Paper Outline

We review the existing solutions in the next section to show that the TPM key binding problem has not yet been solved satisfactorily, then introduce our eACAS protocol and a comparison with the existing solutions in §3. In §4 we construct the eACAS protocol with the TPM 2.0 functionality. After that, we discuss the security model of eACAS in §5 and use it to prove key binding and other security properties of our eACAS protocol in §6. We then demonstrate how this protocol has been implemented using a real TPM 2.0 chip and discuss its performance in §7. Finally we conclude the paper in §8.

2. EXISTING SOLUTIONS

In the literature there are many existing solutions described. We divide them into three groups: the TCG solutions, the DAA solutions and the authenticated channel solutions. We now discuss why none of the existing solutions are satisfactory.

2.1. The TCG solutions

In 2003, the TCG suggested the **Privacy-CA Solution (PCAS)** for TPM version 1.2 [2]. Figure 2 gives a summary of this solution. Given the public portions of EK and AK (EPK , APK), the CA generates a nonce n_i and encrypts (APK, n_i) under the EPK . The TPM decrypts the ciphertext and returns n_i if the AK is owned by the TPM. The CA then creates the credential CRE for the APK then repeats the previous operation except that the nonce is replaced with a symmetric encryption key that is used to encrypt CRE . After the TPM releases the symmetric key, its host decrypts CRE . The protocol lets the TPM self check the possession of its AK twice.

In 2010, Chen and Warinschi [21] demonstrated that a corrupted and undetected TPM, say t_A , can mount a man-in-the-middle attack between an honest TPM, say t_B , and the CA. They suggested a countermeasure, as shown in Fig. 3, that lets a TPM sign the EK under the AK , and the CA verifies the signature before issuing the credential. This modification is called enhanced PCAS (ePCAS), and its security proof was given in [20]. In this paper, we further discuss their mitigation and notice that this mitigation can only work if t_B 's Host h_B is trusted to behave correctly, otherwise a malicious h_B can collude with t_A and fool t_B to sign the EK_A . This attack works for TPM 1.2, since the CA has no way to check whether t_B 's signature was created in the control of t_B or h_B .

The ePCAS of [20, 21] has not been adopted by the TCG, since the TCG moved from TPM version 1.2 to TPM version 2.0. To enhance the key management in TPM 2.0, the TCG has made two major changes: one is requiring an Attestation Key AK to have the attribute 'Restricted', meaning that such a key can only be used to sign a message created by the TPM, and another is replacing PCAS with ACAS, which, in this paper, is referred to as ACAS-in-TPM-2.0 and has been shown in Fig. 1. As we have demonstrated in Section 1, this protocol also suffers from the rogue TPM attack.

2.2. The DAA solutions

A DAA scheme includes a DAA Join protocol as well as other algorithms and protocols: Setup, Sign, Verify and Link. The details can be found in any DAA papers (e.g. [13, 22]). As mentioned before, the DAA Join protocol is virtually an ACAS protocol. There are many versions of the DAA Join protocol, e.g. [13, 23, 24, 15, 25, 26, 27, 28, 29, 30]. Due to the limited space, we will only discuss

TPM	Host	CA
ASK, ESK	EPK, APK, CPK	EPK, CSK
		$\xrightarrow{\text{EPK, APK}}$
		If EPK is invalid, return \perp else $n_I \leftarrow \{0, 1\}^t$ $a \leftarrow \text{aENC}_{\text{EPK}}(\text{APK} \ n_I)$
$\text{APK} \ n_I \leftarrow \text{aDEC}_{\text{ESK}}(a)$ If APK is unknown reject else	\xleftarrow{a} $\xrightarrow{n_I}$	\xleftarrow{a} $\xrightarrow{n_I}$
		If n_I is invalid, reject else create k $c \leftarrow \text{aENC}_{\text{EPK}}(\text{APK} \ k)$ $\text{CRE} \leftarrow \text{SIG}_{\text{CSK}}(\text{APK})$ $d \leftarrow \text{sENC}_k(\text{CRE})$
$\text{APK} \ k \leftarrow \text{aDEC}_{\text{ESK}}(c)$ If APK is unknown reject else	\xleftarrow{c} \xrightarrow{k}	$\xleftarrow{c, d}$
	$\text{CRE} \leftarrow \text{sDEC}_k(d)$ Verify CRE	Record (EPK, CRE/APK)

Figure 2. The TPM 1.2 privacy-CA solution (PCAS), presented in [21] (2003)

TPM	Host	CA
ASK, ESK	EPK, APK, CPK	EPK, CSK
load AK = (APK, ASK) If AK is unknown reject else		
$\text{SCER} \leftarrow \text{SIG}_{\text{ASK}}(\text{EPK})$	$\xrightarrow{\text{SCER}}$	$\xrightarrow{\text{APK, EPK, SCER}}$
		If EPK is invalid, return \perp If SCER/APK/EPK is invalid return \perp else create k $c \leftarrow \text{aENC}_{\text{EPK}}(\text{APK} \ k)$ $\text{CRE} \leftarrow \text{SIG}_{\text{CSK}}(\text{APK})$ $d \leftarrow \text{sENC}_k(\text{CRE})$
$\text{APK} \ k \leftarrow \text{aDEC}_{\text{ESK}}(c)$ If APK is unknown reject else	\xleftarrow{c} \xrightarrow{k}	$\xleftarrow{c, d}$
	$\text{CRE} \leftarrow \text{sDEC}_k(d)$ Verify CRE	Record (EPK, CRE/APK)

Figure 3. The enhanced privacy-CA solution (ePCAS) in [20, 21] (2010)

two examples below and all of the other schemes suffer from the similar attacks.

Figure 4 shows the Join protocol of the first DAA scheme, introduced by Brickell et al. in 2004 [13]. It works as follows: After receiving a join request with EPK and APK, the CA (called the DAA Issuer in their paper) chooses a nonce n_I and encrypts it under EPK, the TPM decrypts the nonce, computes a hash code, a , of APK and n_I and sends the value a to the CA. We now show that this Join protocol does not hold a key binding if there is a corrupted TPM, which, from the CA's view point, still has a valid EPK. We refer to the corrupted TPM as TPM_A and the honest TPM as TPM_B . TPM_B sends the CA $(\text{EPK}_B, \text{APK}_B)$ but this message is blocked by TPM_A , who, instead, sends the CA EPK_A together with APK_B . The CA follows the protocol and returns the value $c_A \leftarrow \text{aENC}_{\text{EPK}_A}(n_I)$. TPM_A decrypts it, recomputes $c_B \leftarrow \text{aENC}_{\text{EPK}_B}(n_I)$ and sends it to TPM_B , who then follows the protocol and returns $a \leftarrow \text{H}(\text{APK}_B \| n_I)$ to the CA. As the result, the CA will issue the credential of APK_B and record $(\text{EPK}_A, \text{CRE}/\text{APK}_B)$. This is a mismatched key pair.

As shown in Fig. 5, Chen, Morrissey and Smart [27] suggested another Join protocol for DAA. The notation *comm* stands for a commitment to ASK and it is a proof of possession of the AK. In this protocol, the EK is used for creating a signature σ on the commitment together with a challenge nonce. By a quick glance, this protocol satisfies the key binding property between

the TPM's EK and AK, as this is 'a standard self-certificate' solution. However, since the commitment is only a message to be signed and it is separated from the signature generation using the EK, a corrupted TPM can still fool the CA. To see this, we again refer to the corrupted TPM as TPM_A and the honest TPM as TPM_B . TPM_B sends the CA $(\text{EPK}_B, \text{APK}_B)$ but this message is blocked by TPM_A , who sends the CA $(\text{EPK}_A, \text{APK}_B)$ instead. The CA follows the protocol by returning the nonce n_I . TPM_A passes it to TPM_B , who follows the protocol by creating the commitment, *comm*_B, of ASK_B , computing the signature σ_B on *comm*_B $\| n_I$ under ESK_B , and sending it along with *comm*_B to the CA. This message is also blocked by TPM_A , who, instead, computes and sends $\sigma_A \leftarrow \text{SIG}_{\text{ESK}_A}(\text{comm}_B \| n_I)$ to the CA. As the result, like the attack to the previous Join protocol in [13], the CA will issue the credential of APK_B and record $(\text{EPK}_A, \text{CRE}/\text{APK}_B)$. This is again a mismatched key pair. Except for the problem of missing the key binding property, this protocol has two other issues: (1) Since the EK is a signature key, such authentication destroys the property of deniability; (2) In the TPM specifications, an EK is an encryption/decryption key pair, so this protocol cannot be implemented using any existing TPM chips.

The DAA scheme of [28] has been adopted by the TCG and specified in the TPM 2.0 specifications [1], and the DAA Join protocol was modified to be the ACAS-in-TPM-2.0, as shown in Fig. 1. As we have discussed before, this protocol does not hold the

TPM	Host	CA
ASK, ESK	EPK, APK, CPK	EPK, CSK
		$\xrightarrow{\text{EPK,APK}}$
		If EPK is invalid, return \perp $n_I \leftarrow \{0, 1\}^t$ $c \leftarrow \text{aENC}_{\text{EPK}}(n_I)$ Check $a \leftarrow \text{H}(\text{APK} n_I)$ Return \perp if the check fails, else $\text{CRE} \leftarrow \text{SIG}_{\text{CSK}}(\text{APK})$ Record (EPK, CRE/APK)
$n_I \leftarrow \text{aDEC}_{\text{ESK}}(c)$	\xleftarrow{c}	\xleftarrow{c}
$a \leftarrow \text{H}(\text{APK} n_I)$	\xrightarrow{a}	\xrightarrow{a}
Verify CRE		$\xleftarrow{\text{CRE}}$

Figure 4. The DAA Join protocol in the original DAA paper [13] (2004)

TPM	Host	CA
ASK, ESK	EPK, APK, CPK	EPK, CSK
		$\xrightarrow{\text{EPK,APK}}$
		If EPK is invalid, return \perp $n_I \leftarrow \{0, 1\}^t$ If σ not valid, return \perp $\text{CRE} \leftarrow \text{SIG}_{\text{CSK}}(\text{APK})$ Record (EPK, CRE/APK)
$\text{comm} \leftarrow \text{comm}(\text{ASK})$	$\xleftarrow{n_I}$	$\xleftarrow{n_I}$
$\sigma \leftarrow \text{SIG}_{\text{ESK}}(\text{comm} n_I)$	$\xrightarrow{\text{comm}, \sigma}$	$\xrightarrow{\text{comm}, \sigma}$
Verify CRE		$\xleftarrow{\text{CRE}}$

Figure 5. The DAA Join protocol in [27] (2009)

key binding property. Several earlier DAA papers [23, 24, 26, 31, 32] uses an alternative, in which the nonce n_I is encrypted under EPK and comm is a signature signed on n_I under ASK . Obviously this alternative has the same weakness.

Bernhard et al. [33] listed the DAA Join protocols of [13, 21, 27, 28] and suggested that one can ‘mix-and-match different authentication mechanisms’, but they did not provide details of their suggestion. Based on our analysis before, neither of these protocols holds the key binding property, so it is not clear how a mixing and matching mechanism could be built to support this property.

In summary, all the existing concrete DAA Join protocols cannot achieve a cryptographic binding between a TPM’s attestation key and its endorsement key.

2.3. Authenticated channel solutions

There are other documents in this field, including international standards. They require having an authenticated channel or a secure and authentic channel between a TPM and CA as a condition for generating ak credentials without demonstrating how to achieve it. For example, ISO/IEC 20008-2 [19] specifies several group signature schemes and DAA schemes. Let us quote from this document in the description of an RSA-based DAA scheme (Mechanism 2) and an EC-based DAA scheme (Mechanism 4): ‘The group membership issuing process requires a secure and authentic channel between the principal signer (i.e. the TPM) and the group membership issuer (i.e. the CA). How to establish such a channel is out scope of this mechanism’.

Several research papers have also discussed on building an authentication channel between the TPM and CA, but have not proposed any concrete scheme. Camenisch et al. [34] proposed a model for the authentication channel F_{auth^*} and said ‘We

design a functionality F_{auth^*} modelling the desired channel, which allows us to rather use the abstract functionality in the protocol design instead of a concrete sub-protocol. Then, any protocol that securely realizes F_{auth^*} can be used for the initial authentication’. More recent works [35–38] refer to F_{auth^*} from [34] for building the authentication channel, again ignore how to build such a channel.

Solving the key binding problem can be straightforward if there is ‘a mutually authenticated channel’ between the TPM and CA. By this channel, we mean that every message sent between these two entities can be authenticated by its receiver. But unfortunately, such a channel does not exist in PCAS, ACAS or a DAA Join protocol, since, as we have discussed before, the TPM is not able to authenticate the CA.

A number of papers (e.g. [15, 16, 23–26]) require a one-way authenticated channel; more specifically, every message from the CA to TPM is encrypted under the TPM’s endorsement key ek . We argue that this channel is not strong enough to avoid the man-in-the-middle attack, if the attacker is a corrupted but undetected TPM. The attack is the same as we discussed before, a corrupted TPM t_A can block the communications between an honest TPM t_B and the CA, then impersonate the CA to the t_B . Any message from the CA is encrypted under EPK_A , so can be decrypted by t_A , who then re-encrypts it under EPK_B and sends the ciphertext to t_B . As the authentication is only one-way, t_B is not able to find that the sender has been changed. t_B sends its ‘correct’ response to the CA, who then issues the credential for APK_B but makes a mismatching record ($\text{EPK}_A, \text{APK}_B/\text{CRE}$).

From the view point of the authors, the TPM key binding problem has not been solved satisfactorily, or at least a correct solution has not been properly documented and proved so far.

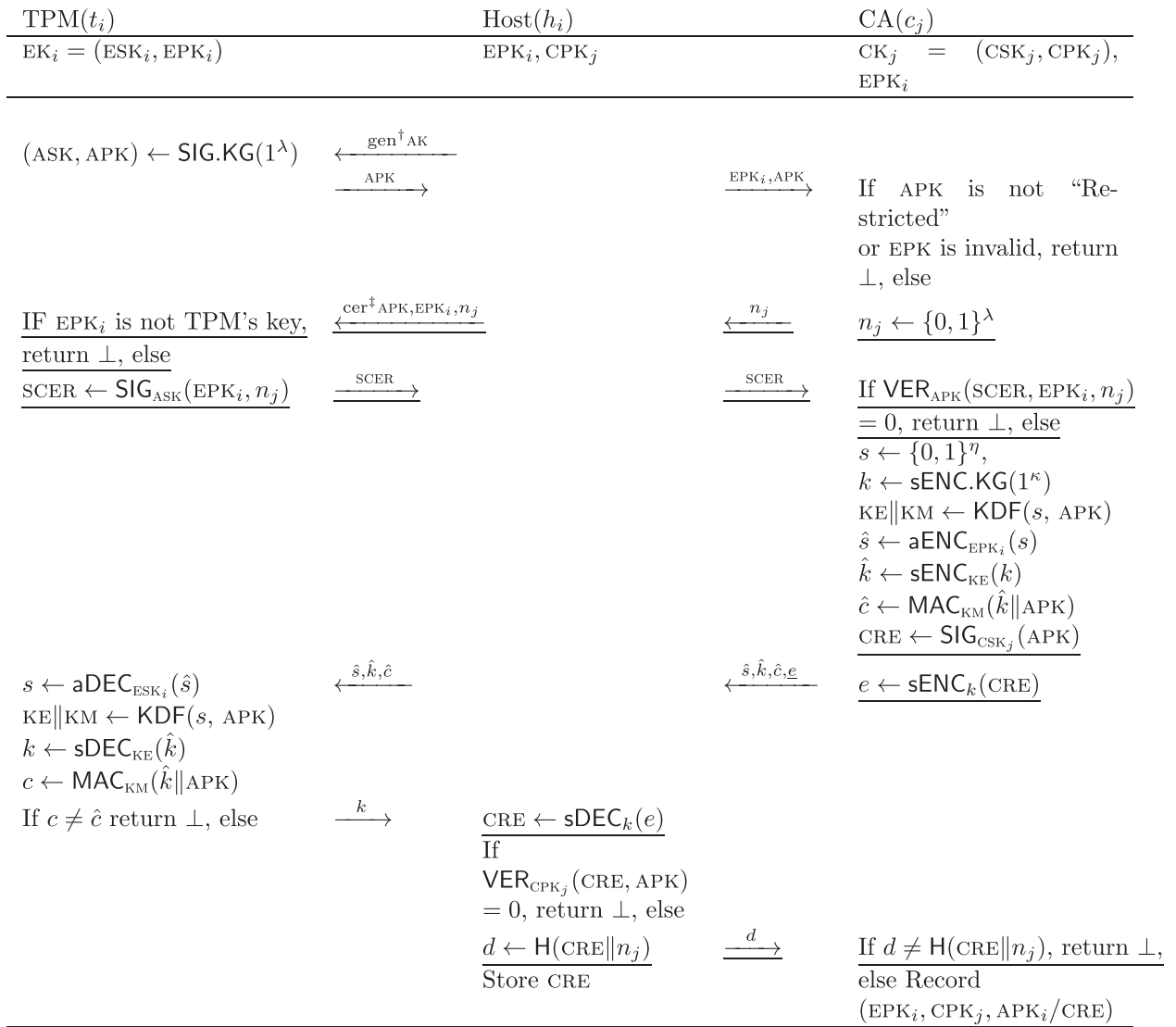


Figure 6. The enhanced Attestation Certificate Authority Solution (eACAS) [†] Request generating a ‘Restricted’ signing key AK^\dagger Request using the AK to certify EK_i

3. OUR ENHANCED ACAS

We now propose an enhanced Attestation CA Solution (eACAS) as shown in Fig. 6 (the notation is also in Table 1). The modification from ACAS-in-TPM-2.0 is underlined. The basic idea of letting the TPM sign EK under AK is taken from the ePCAS protocol in [21], which was proposed for the TPM 1.2 mitigation though having not been adopted by the TCG. We have added the ‘Restricted’ attribute to AK and also added a confirmation message from the Host to the CA, which will convince the CA of the binding $AK-EK$ key pairs.

In the protocol, the TPM t_i creates an Attestation Key AK that has a ‘Restricted’ attribute with the meaning that the AK will only be used to sign a message or a key created by the TPM. The public data of APK , including its attribute, and EPK_i are sent to the CA via the host h_i to request the CA creating a credential to use with the AK . The CA first checks the validation of EPK and the required attribute of the AK , then in order to check whether these two keys belong to the same TPM and whether the TPM is currently online, the CA chooses a fresh nonce n_j used as a challenge to the TPM. To respond to the challenge, the TPM first checks whether both EK_i and AK were created by itself. If this check fails, the TPM ignores

the challenge; otherwise it creates a self-certificate, i.e. certifying its endorsement key EK_i under AK and the nonce is treated as an external input. Upon receiving the TPM’s self-certificate, the CA checks its validation that includes checking if the correct nonce is included, and if the certificate is correct. If any of these checks fails, the CA will abort; otherwise the CA will create the credential, CRE , for the AK .

The CRE is not directly sent to the host. The CA uses the KEM-DEM algorithm to encrypt the credential under EPK_i and involving APK in the KEM operation. The TPM decrypts the KEM ciphertext, adds APK into generation of the symmetric encryption key and MAC key and checks the MAC value. If the check succeeds, the TPM releases the symmetric encryption key to the host, who then decrypts the credential and sends a confirmation to the CA.

The eACAS protocol satisfies the following four properties. The first three properties were introduced in [20] for the ePCAS protocol and we have added the last one.

- (i) *Third-party anonymity*: Given an AK and its credential CRE , no entity can determine which EK binds with the AK , except the engaged TPM and CA.

Table 2. A comparison with the existing solutions

Schemes	Third-party anonymity	Unforgeability	Deniability	Key binding	Implemented in a TPM
TPM 1.2 [2]	✓	✓	✓	–	✓
TPM 2.0 [1]	✓	✓	✓	–	✓
DAA Join protocols [13, 29] - [25–27]	✓	✓	✓	–	✓
[28]	✓	✓	–	–	–
DAA Join protocols from auth. channels [34–37]	✓	✓	✓	✓	–
ePCAS [20, 21]	✓	✓	✓	–	✓
Our eACAS	✓	✓	✓	✓	✓

- (ii) *Unforgeability*: No adversary can forge a credential CRE for an AK.
- (iii) *Deniability*: A TPM can deny that it was engaged in an ACAS transaction with a CA even if this transaction actually took place.
- (iv) *Key binding*: The CA's record (EPK_A, APK_{CRE}) is true, i.e. the EPK_A and AK belong to the same TPM.

Our protocol realizes the key binding property; this will be proved in details in Section 5. To see this, we again refer to the corrupted TPM as TPM_A and the honest TPM as TPM_B . TPM_B sends the CA (EPK_B, APK_B) but this message is blocked by TPM_A , who sends the CA (EPK_A, APK_B) instead. The CA follows the protocol by returning the nonce n_j . TPM_A passes it to TPM_B . Now TPM_B follows the protocol steps and needs to generate a self-certificate SCER on EPK_A , which is a signature on EPK_A using its restricted key APK_B , but this task cannot be completed since APK_B is a restricted key that can only be used to sign a message or a key created by the TPM_B and hence APK_B cannot be used to sign EPK_A . Therefore, the protocol will not be completed unless both the endorsement and the attestation keys originate from the same TPM; this preserves the key binding property.

A brief comparison with the existing schemes is given in Table 2. We consider these four security properties plus the implementation in real TPMs. This table demonstrates that our protocol is the first and only solution achieving all the required properties.

4. IMPLEMENT eACAS USING TPM 2.0 COMMANDS

4.1. Preliminaries

We first introduce the TPM 2.0 key management techniques and relevant commands along with the attestation CA's functions, as specified in [1], which will be used to construct the proposed eACAS protocol using TPM 2.0 commands.

4.1.1. TPM key management

TPM's keys are organized in key hierarchies. Except for leaf keys of a hierarchy, all the other keys serve as parents to protect their child keys. Each TPM key can be created as a primary key retrieved from a secret seed or a random key created randomly. As an implementation example, to construct our eACAS, we will let the TPM endorsement key EPK be a primary key, the TPM attestation key AK be a random key and EPK be the parent of AK . A TPM manages cryptographic keys using the following four items:

- *Key handle*: A key handle connects multiple commands that use this key. We let $k.h$ denote the handle of the key, k .
- *Key attributes*: Attributes of a key determine how the key can be used. The important attributes related to our purpose are 'fixedTPM', 'fixedParent', 'Decrypt', 'Sign' and 'Restricted'. The meanings of the first four attributes are clear from their names. The last one means that the key can only be used on structures that have a known format; e.g. when a signing key is restricted, the key can only sign the data created by the TPM itself. When a restricted signing key is used to certify another key, that certified key must be created by the TPM. We will use this feature to achieve the key binding property in the eACAS protocol.
- *Key public data and name*: For an asymmetric key with a public and secret portion, written as $k = (pk, sk)$, its public data (denoted by $k.p$) includes pk and its attributes, and its name (denoted by $k.n$) is a hash value of $k.p$.
- *Key blob*: A key stored outside of the TPM is in a format of a key blob that is associated with its parent key $PKEY$. For an asymmetric key pair $k = (pk, sk)$, the key blob is $k.b = (ENC_{KE}(sk), pk, MAC_{KM}(ENC_{KE}(sk) || k.n))$, where a symmetric encryption key KE and a MAC key KM are computed by $(KE, KM) \leftarrow KDF(PKEY, salt)$, and $salt$ is a data string used to make $PKEY$ reusable.

4.1.2. CA functions.

In the eACAS protocol, the CA has several functions, including *check_key* that checks if the given keys hold the required attributes, *check_cert* that checks if a given certificate is valid, *Verify_hash* that verifies if a hash value is correctly computed and *record* that records a data tuple to its database. All of these functions are standard or straightforward, so we do not explain them in details. However, the following function needs a more detailed explanation.

The *make_credential* function is used by the CA to create the credential blob for the TPM's AK , denoted by $CRE_t.b$, and it works as follows: Take $(AK.p, EPK_i, CSK_i, \eta, k)$ as input and output $CRE_t.b$, where η is the security parameter and $k \leftarrow sENC.KG(\eta)$ is a symmetric encryption key, and the CA computes

- $s \leftarrow \{0, 1\}^\eta$, $AK.n = H(AK.p)$,
- $KE = KDF(s, 'STORAGE', AK.n)$,
- $KM = KDF(s, 'INTEGRITY', NULL)$,
- $\hat{s} = aENC_{EPK_i}(s)$, $\hat{k} = sENC_{KE}(k)$,
- $\hat{c} = MAC_{KM}(\hat{k} || AK.n)$,
- $CRE_t.b = (\hat{s}, \hat{k}, \hat{c})$.

4.1.3. TPM commands

The following TPM commands are relevant.

- **TPM2_CreatePrimary** is used to create a key from a secret seed. It takes the indication of the seed as input and outputs a key handle for the created primary key.
- **TPM2_Create** is used to create a random key. It takes a parent key handle as input and outputs a key blob for the newly created key.
- **TPM2_Load** is used to load a key into the TPM. It takes a key blob and a key handle of its parent key as input and outputs a key handle for the loaded key.
- **TPM2_Certify** is used to certify a key. Taking the handles of a certified key and signing key, the TPM creates and outputs the self-certificate $scer$. In the eACAS protocol the certified key is ek and the signing key is ak . If the ak is a DAA key, except for **TPM2_Certify**, **TPM2_Commit** is also used to make a commitment to an ephemeral secret in the ECDAA signing procedure; please see [39] for the details.
- **TPM2_ActivateCredential** is run by the TPM to unwrap the credential blob $cre_t.b$ from *make_credential* and return the key k . It works as follows: Taking ek_i , ak and $cre_t.b = (\hat{s}, \hat{k}, \hat{c})$ as input, compute

- (iv) $s = aDEC_{EK_i}(\hat{s})$,
- (iv) $KE = KDF(s, 'STORAGE', ak.n)$,
- (iv) $KM = KDF(s, 'INTEGRITY', NULL)$,
- (iv) $k = sDEC_{KE}(\hat{k})$, $c = MAC_{KM}(\hat{k} || ak.n)$,
- (iv) if $c \neq \hat{c}$ then **abort**, else output k .

4.2. The eACAS Protocol in TPM 2.0 Commands

We now construct our enhanced Attestation CA Solution (eACAS) protocol, as described in Section 3, using the TPM 2.0 commands. In the TPM manufacturing time, each t_i creates an endorsement seed, $seed_i$, and stores it in its non-volatile memory, and also creates an endorsement key ek_i , which is a primary key based on $seed_i$, i.e. it can be recreated using **TPM2_CreatePrimary**, so t_i does not have to store it. However, since creating a primary key is slow (see Section 7 for an implementation example), this key can alternatively be stored in the TPM. In this case, the host has access to $ek_i.h$. We take this case in the following description.

There are two versions of eACAS, denoted by $eACAS_C$ and $eACAS_D$. $eACAS_C$ is suitable for the case that the ak is used in a conventional signature scheme, while $eACAS_D$ is a new proposal for the Direct Anonymous Attestation (DAA) Join protocol. In the later case, the ak is a DAA key and the attestation Certificate Authority (CA) is the DAA Issuer. The protocol message flows of $eACAS_C$ is shown in Fig 7. For $eACAS_D$, the only difference from this figure is that both **TPM_Commit** and **TPM2_Certify** are used in Step 4 to create $scer$; the details are in [39]. These two protocols each includes the following seven steps:

- (i) The host h_i asks its TPM t_i to create an attestation key ak to obtain the key blob $ak.b$. The ak must hold the attributes: 'Restricted', 'fixTPM', 'fixParent' and 'Sign'. The ak can be a DAA key (in $eACAS_D$) or a conventional signing key (in $eACAS_C$). The TPM does not store ak internally at this stage.
- (ii) To use ak , the host loads $ak.b$ to its TPM to obtain the key handle $ak.h$. The TPM will keep ak in its operational area until the key handle is flushed.
- (iii) The host sends $(EPK_i, ak.p)$ to the CA c_j to request for a credential for ak . c_j checks the validation of EPK_i and the attributes

of ak . If the check fails, c_j aborts; otherwise it creates and returns a nonce n_j with a sufficient length λ .

- (iv) The host asks its TPM to certify ek_i under ak and to include n_j in the signed message. To do this, if the ak is a conventional key ($eACAS_C$), only the **TPM2_Certify** command will be called; if the ak is a DAA key ($eACAS_D$), both the **TPM2_Commit** and **TPM2_Certify** commands will be called. In either version, the host obtains a TPM self-certificate $scer$ that uses the ak to certify the ek_i and meanwhile signs the nonce n_j .
- (v) The host sends $scer$ to the CA, who then checks the validation of $scer$. If the check fails, the CA aborts; otherwise it generates a fresh symmetric encryption key, encrypts cre under this key (the ciphertext is denoted by $cre_h.b$), uses the *make_credential* function to create a credential blob for the TPM $cre_t.b$ and outputs the whole credential blob $cre.b = (cre_t.b, cre_h.b)$.
- (vi) The host calls the **TPM2_ActivateCredential** command with $cre_t.b$ and the key handles of ak and ek_i to its TPM. If ak and ek_i have been loaded in the TPM's operational area and $cre_t.b$ matches with these two keys, the TPM returns a symmetric decryption key k .
- (vii) The host uses k to decrypt $cre_h.b$ to obtain cre , and checks the validation of cre . If the checks fail, the host aborts; otherwise it computes $d = H(cre || n_j)$ and returns it to the CA, who then checks the validation of this value. If the check fails, the CA aborts; otherwise it records $(EPK_i, CPK_j, APK/cre)$.

A successful eACAS run lets the CA holds a binding key record and end up with a conclusion that these two keys (ek_i, ak) belong to the same TPM t_i . We should highlight that a dishonest host can fool its embedded TPM in our eACAS by conducting a denial-of-service attack such as blocking the value d or sending other message to the CA instead of d ; however, this attack will not end with an incorrect key binding as the protocol will not be completed and hence no new binding record will be generated. A new binding record will only be generated when the CA receives a correct matching value of d .

5. SECURITY MODEL FOR eACAS

In this section we review and modify the security model of the ePCAS protocol in [20] to develop a security model for our eACAS protocol. The major modifications are (i) creating the key binding experiment to formalise the key binding property, (ii) changing the TPM and CA oracles to capture the attack that a malicious host makes an honest TPM sign a mismatching ek and (iii) adding the key attribute checking and credential confirmation to capture the key binding property. We also make several changes to align our model with the TPM 2.0 specifications, as ePCAS was designed for TPM 1.2.

5.1. Oracles needed in the security games

Let \mathcal{A} be the adversary aiming to break the eACAS protocol. The security definitions of both ePCAS and eACAS use the following set of oracles, but the content of each oracle is different due to the fact that they support different TPM versions. In addition, we let the CA check the TPM key attributes.

- **TPM(EPK_i, APK):** \mathcal{A} uses this oracle to interact with TPM t_i .
- **CA(CPK_j):** \mathcal{A} uses this oracle to interact with CA c_j .
- **CH_b($EPK_{i_0}, EPK_{i_1}, CPK_j$):** \mathcal{A} sends $(EPK_{i_0}, EPK_{i_1}, CPK_j)$ to the oracle and gets back (APK, cre) of the eACAS protocol executed by TPM t_{i_0} and CA c_j .

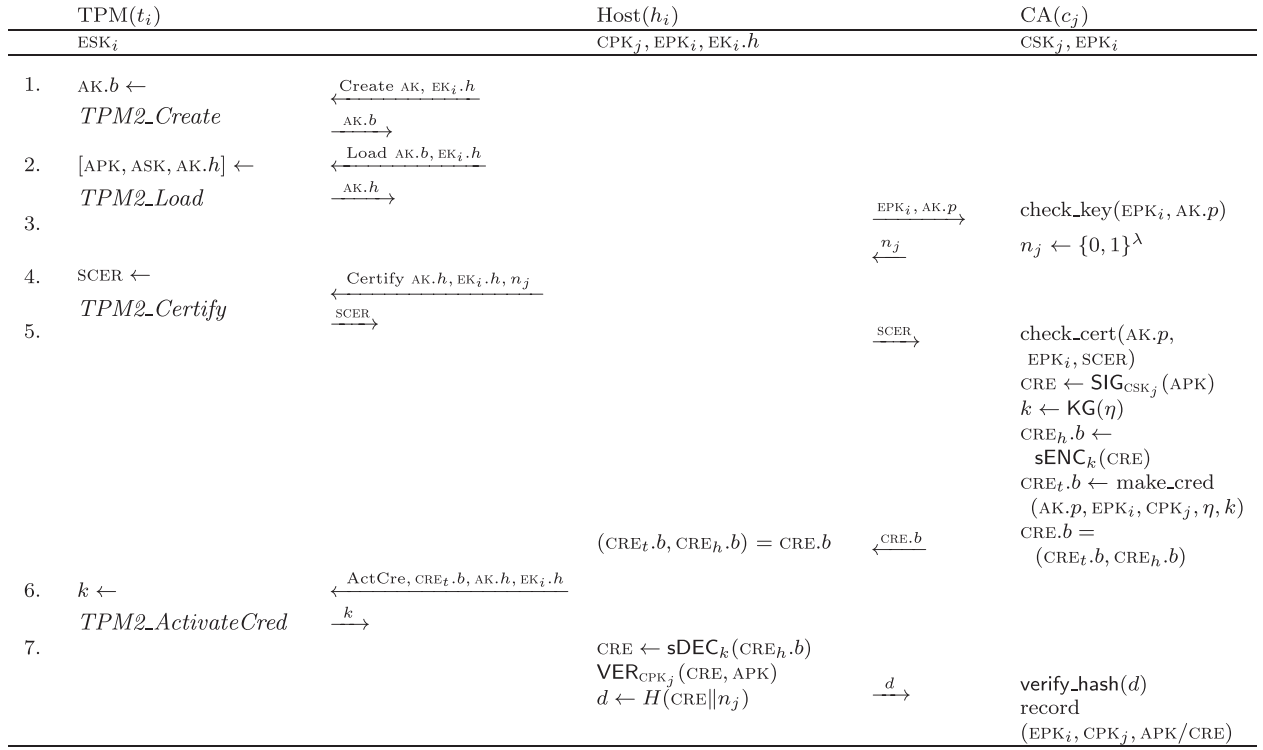


Figure 7. The eACAS protocol when AK is a conventional signature key (eACAS_C)

- TRAN(EPK $_i$, CPK $_j$, APK): The oracle returns a transcript between TPM t_i and CA c_j in a transaction.
- CorrCA(j): This oracle allows \mathcal{A} to access the signing key of CA c_j .
- CorrTPM(i): This oracle allows \mathcal{A} to access the endorsement key of TPM t_i .

The behaviour of the first four oracles, TPM, CA, CH_b and TRAN, is summarized in Fig. 8. The TPM oracle has an alternative TPM* in which the last step is omitted. In addition, \mathcal{A} is given access to CorrCA and CorrTPM oracles. Let η be a security parameter. On an input $1 \leq j \leq p(\eta)$ to the oracle CorrCA it returns CSK $_j$, and on an input $1 \leq i \leq p(\eta)$ to the oracle CorrTPM it returns ESK $_i$.

We adopt the definitions of the following three global variables from [20]:

- ValidKey maintains a list of pairs (EPK, APK), with the meaning that these two keys are held by the same TPM. ValidKey(EPK, APK) = 1/0 indicates that (EPK, APK) occurs/does not occur in ValidKey. This global variable is used by the TPM oracles.
- ValidTPM maintains a list of pairs (CPK, EPK); each such pair indicates that the CA with the public key CPK knows the TPM with the endorsement public key EPK. Also ValidTPM(CPK, EPK) = 1/0 indicates that (CPK, EPK) occurs/does not occur in ValidTPM. This global variable is shared by the CA and CH_b oracles.
- RegList maintains a list of (EPK, CPK, APK/CRE); each item is added when the CA oracle creates the CRE.

We propose two new global variables to capture the key binding property and they are accessed by the CA oracle.

- confirmedList maintains a list of (EPK, CPK, APK/CRE) and each item is added after the CA oracle receives the confirmation that the CRE has been retrieved.
- scerList maintains a list of (EPK, APK, SCER) and each item is added when the CA receives the SCER.

5.2. Security games

In the ePCAS security model, the security of the ePCAS protocol is associated with three properties, third-party anonymity, unforgeability and deniability [20]. We adopt these three properties to address the security of our proposed eACAS protocol and add a new security property: key binding.

Each security property is specified using a game-based experiment, as shown in Fig. 9. Except for the newly introduced key binding game, the others are similar to the games in [20], although we use different TPM and CA oracles to align with TPM version 2.0. In each game, first, the TPM endorsement keys and the CA signature keys are generated. We write CPK for the set {CPK $_1$, CPK $_2$, ...} of all public keys of the CAs, and EPK for the set {EPK $_1$, EPK $_2$, ...} of all public keys of the TPMs. The adversary is given all of the public keys. We do not impose any conditions on how the attestation keys are generated; the adversary is allowed to select arbitrary bit-strings for these keys, as long as they have the same distributions.

5.3. Third-party anonymity

An eACAS protocol is third-party anonymous if given a pair (CRE, APK), no third party can determine which TPM owns the APK, except for the engaged TPM and CA. The third-party anonymity experiment, specified in detail in Fig. 9, proceeds as follows. The adversary \mathcal{A} decides which CAs know which TPMs, by specifying a list ValidTPM $\subset \text{CPK} \times \text{EPK}$. \mathcal{A} also decides for each TPM what are the list of attestation keys that the TPM knows by specifying the list ValidKey $\subset \{\text{EPK}_1, \text{EPK}_2, \dots\} \times \{0, 1\}^*$. We require that ValidKey satisfies that for any $\text{APK} \in \{0, 1\}^*$ ValidKey(EPK $_i$, APK) = 1 then ValidKey(EPK $_k$, APK) = 0 for any $k \neq i$. This reflects the idea that the attestation keys valid for the TPM are disjoint.

\mathcal{A} first queries some TPM, CA, CorrTPM and CorrCA oracles, and then outputs a tuple (EPK $_{i_0}$, EPK $_{i_1}$, CPK $_j$), where EPK $_{i_0}$ and EPK $_{i_1}$ are two different TPM public keys and CPK $_j$ is a CA public key. Note that the attestation key generation algorithms of TPMs t_{i_0}

TPM(ESK_i)

- 1 create AK as a child of EK_i
send AK.b
- 2 load (AK.b)
if ValidKey(EPK, APK) = 0 then **abort**
else send AK.h
- 3 receive nonce n_j
SCER \leftarrow TPM2_Certify(ASK, EPK_i, n_j)
send SCER
- 4 receive CRE_t.b = ($\hat{s}, \hat{k}, \hat{c}$)
 $s = \text{DEC}_{\text{ESK}_i}(\hat{s})$
KE = KDF(s , "STORAGE", AK.n)
KM = KDF(s , "INTEGRITY", NULL)
 $k = \text{DEC}_{\text{KE}}(\hat{k})$, $c = \text{MAC}_{\text{KM}}(\hat{k} \parallel \text{AK.n})$
if $c \neq \hat{c}$ then **abort**
else send k

TPM*(ESK_i)

- 1 create AK as a child of EK_i
send AK.b
- 2 load (AK.b)
if ValidKey(EPK, APK) = 0 then **abort**
else send AK.h
- 3 receive nonce n_j
SCER \leftarrow TPM2_Certify(ASK, EPK_i, n_j)
send SCER

TRAN(EPK_i, CPK_j)

(APK, ASK) \leftarrow KG(η)
(CRE.b, n_j) \leftarrow CA(CPK_j, CSK_j, APK)
(SCER, k) \leftarrow TPM(EPK_i, ESK_i, APK)
 $\tau \leftarrow$ (SCER, CRE.b, k , n_j)
return τ

CA(CPK_j)

- 1 receive (AK.p, EPK_i)
if ValidTPM(CPK, EPK) = 0 then **abort**
if $0 \leftarrow \text{attribute_check}(\text{AK.p})$ then **abort**
else generate and send nonce n_j
- 2 receive (AK.p, EPK_i, SCER)
if $0 \leftarrow \text{VER}(\text{APK}, \text{EPK}_i, \text{SCER}, n_j)$ then **abort**
else append (APK, EPK_i, SCER) \in scerList
CRE = SIG_{CSK_j}(APK), AK.n = H(AK.p)
 $k \leftarrow \text{KG}(\eta)$, $s \leftarrow \{0, 1\}^t$
KE = KDF(s , "STORAGE", AK.n)
KM = KDF(s , "INTEGRITY", NULL)
 $\hat{s} = \text{ENC}_{\text{EPK}_i}(s)$, $\hat{k} = \text{ENC}_{\text{KE}}(k)$
 $\hat{c} = \text{MAC}_{\text{KM}}(\hat{k} \parallel \text{AK.n})$
CRE_t.b = ($\hat{s}, \hat{k}, \hat{c}$), CRE_h.b = ENC_k(CRE)
send CRE.b = (CRE_t.b, CRE_h.b)
append (EPK_i, CPK_j, APK/CRE) \in RegList
- 3 receive d
if $d \neq \text{H}(\text{CRE} \parallel n_j)$ then **abort**, else
append (EPK_i, CPK_j, APK/CRE) \in confirmedList

CH_b(EPK_{i0}, EPK_{i1}, CPK_j)

if ValidTPM(CPK_j, EPK_{i0}) \oplus
ValidTPM(CPK_j, EPK_{i1}) = 1 then **abort**
if CSK_j is corrupt then **abort**
if ESK₀ or ESK₁ is corrupt then **abort**
(APK, ASK) \leftarrow KG(η)
 $b \leftarrow \{0, 1\}$
(CRE.b, n_j) \leftarrow CA(CPK_j, EPK_{ib}, APK)
(SCER, k) \leftarrow TPM(EPK_{ib}, APK)
CRE \leftarrow DEC_k(CRE_h.b)
return (APK, CRE)

Figure 8. TPM, TPM*, CA, TRAN and CH_b oracles

Exp^{anon-b}_{eACAS, A}(η)

for each CA c_j do (CPK_j, CSK_j) \leftarrow C.KG(η)
for each TPM t_i do (EPK_i, ESK_i) \leftarrow E.KG(η)
ValidTPM, ValidKey \leftarrow A(CPK, EPK)
(EPK_{i0}, EPK_{i1}, CPK_j) \leftarrow
A^{TPM, CA, CorrCA, CorrTPM}(EPK, CPK,
ValidTPM, ValidKey)
 $\hat{b} \leftarrow$ A^{CH_b, TPM, CA, CorrCA, CorrTPM}(EPK,
CPK, ValidTPM)
return \hat{b}

Exp^{unforge}_{eACAS, A}(η)

for each CA c_j do (CPK_j, CSK_j) \leftarrow C.KG(η)
for each TPM t_i do (EPK_i, ESK_i) \leftarrow E.KG(η)
ValidTPM, ValidKey \leftarrow Auth(CPK, EPK)
RegList \leftarrow \emptyset
(EPK_{i*}, CPK_{j*}, APK*, CRE*) \leftarrow
A^{TPM_i, CA_j, CorrCA, CorrTPM}(EPK, CPK,
ValidTPM, ValidKey)
if VER(CPK_{j*}, (APK*, CRE*)) = 0 return 0
if CSK_{j*} is corrupt return 0
if (EPK_{i*}, CPK_{j*}, APK*, CRE*) \notin RegList return 1
if (EPK_{i*}, CPK_{j*}, APK*, CRE*) \in RegList and
ValidKey(EPK_{i*}, APK*) = 0 return 1
return 0

Exp^{keybind}_{eACAS, A}(η)

for each CA c_j do (CPK_j, CSK_j) \leftarrow C.KG(η)
for each TPM t_i do (EPK_i, ESK_i) \leftarrow E.KG(η)
ValidTPM, ValidKey \leftarrow Auth(CPK, EPK)
confirmedList \leftarrow \emptyset
(EPK_{i*}, CPK_{j*}, APK*, CRE*) \in confirmedList
 \leftarrow A^{TPM_i, CA_j, CorrCA, CorrTPM}(EPK, CPK,
ValidTPM, ValidKey)
if CSK_{j*} is corrupt return 0
if ValidKey(EPK_{i*}, APK*) = 0 return 1
return 0

Exp^{x-deni-b}_{eACAS, D}(η)

for each CA c_j do (CPK_j, CSK_j) \leftarrow C.KG(η)
for each TPM t_i do (EPK_i, ESK_i) \leftarrow E.KG(η)
ValidTPM, ValidKey \leftarrow Auth(CPK, EPK)
 $\tau_0 \leftarrow$ A^{O_x}(EPK, CPK, CSK, ValidTPM)
(if $x = s$ then O_x = TPM*;
if $x = w$ then O_x = TRAN)
 $\tau_1 \leftarrow$ S^{CorrCA, A}(EPK, CPK, CSK, ValidTPM)
 $\hat{b} \leftarrow$ D(τ_b), return \hat{b}

Figure 9. Experiments defining third-party anonymity, unforgeability, key binding and deniability

and t_i should be the same, for example, both are RSA keys or discrete logarithm keys. After generating the tuple $(\text{EPK}_{i_0}, \text{EPK}_{i_1}, \text{CPK}_j)$, \mathcal{A} queries a CH_b oracle (this oracle can be queried just once in an experiment) and some TPM, CA, CorrCA and CorrTPM oracles. \mathcal{A} is not allowed to query the TRAN oracle to the challenge phase, the challenge attestation public key APK to CA or TPM oracle or access the variable ValidKey after the CH_b oracle query. At the end of the execution, \mathcal{A} outputs a decision bit \hat{b} .

Definition 5.1. *Third-party anonymity.* We define the advantage of the adversary \mathcal{A} in the anonymity game as

$$\text{Adv}_{\text{eACAS}, \mathcal{A}}^{\text{anon-b}}(\eta) = \left| \Pr[\text{Exp}_{\text{eACAS}, \mathcal{A}}^{\text{anon-0}}(\eta) = 1] - \Pr[\text{Exp}_{\text{eACAS}, \mathcal{A}}^{\text{anon-1}}(\eta) = 1] \right|,$$

and say that the eACAS protocol is *anonymous* if $\text{Adv}_{\text{eACAS}, \mathcal{A}}^{\text{anon-b}}(\eta)$ is a negligible function of η for all polynomial time adversaries \mathcal{A} .

5.4. Unforgeability

An eACAS protocol is unforgeable if no adversary can forge a credential for an attestation public key of a TPM. The unforgeability experiment allows the adversary \mathcal{A} to corrupt some TPMs (i.e. we consider the chosen corrupted TPM attack). In the game, \mathcal{A} is given the variables ValidKey and ValidTPM. The unforgeability experiment, as specified in Fig. 9, proceeds as follows. The variables ValidTPM $\subset \text{cPK} \times \text{EPK}$ and ValidKey $\subset \{\text{EPK}_1, \text{EPK}_2, \dots\} \times \{0, 1\}^*$ are specified by an authority Auth. We require that ValidKey satisfies that for any $\text{APK} \in \{0, 1\}^*$ ValidKey(EPK_i, APK) = 1 then ValidKey(EPK_k, APK) = 0 for any $k \neq i$. This reflects the idea that the attestation keys valid for the TPM are disjoint. \mathcal{A} is initially given $(\text{EPK}, \text{cPK}, \text{ValidTPM}, \text{ValidKey})$.

We also write TPM_i^n for the n -th instance of the TPM oracle initialized with the key pair $(\text{EPK}_i, \text{ESK}_i)$. Similarly, we write CA_j^m for the m -th instance of the CA oracle initialized with the key pair $(\text{CPK}_j, \text{CSK}_j)$. \mathcal{A} is given access to some TPM, CA, CorrCA and CorrTPM oracles. At the end of the execution, \mathcal{A} outputs a tuple $(\text{EPK}_{i^*}, \text{CPK}_{j^*}, \text{APK}^*, \text{CRE}^*)$ with $\text{EPK}_{i^*} \in \text{EPK}$, $\text{CPK}_{j^*} \in \text{cPK}$ and $\text{APK}^*, \text{CRE}^* \in \{0, 1\}^*$. The outcome of the experiment is determined as follows.

If CRE^* is not a valid signature on APK^* under CPK_{j^*} or the key CSK_{j^*} has been corrupt then the experiment returns 0. Otherwise, if c_{j^*} (associated with CPK_{j^*}) never generated the credential for the key APK^* so a tuple of the form $(\text{EPK}_{i^*}, \text{CPK}_{j^*}, \text{APK}^*, \text{CRE}^*)$ does not occur in RegList, or if $(\text{EPK}_{i^*}, \text{CPK}_{j^*}, \text{APK}^*, \text{CRE}^*)$ does occur in RegList but ValidKey($\text{EPK}_{i^*}, \text{APK}^*$) = 0 the experiment returns 1. If neither condition is satisfied, then the experiment returns 0.

Definition 5.2. *Unforgeability.* We define the advantage of the adversary \mathcal{A} in the unforgeability game as

$$\text{Adv}_{\text{eACAS}, \mathcal{A}}^{\text{unforge}}(\eta) = \left| \Pr[\text{Exp}_{\text{eACAS}, \mathcal{A}}^{\text{unforge}}(\eta) = 1] \right|,$$

and say that the eACAS protocol is *unforgeable* if $\text{Adv}_{\text{eACAS}, \mathcal{A}}^{\text{unforge}}(\eta)$ is a negligible function of η for all polynomial time adversaries \mathcal{A} .

5.5. Key binding

An eACAS protocol holds the key binding property if every CA's record $(\text{EPK}, \text{CPK}, \text{APK}/\text{CRE})$ is true, i.e. the EK and AK belong to the same TPM. Similar to unforgeability, the key binding experiment allows

the adversary \mathcal{A} to corrupt some TPMs (i.e. we consider the chosen corrupted TPM attack). In the game, \mathcal{A} is given the variables ValidKey and ValidTPM. The key binding experiment, as specified in Fig. 9, proceeds as follows. The variables ValidTPM $\subset \text{cPK} \times \text{EPK}$ and ValidKey $\subset \{\text{EPK}_1, \text{EPK}_2, \dots\} \times \{0, 1\}^*$ are specified by an authority Auth. We require that ValidKey satisfies that for any $\text{APK} \in \{0, 1\}^*$ ValidKey(EPK_i, APK) = 1 then ValidKey(EPK_k, APK) = 0 for any $k \neq i$. This reflects the idea that the attestation keys valid for the TPM are disjoint. \mathcal{A} is initially given $(\text{EPK}, \text{cPK}, \text{ValidTPM}, \text{ValidKey})$.

We also write TPM_i^n for the n -th instance of the TPM oracle initialized with the key pair $(\text{EPK}_i, \text{ESK}_i)$. Similarly, we write CA_j^m for the m -th instance of the CA oracle initialized with the key pair $(\text{CPK}_j, \text{CSK}_j)$. \mathcal{A} is given access to some TPM, CA, CorrCA and CorrTPM oracles. At the end of the execution, CA_{j^*} records a tuple $(\text{EPK}_{i^*}, \text{CPK}_{j^*}, \text{APK}^*, \text{CRE}^*)$ in confirmedList with $\text{EPK}_{i^*} \in \text{EPK}$, $\text{CPK}_{j^*} \in \text{cPK}$, $\text{APK}^*, \text{CRE}^* \in \{0, 1\}^*$ and CRE^* is a valid credential for APK^* created under CSK_{j^*} . The outcome of the experiment is determined as follows. If the key CSK_{j^*} has been corrupt then the experiment returns 0. Otherwise, if ValidKey($\text{EPK}_{i^*}, \text{APK}^*$) = 0 the experiment returns 1, else returns 0.

Definition 5.3. *Key binding.* We define the advantage of the adversary \mathcal{A} in the key binding game as

$$\text{Adv}_{\text{eACAS}, \mathcal{A}}^{\text{keybind}}(\eta) = \left| \Pr[\text{Exp}_{\text{eACAS}, \mathcal{A}}^{\text{keybind}}(\eta) = 1] \right|,$$

and say that the eACAS protocol holds *key binding* if $\text{Adv}_{\text{eACAS}, \mathcal{A}}^{\text{keybind}}(\eta)$ is a negligible function of η for all polynomial time adversaries \mathcal{A} .

5.6. Deniability

An eACAS protocol is deniable if a TPM can deny that it was engaged in a transaction with a CA even if this transaction actually took place. We define two kinds of deniability: stronger deniability and weaker deniability, both of which catch this security requirement. We say the scheme is *strongly/weakly deniable* if for each adversary \mathcal{A} that produces τ_0 (the output in interaction in real transaction), there exists a simulator \mathcal{S} that can produce τ_1 (the output in simulation) which is indistinguishable from τ_0 , i.e. no ppt algorithm \mathcal{D} can distinguish them with non-negligible probability. We consider the scenario that the TPMs and hosts are honest, but the CAs are malicious. The experiment of deniability is specified in Fig. 9. Let $x - \text{deni}$ be $s - \text{deni}$ (stronger deniability) or $w - \text{deni}$ (weaker deniability). The variables ValidTPM $\subset \text{cPK} \times \text{EPK}$ and ValidKey $\subset \{\text{EPK}_1, \text{EPK}_2, \dots\} \times \{0, 1\}^*$ are also specified by an authority Auth. We require that ValidKey satisfies that for any $\text{APK} \in \{0, 1\}^*$ ValidKey(EPK_i, APK) = 1 then ValidKey(EPK_k, APK) = 0 for any $k \neq i$. This reflects the idea that the attestation keys valid for the TPM are disjoint. The adversary \mathcal{A} and the simulator \mathcal{S} are given the same information $(\text{EPK}, \text{cPK}, \text{cSK}, \text{ValidTPM})$ initially. This reflects that the adversary can collude with all CAs.

If $x = s$ (i.e. stronger deniability), then the active adversary \mathcal{A} accesses some TPM^* oracles and outputs τ_0 (a transcript or an arbitrary string). We do not let \mathcal{A} access to the CA oracle since \mathcal{A} has been given cSK . Note that we assume the hosts are honest, thus \mathcal{A} cannot obtain the second stage output k from the original TPM oracle. \mathcal{A} is just given $(\text{APK}, \text{EPK}, \text{SCER})$. Therefore, in this experiment \mathcal{A} accesses a modified TPM^* oracle which is specified in Fig. 8. If $x = w$ (i.e. weaker deniability), then the passive adversary \mathcal{A} accesses the TRAN oracle and outputs τ_0 (a transcript or an arbitrary string). The simulator \mathcal{S} that can access all the secret signing keys of all CAs runs \mathcal{A} as a black-box and outputs τ_1

(a transcript or an arbitrary string). A distinguisher \mathcal{D} is given $\tau_b \in \{\tau_0, \tau_1\}$ and finally outputs a decision bit \hat{b} .

Definition 5.4. *Deniability.* We define the advantage of the adversary \mathcal{A} in the deniability game as

$$\text{Adv}_{\text{eACAS}, \mathcal{D}}^{x-\text{deni}-b}(\eta) = \left| \Pr[\text{Exp}_{\text{eACAS}, \mathcal{D}}^{x-\text{deni}-0}(\eta) = 1] - \Pr[\text{Exp}_{\text{eACAS}, \mathcal{D}}^{x-\text{deni}-1}(\eta) = 1] \right|,$$

and say that it is strongly ($x = s$) or weakly ($x = w$) deniable if $\text{Adv}_{\text{eACAS}, \mathcal{D}}^{x-\text{deni}-b}(\eta)$ is a negligible function of η for all polynomial time distinguishers \mathcal{D} .

6. SECURITY ANALYSIS

In this section, we proof that the proposed eACAS protocol holds the properties of third-party anonymity, unforgeability, key binding and strong deniability. Before analysing these properties, we first discuss the security of a TPM self-certification scheme.

Definition 6.1. A TPM self-certification scheme, written as $\mathbf{t.CER} = (\mathbf{t.KG}, \mathbf{t.SIG}, \mathbf{t.VER})$, uses a digital signature scheme, $\text{SIG} = (\text{KG}, \text{SIG}_{\text{sk}}, \text{VER}_{\text{pk}})$, to certify a key by a TPM. $\mathbf{t.CER}$ has a special feature that the signer TPM only certifies a key that is created by itself, and the variable $\text{ValidKey}(\text{certifiedKey}, \text{signingKey})$ is checked by both of the signer and verifier. A valid certificate indicates $\text{ValidKey}(\text{certifiedKey}, \text{signingKey}) = 1$.

Theorem 6.1. The $\mathbf{t.CER}$ scheme is EU-CMA (Existential Unforgeability under a Chosen Message Attack) secure if its underlying digital signature scheme SIG is EU-CMA secure.

We recall the definition of the EU-CMA security of a digital signature scheme given in [40]. The security of SIG is defined through game $\text{Exp}_{\text{SIG}, \mathcal{A}}^{\text{eucma}}(\eta)$, which is run between a simulator \mathcal{S} and an adversary \mathcal{A} .

In this game a pair of signing and verification keys (sk, pk) is generated by running the key generation algorithm KG . Then, \mathcal{A} is given the verification key pk and provided with access to a signing oracle $\text{SIG}_{\text{sk}}(\cdot)$. For each message m that \mathcal{A} sends to the oracle via \mathcal{S} , the oracle responds with a signature $\sigma = \text{SIG}_{\text{sk}}(m)$. Eventually, \mathcal{A} terminates its execution and outputs a message and signature pair (m^*, σ^*) . The experiment returns 1 if σ^* is a valid signature on m^* under pk , i.e. $\text{VER}_{\text{pk}}(\sigma^*, m^*) = \text{accept}$, and the message m^* was never queried to the signing oracle. The experiment returns 0 otherwise. The advantage of the adversary \mathcal{A} in breaking EU-CMA for the scheme SIG is defined as

$$\text{Adv}_{\text{SIG}, \mathcal{A}}^{\text{eucma}}(\eta) = \Pr[\text{Exp}_{\text{SIG}, \mathcal{A}}^{\text{eucma}}(\eta) = 1].$$

The scheme SIG is EU-CMA secure if for any probabilistic polynomial time adversary \mathcal{A} , its advantage $\text{Adv}_{\text{SIG}, \mathcal{A}}^{\text{eucma}}(\eta)$ is a negligible function.

We argue that Theorem 6.1 follows from a simple observation: assume a signer TPM in $\mathbf{t.CER}$ has at least two keys, one of them is used as a signing key. This key is also the signing key of SIG . Obviously, an adversary \mathcal{B} who has access to the public portions of the TPM keys and successfully breaks the EU-CMA security of

$\mathbf{t.CER}$ can be used by the adversary \mathcal{A} in the above $\text{Exp}_{\text{SIG}, \mathcal{A}}^{\text{eucma}}(\eta)$ game to generate a forgery of the scheme SIG .

Theorem 6.2. The eACAS protocol is third-party anonymous.

Proof. This property holds information-theoretically and does not rely on any security property of the underlying primitives. Let \mathcal{A} be an adversary against third-party anonymity of the eACAS protocol in the $\text{Exp}_{\text{eACAS}, \mathcal{A}}^{\text{anon}-b}(\eta)$, as shown in Fig. 9. $\text{Adv}_{\text{eACAS}, \mathcal{A}}^{\text{anon}-b}(\eta)$ is negligible with the following reasoning. After randomly selecting a bit b , the CH_b oracle assigns a freshly generated public key APK to the $\text{TMP } t_{i_b}$, activates CA and TPM oracles with $(\text{EPK}_{i_b}, \text{APK})$ and finally outputs a credential CRE . No matter what the identity of $\text{TPM } t_{i_b}$ is, the CA always creates the credential according to the following equation: $\text{CRE} = \text{SIG}_{\text{CSK}}(\text{APK})$ without any inference of the identity or EPK_{i_b} of t_{i_b} . This credential CRE on the uniformly chosen APK is exactly the output of the CH_b oracle. Because \mathcal{A} does not have access to the transaction on the protocol running and obtains no information about the selected TPM from the outputting pair (CRE, APK) , therefore, \mathcal{A} always outputs a decision bit b by guessing. It is essential that the outputs of $\text{Exp}_{\text{eACAS}, \mathcal{A}}^{\text{anon}-0}(\eta)$ and $\text{Exp}_{\text{eACAS}, \mathcal{A}}^{\text{anon}-1}(\eta)$ have the same probability distribution, as long as that the TPM attestation keys are selected from the same distribution. So for any ppt adversary \mathcal{A} , the advantage

$$\text{Adv}_{\text{eACAS}, \mathcal{A}}^{\text{anon}}(\eta) = \left| \Pr[\text{Exp}_{\text{eACAS}, \mathcal{A}}^{\text{anon}-0}(\eta) = 1] - \Pr[\text{Exp}_{\text{eACAS}, \mathcal{A}}^{\text{anon}-1}(\eta) = 1] \right|$$

is obviously a negligible function, i.e. the eACAS protocol is third-party anonymous. ■

Theorem 6.3. The eACAS protocol is unforgeable if the digital signature schemes SIG_{CK} is EU-CMA secure, the asymmetric encryption scheme aENC_{EK} is CPA secure, the TPM self-certification scheme $\mathbf{t.CER}_{\text{AK}}$ is EU-CMA secure and the key derivation function KDF is a random oracle.

Proof. The adversary \mathcal{A} that works in the $\text{Exp}_{\mathcal{A}}^{\text{unforge}}$ as shown in Fig. 9 wins the unforgeability game if \mathcal{A} can produce a valid credential CRE^* on some public key APK^* in one of the following three cases:

- Case1 $(\text{EPK}_{i^*}, \text{CPK}_{j^*}, \text{APK}^*/\text{CRE}^*) \notin \text{RegList}$. This case indicates that APK^* has not been involved in an execution with the $\text{CA } c_j$, and \mathcal{A} has created a forgery, $\text{CRE}^* = \text{SIG}_{\text{CSK}_{j^*}}(\text{APK}^*)$. This is contradict to the assumption that $\text{SIG}_{\text{CK}} = (\text{C.KG}, \text{SIG}_{\text{CSK}}, \text{VER}_{\text{CPK}})$ is EU-CMA secure.
- Case2 $(\text{EPK}_{i^*}, \text{APK}^*, \text{SCER}^*) \in \text{scerList} \wedge \text{ValidKey}(\text{EPK}_{i^*}, \text{APK}^*) = 0$. This case indicates that \mathcal{A} has successfully created a forgery, $\mathbf{t.SIG}_{\text{ASK}^*}(\text{EPK}_{i^*})$ without corrupting t_{i^*} . This is contradict to the assumption that $\mathbf{t.CER}_{\text{AK}}$ is EU-CMA secure.

We now prove that for any adversary \mathcal{A} against the eACAS protocol, there exists adversaries \mathcal{B} and \mathcal{D} against their individual scheme, associated with each case, such that:

$$\text{Adv}_{\text{ePCAS}, \mathcal{A}}^{\text{unforge}}(\eta) \leq \frac{1}{p(\eta)} \cdot \text{Adv}_{\text{SIG}, \mathcal{B}}^{\text{eucma}}(\eta) + \frac{1}{p'(\eta)} \text{Adv}_{\text{SIG}, \mathcal{D}}^{\text{eucma}}(\eta) \quad (1)$$

Fix an adversary \mathcal{A} for the experiment $\text{Exp}_{\text{eACAS}, \mathcal{A}}^{\text{unforge}}(\eta)$. Let Case_1 and Case_2 be the events of Cases 1 and 2, respectively, then we

have that

$$\begin{aligned} \Pr[\text{Exp}_{\text{ePCAS}, \mathcal{A}}^{\text{unforge}}(\eta) = 1] &= \\ \Pr[\text{Exp}_{\text{eACAS}, \mathcal{A}}^{\text{unforge}}(\eta) = 1 \wedge \text{Case_1}] &+ \\ \Pr[\text{Exp}_{\text{eACAS}, \mathcal{A}}^{\text{unforge}}(\eta) = 1 \wedge \text{Case_2}] & \end{aligned}$$

Next, we upper bound each of the terms above. In Case_1, by the notion of EU-CMA security in [40], the adversary \mathcal{B} (for $\text{Exp}_{\text{SIG}, \mathcal{B}}^{\text{eucma}}$) has as input some verification key pk and access to a signing oracle under the corresponding secret key sk . \mathcal{B} runs the adversary \mathcal{A} internally and simulates for \mathcal{A} all of the oracles to which \mathcal{A} has access in $\text{Exp}_{\text{eACAS}, \mathcal{A}}^{\text{unforge}}(\eta)$. The keys for all of the TPM and CorrTPM oracles are obtained by running $(\text{EPK}_i, \text{ESK}_i) \leftarrow \text{E.KG}(\eta)$ (for all $1 \leq l \leq p(\eta)$). The keys for all but one randomly selected CAs are obtained using $(\text{CSK}_j, \text{CPK}_j) \leftarrow \text{C.KG}(\eta)$ (for all $1 \leq j \neq j^* \leq p(\eta)$, where j^* is selected uniformly at random from $1, 2, \dots, p(\eta)$). The public key of CA_{j^*} , CPK_{j^*} , is set to be pk . \mathcal{B} can therefore simulate perfectly all of the oracles: for TPM_i^l oracles \mathcal{B} executes the code defined by TPM (Fig. 8). For any oracle CA_j^m with $j \neq j^*$ \mathcal{B} executes the code defined by CA (Fig. 8). For all oracles CA_j^m \mathcal{B} executes the same code with the difference that any signature σ that such an oracle needs to produce on some message m is obtained from the signing oracle to which \mathcal{B} has access. \mathcal{B} can also answer corruption queries for all of the CorrTPMs and CorrCAs except for CA_{j^*} ; if \mathcal{A} ever queries j^* to his CorrCA oracle, then \mathcal{B} aborts its execution. Provided that j^* is not submitted by \mathcal{A} to his decryption oracle, the simulation that \mathcal{B} offers to \mathcal{A} is perfect. When \mathcal{A} produces his forgery $(\text{CPK}^*, \text{APK}^*, \text{CRE}^*)$, if $\text{CPK}^* \neq pk$ then \mathcal{B} aborts; otherwise \mathcal{B} outputs $(\text{APK}^*, \text{CRE}^*)$ as his forgery.

Since the simulation is perfect (if \mathcal{B} does not abort), CPK^* in the output of \mathcal{A} equals CPK_{j^*} (i.e. pk) with the probability $\frac{1}{p(\eta)}$. It remains to determine when the forgery that is output by \mathcal{B} is valid (if the guess j^* is correct, that is, it does correspond to the forgery). This is the case when the message/signature pair $(\text{APK}^*, \text{CRE}^*)$ is valid under sk , the signature CRE^* had not been produced by the oracle to which \mathcal{B} has access in response to a query APK^* by \mathcal{B} , and \mathcal{B} did not abort due to a CorrCA query. Whenever \mathcal{A} wins the triple $(\text{CPK}_{j^*}, \text{APK}^*, \text{CRE}^*)$ is such that CRE^* had not been produced by some signing oracle as a signature on APK^* , since otherwise, a tuple of the form $(\text{EPK}_i, \text{CPK}_{j^*}, \text{APK}^*, \text{CRE}^*)$ would appear in `confirmedList`, which contains all tuples $(\text{EPK}, m, \text{APK}, \text{CRE})$, where pk is the verification key and CRE had been obtained from the signing oracle of \mathcal{B} . In this case, j^* had also not been queried by \mathcal{A} to its corruption oracle (as otherwise the output of \mathcal{A} would not lead to a successful break), so \mathcal{B} does not abort.

To conclude, when \mathcal{A} produces his output $(\text{CPK}^*, \text{APK}^*, \text{CRE}^*)$, with probability $\frac{1}{p(\eta)}$ the forgery is with respect to the public key of CA_{j^*} , pk . Furthermore, if this event occurs in the simulation of $\text{Exp}_{\text{ePCAS}, \mathcal{A}}^{\text{unforge}}(\eta)$, then $(*, pk, \text{APK}^*, \text{CRE}^*)$ does not appear in `RegList`. According to the observation above CRE^* has not been obtained by sending APK^* to the signing oracle under sk , hence $(\text{APK}^*, \text{CRE}^*)$ is a successful forgery for $\text{Exp}_{\text{SIG}, \mathcal{B}}^{\text{eucma}}(\eta)$. We therefore obtain that

$$\begin{aligned} \Pr[\text{Exp}_{\text{ePCAS}, \mathcal{A}}^{\text{unforge}}(\eta) = 1 \wedge \text{Case_1}] &\leq \\ \frac{1}{p(\eta)} \cdot \Pr[\text{Exp}_{\text{SIG}, \mathcal{B}}^{\text{eucma}}(\eta) = 1]. & \end{aligned}$$

In Case_2, if the adversary obtains a credential blob cre_b from the CA oracle under EPK_i , \mathcal{A} can access the CorrTPM

oracle to obtain ESK_i and consequently get a certification cre^* on APK^* . However, to access the CA oracle, a tuple of the form $(\text{EPK}_i, \text{APK}^*, \text{SCER}^*)$ should exist. Since the TPM oracle first checks whether $\text{ValidKey}(\text{EPK}_i, \text{APK}^*) = 1$ and then returns SCER^* if it holds, this contradicts that $\text{ValidKey}(\text{EPK}_i, \text{APK}^*) = 0$. Thus, \mathcal{A} cannot get such tuple by querying the TPM oracle. The only way to send such tuple is to produce it on his own. Therefore, if event Case_2 occurs, then \mathcal{A} must have produced a valid self-certificate which the CA oracle accepted, hence \mathcal{A} forged a signature on EPK_i under ASK^* which is corresponding to APK^* and $\text{ValidKey}(\text{EPK}_i, \text{APK}^*) = 0$. This intuition is behind the construction of the following EU-CMA adversary \mathcal{D} for $\text{Exp}_{\text{SIG}, \mathcal{D}}^{\text{eucma}}(\eta)$ out of an adversary \mathcal{A} for $\text{Exp}_{\text{eACAS}, \mathcal{A}}^{\text{unforge}}(\eta)$.

\mathcal{D} has as input some verification key pk and access to a signing oracle under the corresponding secret key sk . \mathcal{D} runs \mathcal{A} internally and simulates for \mathcal{A} all of the oracles to which \mathcal{A} has access in $\text{Exp}_{\text{eACAS}, \mathcal{A}}^{\text{unforge}}(\eta)$. The keys for all of the TPMs are obtained from $(\text{EPK}_i, \text{ESK}_i) \leftarrow \text{E.KG}(\eta)$ (for all $1 \leq l \leq p(\eta)$). The keys for all CAs are obtained from $(\text{CSK}_j, \text{CPK}_j) \leftarrow \text{C.KG}(\eta)$ (for all $1 \leq j \leq p(\eta)$). The attestation keys for all but one are obtained from $(\text{APK}^*, \text{ASK}^*) \leftarrow \text{A.KG}(\eta)$ and the public portion of the special one is set to be the verification key pk that \mathcal{D} has as input. Assume there are $p'(\eta)$ attestation public keys for the $p(\eta)$ TPMs.

\mathcal{D} can therefore simulate perfectly all of the oracles: for TPM_i^l oracles \mathcal{D} executes the code defined by TPM (Fig. 8). For any oracle CA_j^m \mathcal{D} executes the code defined by CA (Fig. 8). \mathcal{D} can also answer all of the CorrTPM and CorrCA oracles, the simulation that \mathcal{D} offers to \mathcal{A} is therefore perfect. If \mathcal{A} with corrupted ESK_{i^*} sends $(\text{APK}^*, \text{EPK}_{i^*}, \text{SCER}^*)$ to the CA oracle such that $1 \leftarrow \text{VER}(\text{APK}^*, \text{EPK}_{i^*}, \text{SCER}^*)$, then the CA oracle will generate CRE^* and then append $(\text{EPK}_{i^*}, \text{CPK}_{j^*}, \text{APK}^*, \text{CRE}^*)$ to `confirmedList`. In this case, \mathcal{A} with ESK_{i^*} can obtain CRE^* and then consequently win the game. When \mathcal{A} outputs $(\text{EPK}_{i^*}, \text{CPK}_{j^*}, \text{APK}^*, \text{CRE}^*)$, if $\text{APK}^* \neq pk$ then \mathcal{D} aborts; otherwise \mathcal{D} outputs $(\text{APK}^*, \text{SCER}^*)$ as his forgery. If \mathcal{D} does not abort, APK^* in the output of \mathcal{A} equals pk with probability $\frac{1}{p'(\eta)}$. When the event Case_2 occurs and \mathcal{A} wins, the message/signature pair $(\text{EPK}^*, \text{SCER}^*)$ should be valid under pk . According to the observation above SCER^* has not been obtained by sending pk^* to the signing oracle under sk , hence $(\text{APK}^*, \text{SCER}^*)$ is a successful forgery for $\text{Exp}_{\text{SIG}, \mathcal{D}}^{\text{eucma}}(\eta)$. We therefore obtain that

$$\begin{aligned} \Pr[\text{Exp}_{\text{eACAS}, \mathcal{A}}^{\text{unforge}}(\eta) = 1 \wedge \text{Case_2}] &\leq \\ \frac{1}{p'(\eta)} \cdot \Pr[\text{Exp}_{\text{SIG}, \mathcal{D}}^{\text{eucma}}(\eta) = 1]. & \end{aligned}$$

Together with the other bounds, we obtain the Equation 1. ■

Theorem 6.4. The eACAS protocol holds key binding if the asymmetric encryption scheme aENC_{EK} is CPA secure, the TPM self-certification scheme t.CER_{AK} is EU-CMA secure and the key derivation function KDF is a random oracle.

Proof. The adversary \mathcal{A} that works in the $\text{Exp}_{\mathcal{A}}^{\text{keybind}}$ as shown in Fig. 9 wins the key binding game if \mathcal{A} can make the simulator \mathcal{S} who simulates an uncorrupted CA (c_j) accept a binding record between an endorsement key EK_i and an attestation key AK^* (i.e. $(\text{EPK}_i, \text{CPK}_{j^*}, \text{APK}^*/\text{CRE}^*) \in \text{confirmedList}$) but these two keys do not belong to the same TPM TPM_i , which is also uncorrupted, (i.e. $\text{ValidKey}(\text{EPK}_i, \text{APK}^*) = 0$). Two cryptographic mechanisms are used

by the TPM: one is self-certifying EPK_i under AK^* and another is using EPK_i in $TPM2_ActivateCredential$ to retrieve the credential CRE^* for AK^* . We will analyse the contributions of these two mechanisms in the following two separated cases:

- Case1 \mathcal{A} does not own EPK_i . This case indicates that \mathcal{A} can manage to find the plaintext s from $aENC_{EPK_i}(s)$ without knowing EPK_i . This is contradict to the assumption that $aENC_{EK} = (E.KG, aENC_{EPK}, aDEC_{ESK})$ is CPA (Chosen Plaintext Attack) secure.
- Case2 \mathcal{A} does not own AK^* . This case indicates that \mathcal{A} can successfully create a forgery, $t.SIG_{ASK^*}(EPK_i)$ without knowing ASK^* . This is contradict to the assumption that $t.CER_{AK}$ is EU-CMA secure.

We now prove that for any adversary \mathcal{A} against the eACAS protocol, there exists adversaries \mathcal{C} and \mathcal{D} against their individual scheme, associated with each case, such that

$$\text{Adv}_{ePCAS, \mathcal{A}}^{\text{keybind}}(\eta) \leq \frac{1}{p(\eta)} \cdot \text{Adv}_{aENC, \mathcal{C}}^{\text{CPA}} + \frac{1}{p'(\eta)} \text{Adv}_{SIG, \mathcal{D}}^{\text{eucma}} \quad (2)$$

Fix an adversary \mathcal{A} for the experiment $\text{Exp}_{eACAS, \mathcal{A}}^{\text{keybind}}(\eta)$. Let Case_1 and Case_2 be the events of Cases 1 and 2, respectively, then we have that

$$\begin{aligned} \Pr[\text{Exp}_{ePCAS, \mathcal{A}}^{\text{keybind}}(\eta) = 1] = \\ \Pr[\text{Exp}_{eACAS, \mathcal{A}}^{\text{keybind}}(\eta) = 1 \wedge \text{Case}_1] + \\ \Pr[\text{Exp}_{eACAS, \mathcal{A}}^{\text{keybind}}(\eta) = 1 \wedge \text{Case}_2] \end{aligned}$$

Next, we upper bound each of the terms above.

In Case_1, there exists some oracle $\mathcal{O} = \text{CA}$ such that $(CPK_j^*, EPK_i, AK^*, p, SCER^*)$ is the message received by \mathcal{O} . The oracle checks the attributes of AK^* and validation of $SCER^*$ associated with EPK_i , outputs a credential CRE to AK , and then creates/returns the credential blob $CRE.b = (CRE_t.b, CRE_h.b)$ using the $aENC_{EK}$ scheme, which is actually a KEM-DEM scheme under the key EPK_i . $CRE_t.b$ is the KEM cipher and $CRE_h.b$ is the DEM cipher. If \mathcal{A} accesses the TPM oracle to obtain the encryption key from $CRE_t.b$, it will not succeed, since the TPM oracle first checks whether $\text{ValidKey}(EPK_i, APK^*) = 1$ and this contradicts that $\text{ValidKey}(EPK_i, APK^*) = 0$. Thus, \mathcal{A} cannot get such tuple $(EPK_i, CPK_j^*, APK^*, CRE^*)$ by querying the TPM oracle. The only way to send such tuple is to decrypt the KEM cipher on his own. Therefore, if event Case_1 occurs, then \mathcal{A} must have performed decryption successfully to retrieve CRE^* which the CA oracle accepted, hence \mathcal{A} broke the $aENC_{EK}$ scheme, which is corresponding to EPK_i and $\text{ValidKey}(EPK_i, APK^*) = 0$. This intuition is behind the construction of the following CPA (Chosen-Plaintext Attack) security adversary \mathcal{C} for $\text{Exp}_{aENC, \mathcal{C}}^{\text{CPA}}(\eta)$ out of an adversary \mathcal{A} for $\text{Exp}_{eACAS, \mathcal{A}}^{\text{keybind}}(\eta)$.

The adversary \mathcal{C} is for $\text{Exp}_{aENC, \mathcal{C}}^{\text{CPA}}$, and by the notion of CPA secure encryption schemes [41], \mathcal{C} has as input some encryption key pk and access to a decryption oracle under the corresponding secret key sk . \mathcal{C} will output two selected plaintexts m_0 and m_1 to the simulator of $\text{Exp}_{aENC, \mathcal{C}}^{\text{CPA}}$ to obtain a ciphertext $c = aENC_{pk}(m_b)$, which is the same as \hat{s} (part of $CRE.b$) in the CA oracle, and then output a correct guessed value \hat{b} . \mathcal{C} runs \mathcal{A} internally and simulates for \mathcal{A} all of the oracles to which \mathcal{A} has access in $\text{Exp}_{eACAS, \mathcal{A}}^{\text{keybind}}(\eta)$. The keys for all of the TPMs are obtained from $(EPK_i, ESK_i) \leftarrow E.KG(\eta)$ (for all $1 \leq i \leq p(\eta)$), where the special key for i is set as (pk, sk) that \mathcal{C} has as input. The keys for all CAs are obtained from $(CSK_j, CPK_j) \leftarrow C.KG(\eta)$ (for all $1 \leq j \leq p(\eta)$). The attestation keys are

obtained from $(APK^*, ASK^*) \leftarrow A.KG(\eta)$. Assume there are $p(\eta)$ EKS for the $p(\eta)$ TPMs.

\mathcal{C} can therefore simulate perfectly all of the oracles: for TPM_i^n oracles \mathcal{C} executes the code defined by TPM (Fig. 8), except for TPM_i . For any oracle CA_j^m \mathcal{C} executes the code defined by CA (Fig. 8). \mathcal{C} can also answer corruption queries for all of the CorrTPM and CorrCA oracles, but it will be abort if \mathcal{A} wants to corrupt TPM_i . If \mathcal{A} with uncorrupted ESK_i sends a request with APK^* to the CA oracle then the oracle will generate CRE^* . In this case, \mathcal{C} follows the CA oracle, except using $c = aENC_{pk}(m_b)$ as \hat{s} . \mathcal{C} also maintains the KDF as a random oracle, and uses randomly selected keys for KE and KM . If \mathcal{A} wins the game, then \mathcal{C} must have the entry $(m_b, 'STORAGE', AK, \eta)$ to KDF. From this, \mathcal{C} knows the right value of b for $\text{Exp}_{aENC, \mathcal{C}}^{\text{CPA}}$.

If \mathcal{C} does not abort, the simulation that \mathcal{C} provides to \mathcal{A} is perfect. EPK_i in the output of \mathcal{A} equals pk with the probability $\frac{1}{p(\eta)}$. When the event $\Pr[\text{Case}_1]$ occurs and the adversary \mathcal{A} wins, \mathcal{C} can break the CPA security of $aENC_{EK}$ under the random oracle model. We therefore obtain that

$$\Pr[\text{Exp}_{eACAS, \mathcal{A}}^{\text{keybind}}(\eta) = 1 \wedge \text{Case}_1] \leq$$

$$\frac{1}{p(\eta)} \cdot \Pr[\text{Exp}_{aENC, \mathcal{C}}^{\text{CPA}}(\eta) = 1].$$

In Case_2, if the adversary obtains a credential blob $CRE.b$ from the CA oracle under EPK_i , \mathcal{A} can access the CorrTPM oracle to obtain ESK_i and consequently get a certification CRE^* on APK^* . However, to access the CA oracle, a tuple of the form $(EPK_i, APK^*, SCER^*)$ should exist. Since the TPM oracle first checks whether $\text{ValidKey}(EPK_i, APK^*) = 1$ and then returns $SCER^*$ if it holds, this contradicts that $\text{ValidKey}(EPK_i, APK^*) = 0$. Thus, \mathcal{A} cannot get such tuple by querying the TPM oracle. The only way to send such tuple is to produce it on his own. Therefore, if event Case_2 occurs, then \mathcal{A} must have produced a valid self-certificate which the CA oracle accepted, hence \mathcal{A} forged a signature on EPK_i under ASK^* which is corresponding to APK^* and $\text{ValidKey}(EPK_i, APK^*) = 0$. This intuition is behind the construction of the following EU-CMA adversary \mathcal{D} for $\text{Exp}_{SIG, \mathcal{D}}^{\text{eucma}}(\eta)$ out of an adversary \mathcal{A} for $\text{Exp}_{eACAS, \mathcal{A}}^{\text{keybind}}(\eta)$.

\mathcal{D} has as input some verification key pk and access to a signing oracle under the corresponding secret key sk . \mathcal{D} runs \mathcal{A} internally and simulates for \mathcal{A} all of the oracles to which \mathcal{A} has access in $\text{Exp}_{eACAS, \mathcal{A}}^{\text{keybind}}(\eta)$. The keys for all of the TPMs are obtained from $(EPK_i, ESK_i) \leftarrow E.KG(\eta)$ (for all $1 \leq i \leq p(\eta)$). The keys for all CAs are obtained from $(CSK_j, CPK_j) \leftarrow C.KG(\eta)$ (for all $1 \leq j \leq p(\eta)$). The attestation keys for all but one are obtained from $(APK^*, ASK^*) \leftarrow A.KG(\eta)$ and the public portion of the special one is set to be the verification key pk that \mathcal{D} has as input. Assume there are $p'(\eta)$ attestation public keys for the $p(\eta)$ TPMs.

\mathcal{D} can therefore simulate perfectly all of the oracles: for TPM_i^n oracles \mathcal{D} executes the code defined by TPM (Fig. 8). For any oracle CA_j^m \mathcal{D} executes the code defined by CA (Fig. 8). \mathcal{D} can also answer all of the CorrTPM and CorrCA oracles, the simulation that \mathcal{D} offers to \mathcal{A} is therefore perfect. If \mathcal{A} with corrupted ESK_i sends $(APK^*, EPK_i, SCER^*)$ to the CA oracle such that $1 \leftarrow \text{VER}(APK^*, EPK_i, SCER^*)$, then the CA oracle will generate CRE^* and then append $(EPK_i, CPK_j^*, APK^*, CRE^*)$ to confirmedList. In this case, \mathcal{A} with ESK_i can obtain CRE^* and then consequently win the game. When \mathcal{A} outputs $(EPK_i, CPK_j^*, APK^*, CRE^*)$, if $APK^* \neq pk$ then \mathcal{D} aborts; otherwise \mathcal{D} outputs $(APK^*, SCER^*)$ as his forgery. If \mathcal{D} does not abort, APK^* in the output of \mathcal{A} equals pk with probability $\frac{1}{p'(\eta)}$. When the event Case_2 occurs and \mathcal{A} wins, the message/signature pair $(EPK_i, SCER^*)$ should be valid under pk . According to the observation

$\text{TPM}_S^*(\text{EPK}_i)$
 $(\text{APK}, \text{ASK}) \leftarrow \text{KG}(\eta)$
 $\text{SCER} = \text{TPM2_Certify}(\text{ASK}, \text{EPK}_i)$
 send SCER

 $\mathcal{S}(\text{EPK}, \text{CPK}, \text{CSK}, \text{ValidTPM})$
 run \mathcal{A} and answer \mathcal{A} 's queries according to
 $\text{TPM}_S^*(\text{EPK}_i)$
 $\tau_S \leftarrow \mathcal{A}^{\text{TPM}_S^*}(\text{EPK}, \text{CPK}, \text{CSK}, \text{ValidTPM})$
 $\tau_1 \leftarrow \tau_S$
 output τ_1

Figure 10. The TPM_S^* oracle and simulator in the deniability game

above SCER^* has not been obtained by sending pk^* to the signing oracle under sk , hence $(\text{EPK}_i^*, \text{SCER}^*)$ is a successful forgery for $\text{Exp}_{\text{SIG}, \mathcal{D}}^{\text{eucma}}(\eta)$. We therefore obtain that

$$\Pr[\text{Exp}_{\text{eACAS}, \mathcal{A}}^{\text{keybind}}(\eta) = 1 \wedge \text{Case_2}] \leq \frac{1}{p'(\eta)} \cdot \Pr[\text{Exp}_{\text{SIG}, \mathcal{D}}^{\text{eucma}}(\eta) = 1].$$

Together with the Case_1 bound, we obtain the Equation 2. ■

Theorem 6.5. The eACAS protocol satisfies strong deniability.

Proof. The property of strong deniability is held in the proposed eACAS protocol also information-theoretically without relying on any security property of the underlying primitives. Let the simulator \mathcal{S} run the game with the adversary \mathcal{A} and simulate the TPM_S^* oracle, as shown in Fig. 10. When \mathcal{A} queries this oracle with the input EPK_i , \mathcal{S} generates a new attestation key pair (APK, ASK) , creates a self-certificate SCER on EPK_i under ASK , using the same operation as the TPM2_Certify command, and outputs SCER . From \mathcal{A} 's point of view, the output of the simulated TPM_S^* and SCER from the TPM oracle are indistinguishable since we assume that \mathcal{A} doesn't have access to ValidKey . Once \mathcal{A} outputs a transcript τ_S , \mathcal{S} uses it as its output τ_1 . Since we have perfect simulation, τ_1 is indistinguishable from \mathcal{A} 's output τ_0 in interaction with a real TPM^* oracle. Therefore, $\text{Adv}_{\mathcal{A}}^{\text{deni}}(\eta)$ is a negligible function of η for all polynomial time distinguishers \mathcal{D} . ■

7. IMPLEMENTATION

The proposed eACAS protocol for the conventional signature case eACAS_C and the DAA signature case eACAS_D has been tested using an Infineon TPM 2.0 module (Infineon SLB9760) installed on a Raspberry Pi 3 (ARMv7) running Raspbian Linux 4.14.30. The code was written in C++ and uses the IBM TSS (TPM Software Stack), OpenSSL (version 1.1.0f) and the Apache-Milagro Crypto library (version 3 of this library is used for the pairing functions required in the ECDA scheme [28]). We use the GNU g++ compiler and linker, version 6.3.0 to compile and link the code.

As the purpose of this experiment is to test the protocol in the TPM environment the focus was on making the code clear and robust, we have not yet carried out any optimization. In our

Table 3. Timings in the eACAS Experiment (Each number is in ms)

Operations	eACAS _C	eACAS _D
TPM2_CreatePrimary (2048-bit RSA key [42])	18 900	18 900
TPM2_Create (256-bit ECDSA key [43])	217	-
TPM2_Create (256-bit ECDA key [28])	-	215
TPM2_Load	38.2	38.2
TPM2_Commit	-	91
TPM2_Certify	86	59.5
TPM2_ActivateCredential	220	220
Compute a pairing	-	171
Calculate an AK credential	1.2	47.2

tests we have no network latency and so the time taken to run the protocol is dominated by the TPM calls and computing the pairings. The timings from our experiment are shown in Table 3. There is some variability in the values and so the results given in the table are the median values. The Host and CA operations are implemented on the Raspberry Pi and most of them are very fast compared with the operations on the TPM. We give two of the slower examples in the table: calculating the pairings and creating the credential of an attestation key. Calculating the credential in the eACAS_C case is a single call to OpenSSL to calculate an ECDSA signature. This is much faster than the calculation for the eACAS_D case which generates an ECDA key credential and involves many separate operations.

8. CONCLUSIONS

The attestation service is one of the most important service provided by TPMs, which has subtle requirements on security, privacy and trust. The question on how to bind a TPM's attestation keys with its endorsement key has not been properly answered in the literature. This paper has investigated the existing solutions from the TPM specifications, the DAA papers and relevant international standards and shown that none of them are satisfactory, as they either do not hold required security properties or cannot be implemented using a real TPM chip. This paper has answered this question by proposing a novel attestation CA solution that has been proved secure and implemented using a TPM 2.0 chip. Our solution does not need any modification to the current TPM 2.0 command interface and has good performance with the best aimed security.

ACKNOWLEDGMENTS

This work is supported by the European Union's Horizon research and innovation program under grant agreement numbers: 779391 (FutureTPM), 952697 (ASSURED), 101019645 (SECANT), 101069688 (CONNECT), and 101070627 (REWIRE). These projects are funded by the UK government Horizon Europe guarantee and administered by UKRI.

DATA AVAILABILITY STATEMENT

The data used to support the findings of this study are included within the article.

REFERENCES

1. TCG. TPM 2.0 Library Specification. <https://trustedcomputinggroup.org/resource/tpm-library-specification/>. Accessed: 2022-03-02.

2. TCG. TPM 1.2 Specification. <https://trustedcomputinggroup.org/resource/tpm-main-specification/>. Accessed: 2022-04-02.
3. TCG. About TCG. <https://trustedcomputinggroup.org/about/>. Accessed: 2022-04-02.
4. Microsoft. How Windows 10 uses the Trusted Platform Module. <https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/how-windows-uses-the-tpm>. Accessed: 2022-03-25.
5. Microsoft. Minimum hardware requirements. <https://docs.microsoft.com/en-gb/windows-hardware/design/minimum/minimum-hardware-requirements-overview>. Accessed: 2022-04-25.
6. Arthur, W., Challener, D. and Goldman, K. (2015) A practical guide to TPM 2.0: Using the new trusted platform module in the new age of security. Springer Nature.
7. FIDO. Simpler, Stronger Authentication. Solving the World's Password Problem. <https://fidoalliance.org/>. Accessed: 2022-04-02.
8. Greveler, U., Justus, B. and Loehr, D. (2011) Direct anonymous attestation: enhancing cloud service user privacy. In *OTM Confederated International Conferences' On the Move to Meaningful Internet Systems'*, pp. 577–587. Springer.
9. L. Lorenzin and A. Shah. Trusted network communications. <http://trustedcomputinggroup.org/work-groups/trusted-network-communications/>. Accessed: 2021-03-01.
10. TCG. TPM from PCS to the IoT. <https://trustedcomputinggroup.org/tpm-pcs-iot/>, mar 2017. Accessed: 2022-02-02.
11. TCG. Trusted network connect (TNC) howto. <https://wiki.strongswan.org/projects/1/wiki/trustednetworkconnect>, 2018. Accessed: 2021-06-02.
12. Whitefield, J., Chen, L., Giannetsos, T., Schneider, S. and Treharne, H. (2017) Privacy-enhanced capabilities for vanets using direct anonymous attestation. In *In 2017 IEEE Vehicular Networking Conference (VNC)*, pp. 123–130. IEEE.
13. E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145, 2004.
14. Proudler, G., Chen, L. and Dalton, C. (2014) *Trusted Computing Platforms*. Springer.
15. E. Brickell and J. Li. Enhanced privacy id: A direct anonymous attestation scheme with enhanced revocation capabilities. In *Proceedings of the 2007 ACM workshop on Privacy in electronic society*, pages 21–30, 2007.
16. Brickell, E. and Li, J. (2009) Enhanced privacy id from bilinear pairing. *Cryptology ePrint Archive*.
17. Lowe, G. (1996) Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1055, pp. 147–166. Springer.
18. Needham, R.M. and Schroeder, M.D. (1978) Using encryption for authentication in large networks of computers. *Commun. ACM*, **21**, 993–999.
19. ISO/IEC 20008-2:2013 *Information technology – Security techniques – Anonymous digital signatures – Part 2: Mechanisms using a group public key*.
20. Chen, L., Lee, M.-F. and Warinschi, B. (2011) Security of the enhanced tgc privacy-ca solution. In *International Symposium on Trustworthy Global Computing*, pp. 121–141. Springer.
21. Chen, L. and Warinschi, B. (2010) Security of the tgc privacy-ca solution. In *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pp. 609–616. IEEE.
22. Dai, Y., Zhang, F. and Zhao, C.-A. (2022) Fast hashing to g_2 in direct anonymous attestation. *Cryptology ePrint Archive*.
23. Brickell, E., Chen, L. and Li, J. (2008) A new direct anonymous attestation scheme from bilinear maps. In *International Conference on Trusted Computing*, pp. 166–178. Springer.
24. Brickell, E., Chen, L. and Li, J. (2009) Simplified security notions of direct anonymous attestation and a concrete scheme from pairings. *Int J. Inf. Secur.*, **8**, 315–330.
25. Brickell, E. and Li, J. (2010) A pairing-based daa scheme further reducing tpm resources. In *International Conference on Trust and Trustworthy Computing*, pp. 181–195. Springer.
26. Chen, L. (2010) A daa scheme requiring less tpm resources. In *International Conference on Information Security and Cryptology*, pp. 350–365. Springer.
27. Chen, L., Morrissey, P. and Smart, N.P. (2009) Daa: fixing the pairing based protocols. *Cryptology ePrint Archive*.
28. Chen, L., Page, D. and Smart, N.P. (2010) On the design and implementation of an efficient daa scheme. In *International Conference on Smart Card Research and Advanced Applications*, pp. 223–237. Springer.
29. S. Wesemeyer, C. J. Newton, H. Treharne, L. Chen, R. Sasse, and J. Whitefield. Formal analysis and implementation of a tpm 2.0-based direct anonymous attestation scheme. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 784–798, 2020.
30. B. Larsen, T. Giannetsos, I. Krontiris, and K. Goldman. Direct anonymous attestation on the road: Efficient and privacy-preserving revocation in c-its. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 48–59, 2021.
31. Chen, L., Morrissey, P. and Smart, N.P. (2008) Pairings in trusted computing. In *International Conference on Pairing-Based Cryptography*, pp. 1–17. Springer.
32. Chen, L., Morrissey, P. and Smart, N.P. (2008) On proofs of security for daa schemes. In *International Conference on Provable Security*, pp. 156–175. Springer.
33. Bernhard, D., Fuchsbaue, G., Ghadafi, E., Smart, N.P. and Warinschi, B. (2013) Anonymous attestation with user-controlled linkability. *Int. J. Inf. Secur.*, **12**, 219–249.
34. Camenisch, J., Drijvers, M. and Lehmann, A. (2016) Universally composable direct anonymous attestation. In *Public-Key Cryptography–PKC 2016*, pp. 234–264. Springer.
35. Camenisch, J., Chen, L., Drijvers, M., Lehmann, A., Novick, D. and Urian, R. (2017) One tpm to bind them all: Fixing tpm 2.0 for provably secure anonymous attestation. In *In 2017 IEEE Symposium on Security and Privacy (SP)*, pp. 901–920. IEEE.
36. Camenisch, J., Drijvers, M. and Lehmann, A. (2017) Anonymous attestation with subverted tpms. In *Annual International Cryptology Conference*, pp. 427–461. Springer.
37. El Kassem, N., Chen, L., El Bansarkhani, R., El Kaafarani, A., Camenisch, J., Hough, P., Martins, P. and Sousa, L. (2019) More efficient, provably-secure direct anonymous attestation from lattices. *Future Gener. Comput. Syst.*, **99**, 425–458.
38. Yang, K., Chen, L., Zhang, Z., Newton, C.J., Yang, B. and Xi, L. (2021) Direct anonymous attestation with optimal tpm signing efficiency. *IEEE Trans. Inf. Forensics Secur.*, **16**, 2260–2275.
39. L. Chen and J. Li. Flexible and scalable digital signatures in tpm 2.0. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 37–48, 2013.
40. Goldwasser, S., Micali, S. and Rivest, R.L. (1988) A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, **17**, 281–308.

41. M. Naor and M. Yung, Public-key cryptosystems provably secure against chosen ciphertext attacks. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 427–437, 1990.
42. ISO/IEC 18033-2:2006 *Information technology – Security techniques – Encryption algorithms – Part 2: Asymmetric ciphers*.
43. ANSI X9.62 *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*.