A New Way to Prevent UKS Attacks Using Trusted Computing

Qianying Zhang, Shijun Zhao, and Dengguo Feng

Institute of Software Chinese Academy of Sciences,
ISCAS, Beijing, China.
zqyzsj@gmail.com

Abstract. UKS (unknown key-share) attacks are now common attacks to Authenticated Key Exchange (AKE) protocols. We summarize two popular countermeasures against UKS attacks on the implicitly authenticated key exchange protocols. The first one forces the CA to check the possession of private keys during registration, which is impractical for the CA. The second one adds identities in the derivation of the session key, which leads to modify the protocols already used in practice. By using the key protection capability of hardware security chips, such as TPM or TCM, we propose a new way to prevent UKS attacks that needs no check of possession of private keys and no addition of identities during the derivation of the session key. We modify the CK model to adapt the protocols using hardware security chips. We then implement a protocol once used in NSA, called KEA and subject to UKS attacks, using TCM chips. Our implementation, called tKEA, is secure under our security model. To show the generality of our way, we also show that our new way can prevent UKS attacks on the MQV protocol.

Key words: UKS attacks, Authenticated Key Exchange, Trusted Computing, KEA, CK model

1 Introduction

Diffie and Hellman gave the first key exchange protocol in their seminal paper[12]. Key exchange protocols allow two entities to establish a shared secret key via public communication. In order to provide the authentication of entities' identities, authenticated key exchange(AKE) protocols were proposed. AKE not only allows two entities to compute a shared secret key but also ensures the authenticity of the entities.

To date, a great number of AKE protocols have been proposed and many of them were subsequently broken, such as KEA [26] and MQV [24, 23]. KEA was designed by NSA (National Security Agency) in 1994 and kept secret until 1998. Microsoft researchers K.Lauter and A.Mityagin found that the original KEA protocol is susceptible to UKS attacks [22]. Then they presented a modified version of KEA protocol, called KEA+ [22], which is resistant to UKS attacks, and gave a formal proof. The MQV protocol is another famous and efficient AKE protocol, which is designed by Law, Menezes, Qu, Solinas and Vanstone. This

protocol was first found a UKS attack by Kaliski [19], then Krawczyk [20] found that MQV protocol achieved none of its stated security goals, such as resistance to KCI attacks and the provision of perfect forward secrecy (PFS).

Although KEA and MQV are vulnerable to UKS attacks, they are still widely standardized and used in practice. The OPACITY protocol [29], which is based on KEA, has now been registered as an ISO/IEC 24727-6 authentication protocol [18], and specified in the draft ANSI 504-1 national standard [3]. The MQV protocol has been widely standardized [4, 15, 17, 27, 28]. The new released Trusted Platform Module (TPM) version 2.0 [33] adopts MQV as one of its key exchange primitives. So it's necessary to analyze the security of KEA and MQV protocols with the help of security chips, such as smart card, Trusted Cryptography Module (TCM) [31] and TPM [32], which is also the motivation of our work.

1.1 Related Work

In order to prove the security of AKE protocols formally, Bellare and Rogaway in 1993 provided the first formal definition for an AKE model [5], which we refer to as the BR model. After that, a lot of variants of BR model were presented and many authenticated key exchange protocols were proposed. Based on BR model, Canetti and Krwaczyk proposed the CK model in [8], in which the H-MQV protocol was proved. LaMacchia, Lauter, and Mityagin in [21] defined a new model called eCK, which is much stronger than BR and CK models. They also introduced a new AKE protocol called NAXOS and proved its security in eCK model. However the NAXOS protocol is less efficient in that it requires 4 exponentiations per entity compared to 3 exponentiations for KEA. For more details, we refer the readers to [11] for a comparison and a discussion of the variant models for authenticated key exchange.

1.2 UKS attacks

A UKS attack on an AKE protocol is an attack whereby an entity $\hat{A}$ ends up believing he shares a key with entity $\hat{B}$, and although this is in fact the case, $\hat{B}$ mistakenly believes the key is instead shared with an entity $\hat{E} \neq \hat{A}$. Since the adversary $\hat{E}$ doesn't obtain the shared secret key, he can't modify or decrypt the messages between $\hat{A}$ and $\hat{B}$. However, $\hat{E}$ can take advantage of the entities' false assumptions about the identity with whom they share the key. Let's see an attack scenario described in [13]: $\hat{B}$ is a bank and $\hat{A}$ sends him a digital coin, encrypted with the shared secret key, for deposit into his account. Believing that the key is shared with $\hat{E}$, $\hat{B}$ assumes the coin is from $\hat{E}$ and deposits it into $\hat{E}$'s account instead. Several UKS attacks have been proposed in the literature, such as attacks on STS [6], KEA [22] and MQV [19].

1.3 Contributions

We give our contributions as follows:

1. We summarize UKS attacks and their corresponding solutions on AKE protocols in the literature, and identify two types of attacks, called public key substitution UKS attack and public key registration UKS attack respectively. The details of the two attacks are described in section 2. The usual way to solve these two UKS attacks are: 1) force the CA to check entity's possession of the private key, 2) add entities' identities during the derivation of the session key. We illustrate these solutions by introducing some previous works on preventing UKS attacks on KEA and MQV.
2. We present a new way to prevent the two types of UKS attacks using the key protection capability of hardware security chips, such as TPM or TCM. The main idea is to make use of the security chip to generate the long-term secret key, then register it to a CA who doesn't check the possession of the private key and only makes sure the key comes from a genuine hardware security chip. The key protection capability of the security chip prevents the adversary from getting the plaintext of the private key even he corrupts and controls the security chip. In our security analysis we will show that the key protection capability is crucial for KEA to avoid UKS attacks. To show the generality, we demonstrate our proposed way can prevent the UKS attack on MQV protocol which is adopted by TPM 2.0 version [33].
3. We first modify some adversary abilities in the CK model to adapt the protocols implemented by hardware security chips, and then implement the KEA protocol (subject to UKS attacks) using TCM chips, and finally formally prove our implementation in the variant CK model. Our formal analysis shows that our implementation prevents the UKS attacks.

Readers may say that mandating the use of a TCM or TPM chip is a strong assumption/requirment to the AKE protocol. But we claim that the TPM or TCM will be a common security technology in the future by the following argument. First, more than 2 billion endpoints have been secured with Trusted Computing technology to date. Second, the mainstream processor manufactures now support the Trusted Execution Environment technology (TEE for short), such as Intel's TXT [16], AMD's SVM [1] and ARM's Trustzone [2]. TEE is a special execution mode of the CPU, which protects the codes running in it from being accessed by the normal OS, that's to say, even the adversary comprises the normal OS of the host he cannot get the information running in TEE. To exchange data between the normal OS and the special mode, the Global Platform consortium releases the GP TEE API specification [14]. Now the new released TPM specification, TPM 2.0, supports the TEE technology. In conclusion, it's not a strong assumption to protect AKE protocols using the trusted computing technologies.

1.4 Organization

We summarize the two types of UKS attacks and their corresponding solutions in section 2. In section 3, we give a detail description of the key protection

capability of hardware security chips, show how it can be used on AKE protocols, and give our implementation, which we call tKEA (the letter t stands for trusted). Section 4 modifies the CK model to adapt the protocols implemented by hardware security chips, such as tKEA. Section 5 proves the security of the tKEA implementation. We also show how the key protection capability prevents the UKS attack on the MQV protocol described in [20]. We end the paper with conclusion and our future work in section 6.

2 Two types of UKS attacks and solutions

In this section, we first introduce the Signed Diffie-Hellman and Non-Signed Diffie-Hellman AKE protocols, and then the two types of UKS attacks and the usual ways to prevent these attacks.

2.1 Signed and Non-Signed Diffie-Hellman AKE

Signed Diffie-Hellman AKE. The Signed Diffie-Hellman AKE is such a kind of protocols that first execute a Diffie-Hellman key exchange and then sign all the communication sent between the entities, such as STS [13], SIG-DH [30], SIGMA [9]. Let G be a group of primer order and denote by g a generator of G. Assume that entities have secret/public keys for some digital signature schemes SIG and that entities know each other's registered public key. The hat notation, such as $\hat{A}$, denote an identity of an entity. Denote the signature of a messge $\mathcal{M}$ under the secrete key of an entity $\hat{A}$ as $SIG_{\hat{A}}(M)$. We depict such protocols in figure 1. First, an entity, $\hat{A}$, as an initiator generates an ephemeral secret key x at random and sends a tuple $\{g^x, SIG_{\hat{A}}(g^x, \hat{B})\}$ to $\hat{B}$, the responder. The responder $\hat{B}$ generates an ephemeral secrete key y and replies with a tuple $\{g^y, SIG_{\hat{B}}(g^y, \hat{A})\}$. Both $\hat{A}$ and $\hat{B}$ then verify each other's signature, and compute a shared session key $K = g^{xy}$, if the verification succeeds.

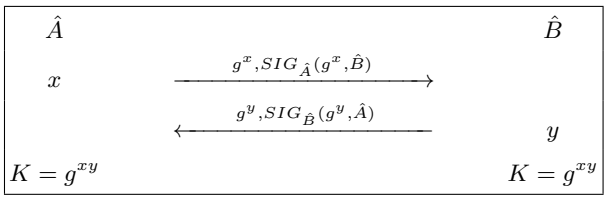

Fig. 1. Signed Diffie-Hellman AKE

Non-Signed Diffie-Hellman AKE. There are some protocols different from the Signed Diffie-Hellman AKE, such as KEA and MQV. As they have no signatures in their exchange messages, we refer to them as Non-Signed Diffie-Hellman AKE protocols. Figure 2 gives an illustration of KEA, a typical Non-Signed

Diffie-Hellman AKE protocol, and its variant KEA+. KEA involves two entities, $\hat{A}$ and $\hat{B}$, with respective secret keys a and b and public keys g^a and g^b. KEA assumes that entities know each other's registered public key. The protocol first runs a Diffie-Hellman key exchange: $\hat{A}$ and $\hat{B}$ each generate its ephemeral secret key, x and y respectively, and exchange the ephemeral public keys g^x and g^y. Then each entity computes g^{ay} and g^{bx} and computes a session key by applying a hash function H to (g^{ay}, g^{bx}). The KEA+ protocol differs from KEA when computing the session key: it incorporates entities' identities in the computation of a session key, i.e., applies the hash function H to a tuple $(g^{ay}, g^{bx}, \hat{A}, \hat{B})$.

$\hat{A}$		$\hat{B}$
$X = g^x$	$\xrightarrow{X}$	
	$\xleftarrow{Y}$	$Y = g^y$
$K = H(\sigma_{\hat{A}})$		$K = H(\sigma_{\hat{B}})$
KEA:	$\sigma_{\hat{A}} = (g^{ay}, g^{bx})$, $\sigma_{\hat{B}} = (g^{ay}, g^{bx})$	
KEA+:	$\sigma_{\hat{A}} = (g^{ay}, g^{bx}, \hat{A}, \hat{B})$, $\sigma_{\hat{B}} = (g^{ay}, g^{bx}, \hat{A}, \hat{B})$	

Fig. 2. Non-Signed Diffie-Hellman AKE: KEA and KEA+

2.2 UKS attacks on Non-Signed Diffie-Hellman AKE

Baek and Kim give a good summarization of UKS attacks on the Signed Diffie-Hellman AKE protocols [22]. Here we summarize UKS attacks on Non-Signed Diffie-Hellman AKE protocols. We categorize them into public key substitution UKS attack and public key registration UKS attack respectively.

Public Key Substitution UKS Attack. This kind of attack happens to some protocols when the CA doesn't check the possession of the private key. [22] points out that the original KEA protocol is insecure under this type of attack. Consider two entities $\hat{A}$ and $\hat{B}$ preparing to start a session. An adversary $\mathcal{M}$ registers a public key g^a of $\hat{A}$ as his own public key. Then $\mathcal{M}$ intercepts the session between $\hat{A}$ and $\hat{B}$. $\mathcal{M}$ forwards the ephemeral public key g^x from $\hat{A}$ to $\hat{B}$ and ephemeral public key g^y from $\hat{B}$ to $\hat{A}$. Since $\mathcal{M}$ has the same public key as $\hat{A}$, both $\hat{A}$ and $\hat{B}$ will compute identical session keys. However, $\hat{A}$ complete a session with $\hat{B}$ and $\hat{B}$ complete a session with $\mathcal{M}$.

The usual way to solve this kind of UKS attack is to force the CA check the possession of the private key. If the CA checks, $\mathcal{M}$ can't register the public key of $\hat{A}$, then $\hat{A}$ and $\hat{B}$ will compute non-identical session keys. However, as the proof of knowledge check is rarely done in practice, this way to prevent UKS attacks is impractical.

Public Key Registration UKS Attack. The typical attack example is a UKS attack on MQV found by Kaliski [19]. Let us review MQV first. MQV is

another important Non-Signed Diffie-Hellman AKE protocol, and it was stated to have a lot of security properties, such as resistance to UKS attacks and KCI attacks. MQV and HMQV are depicted in figure 3. Entities $\hat{A}$ and $\hat{B}$ have their private/public key pairs (a, g^a) and (b, g^b) respectively. The ephemeral public keys in their exchange messages are g^x and g^y. The computation of the session key by $\hat{A}$ (and similarly by $\hat{B}$) is a hash to $(YB^e)^{x+da}$. The only difference between MQV and HMQV is the computation of d and e. The former only uses the ephemeral public key, while the later adds the identity information and uses a hash function in the computation. However, the slight modification is crucial in the security analysis.

We describe the public key registration UKS attack on MQV in figure 4. An adversary $\mathcal{M}$ intercepts the ephemeral public key $X = g^x$ sent from $\hat{A}$ to $\hat{B}$. Based on X, $\mathcal{M}$ computes a private/public key pair (c, g^c), and sends an ephemeral public key Z to $\hat{B}$. After receiving Z, $\hat{B}$ generates a random ephemeral key $Y = g^y$ and sends it to $\mathcal{M}$. $\mathcal{M}$ transmits Y to $\hat{A}$. We denote the session between $\hat{A}$ and $\hat{B}$ by s, and the session between $\hat{B}$ and $\mathcal{M}$ by s'. We can see that the key pair (c, g^c) and the ephemeral key Z are computed so cleverly that s and s' have the identical shared secret key.

From the attack described above, we can see that check proof of knowledge of private key can not thwart this attack as the adversary holds the private key c. The usual way to prevent this kind of attack is to include the identities in the derivation of the session key. Krawczyk and Menezes respectively present HMQV and a variant of MQV [25] which both resist this UKS attack. HMQV adds the identity and use a hash function when computing d and e, while [25] includes identities in the derivation of the session key. From their solutions we can see that the inclusion of identities in the derivation of the session key is an effective way to thwart the public key registration UKS attack.

$\hat{A}$		$\hat{B}$
$X = g^x$	$\xrightarrow{X}$	
	$\xleftarrow{Y}$	$Y = g^y$
$\sigma_{\hat{A}} = (YB^e)^{x+da}$		$\sigma_{\hat{B}} = (XA^d)^{y+eb}$
$K = H_1(\sigma_{\hat{A}})$		$K = H_1(\sigma_{\hat{B}})$
MQV:	$d = 2^l + (X mod 2^l)$, $e = 2^l + (Y mod 2^l)$, $l = \lvert q\rvert/2$	
HMQV:	$d = H_2(X, \hat{B})$, $e = H_2(Y, \hat{A})$	

Fig. 3. Non-Signed Diffie-Hellman AKE: MQV and HMQV

3 Trusted Computing Key Protection Capability

Trusted Cryptography Module (TCM) is a hardware security chip similar to Trusted Platform Module (TPM), a small tamper-resistant cryptographic chip

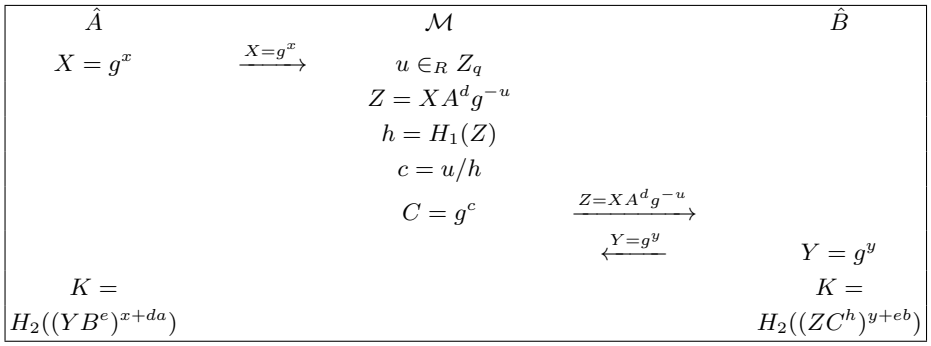

Fig. 4. A UKS attack on MQV

embedded in a computer platform (e.g. on a PC motherboard). TCM provides a set of cryptographic functionalities, such as public-key decryption/encryption (SM2-1), hash (SM3), random number generating, and key exchange (SM2-2) and so on. TCM stores secret data, such as private keys and security crucial user data, in a shielded location where data is protected against interference and prying, and we call it the key protection capability.

To operate the secret data in the shielded location, TCM provides a set of commands for users. Take the SM2-2 key exchange for example, TCM provides the TCM_CreateKeyExchange and TCM_GetKeyExchange to create a private/public key pair and generate a session key respectively:

- *TCM_CreateKeyExchange*: TCM creates a private/public SM2 key pair, which we denote by (a, g^a), in the TCM's shielded location, and returns the public part of the SM2 key pair.
- *TCM_GetKeyExchange*: Input a public key of SM2, e.g, g^b, and return a session key g^{ab}.

We identify two security properties from the key protection capability, which will be used in the security analysis of tKEA. First, a user who has the control of an SM2-2 key pair generated by TCM cannot get the plaintext of the private key, and the only way he can use the SM2-2 key is through TCM APIs. Second, a user cannot have TCM chips generate a specified key pair, such as (a, g^a), as the key generation is controlled by TCM and users have no control of the generation of keys. From the second security property we can see that $\mathcal{M}$ can not register a specified key.

Implementation of tKEA. Here we show how to implement KEA protocol using trusted computing technology. Our implementation has two phases, registration phase and key exchange phase.

The registration phase involves a security TCM chip $\mathcal{T}$, a host $\mathcal{H}$, and a CA $\mathcal{C}$. $\mathcal{T}$ and its host $\mathcal{H}$ compose a whole entity. Before the registration phase, $\mathcal{T}$ generates an Attestation Identity Key (AIK) pair (sk_T, pk_T) (AIK is used to identify a platform in trusted computing, here we use it to certify the long-term key of an entity) and then registers the public key, pk_T, to a CA (this

CA issues certificates to platforms, and is not $\mathcal{C}$ in the registration phase, which issues certificates for long-term keys) through protocols such as Privacy-CA [10], which is out of the scope of this paper. If stronger anonymity is required, please refer to DAA [7] solutions. After getting the AIK certificate, the registration proceeds as follows:

1. $\mathcal{H}$ calls *TCM_CreateKeyExchange* API of $\mathcal{T}$, $\mathcal{T}$ generates an SM2-2 key pair (a, g^a) which is the long-term key of this entity.
2. $\mathcal{H}$ then calls *TCM_CerifyKey* command, which has $\mathcal{T}$ make a **statement** about (a, g^a) using the AIK: "this key is protected by TCM, and its plaintext will not be revealed", and return the **statement** to $\mathcal{H}$. The **statement** is actually a signature of the SM2-2 key by AIK. The AIK has a feature that it only signs keys generated within the TCM. This feature assures $\mathcal{C}$ that the SM2-2 key is a real TCM-generated key.
3. $\mathcal{H}$ transmits the **statement** to $\mathcal{C}$. $\mathcal{C}$ verifies the **statement**, and makes sure that the public key g^a is generated by a real TCM chip. If the verification succeeds, $\mathcal{C}$ issues a certificate on g^a and sends it to $\mathcal{H}$.

The key exchange phase is shown in figure 5, and actually is the procedure of running the KEA protocol between two entities, e.g., $\hat{A}$ and $\hat{B}$. $\hat{A}$ consists of a TCM $\mathcal{T}_1$ and its host $\mathcal{H}_1$, while $\hat{B}$ consists of a TCM $\mathcal{T}_2$ and its host $\mathcal{H}_2$. $\hat{A}$'s long-term public key is $A = g^a$, and $\hat{B}$'s long-term public key is $B = g^b$.

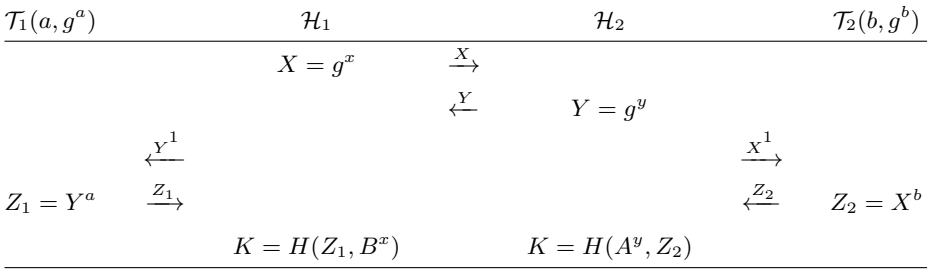

Fig. 5. tKEA: Implementation of KEA using TCM

4 Security Model for tKEA

In this section we introduce a variant of CK model on which the security analysis of tKEA is based. For further details of CK model, please consult [8]. We modify the CK model by 1) revising the corruption query, and 2) adding an establish query. The modified corruption query models the key protection capability of TCM, and the establish query allows an adversary to register public keys of adversary-controlled entities at any time in the experiment, that is to say, the adversary is allowed to mount UKS attacks.

[1] X and Y are transmitted to $\mathcal{T}$ through *TCM_GetKeyExchange*.

4.1 Sessions

tKEA runs in a network of interconnected entities where each entity can be activated to run an instance of the protocol called a session. Within a session an entity can be activated to initiate the session or to respond to an incoming message. As a result of these activations, the entity creates and maintains a session state, generates outgoing messages, and eventually completes the session by outputting a session key and erasing the session state. There are two roles in a session, the entity that sends the first message in a session is called the **initiator** and the other the **responder**. We denote the initiator by $\mathcal{I}$ and the responder by $\mathcal{R}$. We identify an AKE session by a 5-tuple $(role, \hat{A}, \hat{B}, X, Y)$ where $role$ denotes the role, X is the outgoing DH value and Y is the incoming DH value to the session. The session $(\mathcal{R}, \hat{B}, \hat{A}, Y, X)$ (if it exists) is said to be **matching** to session $(\mathcal{I}, \hat{A}, \hat{B}, X, Y)$.

4.2 Attack Model

The AKE experiment involves multiple honest entities and an adversary $\mathcal{M}$ connected via an unauthenticated network. The adversary $\mathcal{M}$ is modeled as a probabilistic Turing machine and controls all communications. $\mathcal{M}$ can intercept and modify messages sent over the network. $\mathcal{M}$ also schedules all session activations and session-message delivery. In addition, in order to model potential disclosure of secret information, the attacker is allowed to access secret information via the following queries:

- *session-state(s)* - $\mathcal{M}$ queries directly at session s while still incomplete and learns the session state for s (which may include, for example, the secret exponent of an ephemeral DH value but not the long-term secret key).
- *session-key(s)* - $\mathcal{M}$ obtains the session key for a session s, provided that the session holds a key.
- *corruption(entity)* - As for the information not stored in the TCM's shielded location, such as the session states and session keys, $\mathcal{M}$ learns all of them. As for the long-term keys stored in the TCM's shielded location, $\mathcal{M}$ has the ability to use it, such as computing $CDH(A, X)$ (A stands for the long-term key, X stands for an element in G whose exponent is unknown) but cannot get the plaintext of the private key.
- *establish(entity)* - This query allows $\mathcal{M}$ to register a public key generated by a controlled TCM, and $\mathcal{M}$ is able to use the cryptographic functionality provided by the key. $\mathcal{M}$ can fully control *entity* by taking advantage of the controlled key. If $\mathcal{M}$ registers a public key not generated in TCM, the CA will deny this registration after checking the AIK signature of the public key.

The adversary can make above queries to gain local information. We say that a completed session is "clean" if this session as well as its matching session (if it exists) is not subject to any of session-state, session-key, corruption queries.

Eventually $\mathcal{M}$ should select a clean completed session, which is called a test session, and make query **Test**(s) and is given a challenge value C.

- *Test(s)* - Pick $b \xleftarrow{R} 0, 1$. If $b = 1$, obtain $C \leftarrow$ *sesssion-key(s)*; otherwise provide C with a value r randomly chosen from the probability distribution of keys generated by tKEA.

$\mathcal{M}$ now can continue to make session-state, session-key, corruption, establish queries but is not allowed to expose the test nor any of the entities involved in the test session. At the end of its run, $\mathcal{M}$ outputs a bit b'. We will refer to an adversary with the above capabilities as a **KE-adversary**.

Definition 1. *An AKE protocol Π is called SK-secure if the following properties hold for any KE-adversary $\mathcal{M}$ defined above*

1. *when two uncorrupted entities complete matching sessions, they output the same key, and*
2. *the probability that $\mathcal{M}$ guesses the bit b (i.e., outputs $b' = b$) from the Test query correctly is no more that 1/2 plus a negligible fraction.*

The advantage of a KE-adversary participating in the above AKE experiment against a protocol Π is defined as

$$\mathbf{Adv}_{\Pi}^{AKE}(\mathcal{M}) = \Pr[\mathcal{M} \text{ wins}] - \tfrac{1}{2}.$$

5 Security of tKEA and MQV

In this section, we first formally prove the tKEA protocol, and then show that the protection capability provided by TCM can prevent the UKS attack on MQV.

5.1 Security proof of tKEA

We show that the tKEA with a hash function modeled as a random oracle satisfies AKE security against a KE-adversary defined in section 4 under the GDH assumption in a group G and with the help of the key protection capability of TCM chips. The GDH assumption in G is that the CDH problem in G cannot be solved in polynomial time with non-negligible success probability even when a DDH oracle for G is available.

Let $\mathcal{M}$ be any KE-adversary against tKEA. We start by observing that since the session key of the test session is computed as $K = H(\sigma)$ for some 2-tuple σ, the adversary $\mathcal{M}$ has only two ways to distinguish K from a random value:

1. Forging attack. At some point $\mathcal{M}$ queries H on the same 2-tuple σ.
2. Key-replication attack. $\mathcal{M}$ succeeds in forcing the establishment of another session that has the same session key as the test session.

We will show that if either of the attacks succeeds with non-negligible probability then there exists an efficient attacker against the GDH problem or breaking the TCM protection capabilities.

Security analysis of the key-replication attack. The security proofs of KEA+ [22] and NAXOS [21] show that the key-replication attack is impossible if random oracles produce no collisions. Let's take KEA+ for example. If $\mathcal{M}$ finds some session with the same 4-tuple as the test session, then this session must be executed by the same two entities, $\hat{A}$ and $\hat{B}$. Let the ephemeral public keys of this session be X' and Y'. Since the session has the same signature as the test session, $CDH(A, Y')$ must be equal to $CDH(A, Y)$ and $CDH(B, X')$ equal to $CDH(B, X)$. This implies that $X = X'$ and $Y = Y'$, and thus these two sessions must be identical.

However, the key-replication attack can happen to KEA. Lauter and Mityagin describe this attack in [22]. We here review this attack. An adversary $\mathcal{M}$ registers a public key g^a of some honest entity $\hat{A}$ as $\mathcal{M}$'s own public key. Then $\mathcal{M}$ intercepts a key-exchange session between $\hat{A}$ and $\hat{B}$ and at the same time starts a session between $\mathcal{M}$ and $\hat{B}$. Now $\mathcal{M}$ forwards ephemeral public key g^x from $\hat{A}$ to $\hat{B}$ and ephemeral public key g^y from $\hat{B}$ to $\hat{A}$. Since $\mathcal{M}$ has the same public key as $\hat{A}$, both $\hat{A}$ and $\hat{B}$ will complete identical session keys, however they participate in two different sessions. $\hat{B}$ participates in a session with $\mathcal{M}$ while $\hat{A}$ participates in a session with $\hat{B}$. Then $\mathcal{M}$ can announces one of the two sessions as a test session and reveals the session key of the other session. To resist UKS attacks, KEA+ adds the identities of the participating entities to the tuples, see figure 2. This slight modification prevents adversaries from activating a session with the same tuple as the test session, thereby preventing $\mathcal{M}$ from performing a key-replication attack. We show below that the key protection capability that TCM provides can also prevent UKS attacks.

In the tKEA, we demonstrate that if an adversary $\mathcal{M}$ mounts a key-replication attack, he can break the key protection capability of TCM chips. We denote the test session by s and denote the corresponding 2-tuple by $(CDH(A, Y), CDH(B, X))$. Correspondingly, we denote another session by s' which has the same session key with s and the corresponding 2-tuple by $(CDH(A', Y'), CDH(B', X'))$ on which $\mathcal{M}$ queries H to get the session key of s. A' and B' are public keys $\mathcal{M}$ registers to the CA through the *establish(entity)*, and $\mathcal{M}$ can do the computation of $CDH(A' or B', T)$ for any T whose exponent is unknown. Since s and s' has the same session key, $CDH(A', Y')$ must be equal to $Z_1 = CDH(A, Y)$ and $CDH(B', X')$ equal to $Z_2 = CDH(B, X)$. Since $CDH(A', Y') = Z_1$, we can get $Y' = Z_1^{\frac{1}{a'}}$ and $A' = Z_1^{\frac{1}{y'}}$. The only two ways for $\mathcal{M}$ to get a pair (A', Y') satisfying equation $CDH(A', Y') = Z_1$ are:

1. Register a controlled key A' to the CA, and compute the ephemeral public key $Y' = Z_1^{\frac{1}{a'}}$ where a' denotes the private key of A'.
2. Generate an ephemeral key pair $(y', Y' = g^{y'})$, and register $A' = Z_1^{\frac{1}{y'}}$ to the CA.

We can see that the first way requires $\mathcal{M}$ to get the plaintext of the public key A', and the second way requires $\mathcal{M}$ to register a specified key. However, we can see that both of the two ways violate the key protection capability described in section 3.

Security analysis of the forging attack. We are left to show impossibility of a forging attack. Let $\mathcal{M}$ be an AKE adversary against tKEA. Consider the following GDH adversary $\mathcal{S}$:

$\mathcal{S}$ takes input a pair $(X_0, Y_0) \in G^2$. $\mathcal{S}$ is also given access to a DDH oracle DDH. $\mathcal{S}$ creates an AKE experiment which includes a number of honest entities and an adversary $\mathcal{M}$. We assume that the experiment involves at most n entities and that each entity participates in at most k AKE sessions. $\mathcal{S}$ randomly selects one of the honest entity (say, this is a entity $\hat{A}$) and sets the public key of $\hat{A}$ to be X_0. All the other entities compute their keys normally. $\mathcal{S}$ picks a number i_k at random from $\{1, ..., k\}$ and initializes the counter at $i = 1$ (i counts sessions that $\hat{A}$ participates in). $\mathcal{S}$ runs an AKE experiment with adversary $\mathcal{M}$ and handles queries made by $\mathcal{M}$ as follows:

1. When $\mathcal{M}$ queries a hash function H on a string v, return the value of $H_{sim}(v)$. The procedure $H_{sim}(\cdot)$ which simulates a random oracle H is described later on.
2. When $\mathcal{M}$ starts a session $(\hat{B}, \hat{C}, role)$ between entities $\hat{B}$ and $\hat{C}$ both different from a selected entity $\hat{A}$, $\mathcal{S}$ follows the protocol for tKEA. Denote $\hat{B}$'s secret key as b, $\hat{B}$'s public key as $B = g^b$ and $\hat{C}$'s public key as C. If role = initiator, $\hat{B}$ picks a random exponent x, returns $X = g^x$, waits for the reply Y and computes a session key $K = H_{sim}(Y^b, C^x)$. If role = responder, $\hat{B}$ waits for $\hat{C}$'s initiating message X, picks a random exponent y, replies with $Y = g^y$ and computes a session key $K = H_{sim}(C^y, X^b)$.
3. When $\mathcal{M}$ starts a session $(\hat{A}, \hat{C}, role)$ (here $\hat{A}$ is the special entity whose public key is a GDH challenge X_0), $\mathcal{S}$ cannot follow the protocol since it doesn't know a secret for $\hat{A}$'s public key. Denote $\hat{C}$'s public key as C. If $\hat{A}$ is an initiator, it picks a random exponent x, sends $X = g^x$ to $\hat{C}$ and waits for the reply Y . Now it sets a session key to be $H_{spec}(1, Y, C^x)$, see the description of the procedure H_{spec} below. If $\hat{A}$ is the responder, it waits for an initiating message X, picks a random exponent y, replies with g^y and computes a session key $K = H_{spec}(2, X, C^y)$.
4. When $\mathcal{M}$ starts a session $(\hat{B}, \hat{A}, role)$ for some entity B, where the second entity is the selected entity $\hat{A}$, $\mathcal{S}$ first checks if $i = i_k$. If "no", $\mathcal{S}$ increments the counter i and behaves according to the rule for Query 2. If the check succeeds, $\mathcal{S}$ declares $(\hat{B}, \hat{A}, role)$ to be a "special session". In a special session, $\hat{B}$ outputs a message Y_0 (which is the second part of the GDH challenge) and doesn't compute a session key.
5. When $\mathcal{M}$ makes a session key-reveal or ephemeral secret key-reveal query against some session (different from the special session), $\mathcal{S}$ returns to $\mathcal{M}$ a session key or an ephemeral secret key for this session (which was computed previously in Queries 2, 3 or 4). If $\mathcal{M}$ tries to reveal a session key or an ephemeral secret key of the special session, $\mathcal{S}$ declares failure and stops the experiment.
6. When $\mathcal{M}$ makes a corruption on some entity $\hat{C}$ (different from $\hat{A}$ and $\hat{B}$), $\mathcal{S}$ returns the secret key of $\hat{C}$ as well as ephemeral secret keys of all current

AKE sessions executed by $\hat{C}$ and gives $\mathcal{M}$ full control over $\hat{C}$. If $\mathcal{M}$ tries to corrupt $\hat{A}$ or $\hat{B}$ (after a special session is selected), $\mathcal{S}$ declares failure.

When $\mathcal{M}$ stops, $\mathcal{S}$ goes over all random oracle queries made by $\mathcal{M}$ and checks (using a DDH oracle DDH) if any of them includes the value of $CDH(X_0, Y_0)$. If "yes", return $CDH(X_0, Y_0)$ to the GDH challenger. If "no", $\mathcal{S}$ declares failure.

Function $H_{sim}(Z_1, Z_2)$. This function implements a random oracle on valid signatures of the tKEA. The function proceeds as follows:

- If the value of the function on that input has been previously defined, return it.
- If not defined, go over all the previous calls to $H_{spec}(\cdot)$ and for each previous call of the form $H_{spec}(i, Y, Z) = v$ check if

$$Z = Z_{3-i} \text{ and } DDH(X_0, Y, Z_i) = 1.$$

  If all these conditions hold, return v.
- If not found, pick a random w, define $H_{sim}(Z_1, Z_2) = w$ and return w.

Function $H_{spec}(i, Y, Z)$. Informally, H_{spec} implements a random oracle on signatures which are not known to $\mathcal{S}$. Specifically, the input corresponds to a signature (Z_1, Z_2), where $Z_i = CDH(X_0, Y)$ (here X_0 is a part of the GDH challenge) and $Z_{3-i} = Z$. This signature is not known to $\mathcal{S}$ since $\mathcal{S}$ cannot compute $CDH(X_0, Y)$. The function proceeds as follows:

- If the value of the function on that input has been previously defined, return it.
- If not defined, go over all the previous calls to $H_{sim}(\cdot)$ and for each previous call of the form $H_{sim}(Z_1, Z_2) = v$ check if

$$Z = Z_{3-i} \text{ and } DDH(X_0, Y, Z_i) = 1.$$

  If all these conditions hold, return v.
- If the check failed for all the calls, pick a random w, define $H_{spec}(i, Y, Z)$ to be w and return w.

Analysis of $\mathcal{S}$. The the running time of $\mathcal{S}$ is the time needed to run an AKE experiment and $\mathcal{M}$ plus the time needed to handle H-queries. Each call to H_{sim} or H_{spec} requires $\mathcal{S}$ to pass over all the previously made queries. Thus, time needed to handle H-queries is proportional to a squared number of queries. Since the number of H-queries is upper-bounded by the running time of $\mathcal{M}$, we can bound the running time of $\mathcal{S}$ by $O(t^2)$, where t is the running time of $\mathcal{M}$.

We are now going to show that if $\mathcal{M}$ doesn't corrupt $\hat{A}$ and doesn't reveal a session key or an ephemeral secret key for the special session, then the simulation of an AKE experiment is perfect. That is, the view of $\mathcal{M}$ in the experiment run by $\mathcal{S}$ is identically distributed to the view of $\mathcal{M}$ in an authentic experiment. To be precise, the view of $\mathcal{M}$ consists of public keys of all the entities, secret keys of the corrupted entities, ephemeral public keys of all the sessions, ephemeral secret keys and session keys of the corrupted sessions and of the random oracle's responses.

We start by observing that secret/public key pairs of all honest entities except $\hat{A}$ are distributed correctly. A public key of $\hat{A}$ is also distributed correctly, however $\mathcal{S}$ doesn't know the secret key for it. By assumption, $\mathcal{M}$ doesn't corrupt $\hat{A}$ and thus $\mathcal{M}$ wouldn't notice that. Similarly, ephemeral secret/public values of all sessions except the test session are distributed as in the original protocol. The ephemeral public key Y_0 in the test session is also distributed correctly, although $\mathcal{S}$ doesn't know a secret for it. Again, we assume that $\mathcal{M}$ doesnt corrupt the test session and so $\mathcal{S}$ wouldn't have to reveal it.

The adversary can obtain the random oracle's responses either by querying H directly or by revealing session keys from honest entities. Without loss of generality, we can assume that the adversary queries a random oracle only on tuples of the form (Z_1, Z_2), where $Z_1, Z_2 \in G$. To ensure that the simulation is perfect, we need to verify that i) the oracle responses are selected at random and ii) if the same argument is queried several times, the same value is returned.

Recall that $\mathcal{S}$ handles two types of queries differently. Queries of the first type are fully specified 2-tuples and such queries are made both by $\mathcal{M}$ and by honest entities. They are handled by the function H_{sim}. Queries of the second type are made only by $\hat{A}$ and such queries have one of the components unspecified. That is, a value Z_i (for some $i = 1, 2$) is unknown and it is specified by $Y \in G$ such that $Z_i = CDH(X_0, Y)$. These queries are handled by H_{spec}. Note that distinct H_{spec} arguments correspond to distinct queries to H.

In our construction of H_{sim} and H_{spec}, a new random value of H is chosen every time the argument wasn't found in the record of previous queries. Thus, condition i) is satisfied and we only need to show that by querying the same argument several times, M always receives the same answers. If the same query is made for the second time either to H_{sim} or to H_{spec}, the same answer is returned. The only conflicts can arise if a query previously handled by H_{sim} is queried again to H_{spec} or vice versa. That is, H_{sim} was called on a tuple (Z_1, Z_2) and H_{spec} - on (i, Y, Z) where $Z_i = CDH(X_0, Y)$ and $Z_{3-i} = Z$. Note that one can check whether these queries correspond to identical signatures by checking that $Z_{3-i} = Z$ and that $DDH(X_0, Y, Z_i) = 1$.Whichever of the functions was called first, on the second call (to the other function) $\mathcal{S}$ will go over all previous calls to the first function and do such a check. If a match is found, the previously defined value is returned. This guarantees that condition ii) is also satisfied.

We showed that, provided $\mathcal{M}$ doesn't corrupt $\hat{A}$ or the special session, the simulation of the AKE experiment is perfect. Since the entity $\hat{A}$ and the special session are chosen at random, a test session selected by $\mathcal{M}$ matches the special session with probability $1/nk$ (recall that n is the number of entities in the experiment and k is the maximal number of sessions any entity can participate in). In this case, the simulation is perfect since $\mathcal{M}$ doesn't corrupt the test session. We know that a successful adversary must reveal the signature of the test session. Whenever $\mathcal{M}$ wins in the AKE experiment and the test session was guessed correctly, $\mathcal{S}$ reveals the signature of the test session which contains $CDH(X_0, Y_0)$, and therefore wins in the GDH experiment. To summarize the

lengthy proof, for any AKE adversary $\mathcal{M}$ running in time t we constructed a GDH solver $\mathcal{S}$ which runs in time $O(t^2)$ such that

$$\mathbf{Adv}^{GDH}(\mathcal{S}) \geq \frac{1}{nk}\mathbf{Adv}^{AKE}_{tKEA}(\mathcal{M})$$

As for the wPFS and KCI security property of tKEA, they can be proved directly following the proof above. So the tKEA implementation achieves the same security protocol of KEA+, i.e., the protection capabilities provided by TCM indeed can improve the AKE protocols.

5.2 Securing MQV

The MQV protocol has been adopted by TPM 2.0 version, which might be widely used in practice (the Microsoft Surface Pro has been equipped by TPM 2.0). We show that our way of using the key protection capability to prevent UKS attacks on KEA can prevent the UKS attack [19] on the MQV protocol. Figure 4 shows this attack. To attack MQV, the adversary $\mathcal{M}$ registers a public key $C = g^c$ to the CA. As $\mathcal{M}$ knows the private key of C, the CA cannot deny the registration of C even it requires proof of knowledge of the private key. However, if the CA requires that the key must come from a security chip, such as TPM or TCM, this UKS attack can be prevented. That's because if the key is generated in a security chip, $\mathcal{M}$ can not specify a key to be registered. That's to say, $\mathcal{M}$ cannot register $C = g^c$ to the CA.

6 Conclusion and Future Work

This paper summarizes two types of UKS attacks to Non-Signed Diffie-Hellman AKE protocols and the usual corresponding solutions to the two attacks. One of the solutions requires the CA to check the possession of the private key, which is unpractical, and the other solution modifies the original protocol. Motivated by the key protection capability of security chips, we present our solution of preventing UKS attacks on AKE protocol.

We introduce the key protection capability of hardware security chips and give a variant of CK model which covers UKS attacks. Through the security proof of tKEA in our variant model, we show that our new way to prevent UKS attacks is effective. We also show the generality of our new way by preventing the UKS attack on MQV protocol.

In section 5, we show that our new way can prevent the UKS attack on MQV but without a formal proof. In the future, we are going to give MQV a formal proof in our security model. We will also check whether the key protection capability can provide other advantages to AKE protocols.

References

1. Advanced Micro Devices. AMD64 Virtualization Codenamed "Pacifica" Technology, Secure Virtual Machine Architecture Reference Manual. `http://www.mimuw.`

edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf, May 2005.
2. T. Alves and D. Felton. TrustZone: Integrated hardware and software security - enabling trusted computing in embedded systems. *Information Quarterly*, 3(4):18–24, 2004.
3. ANSI. 504-1: Information technology-generic identity command set, part 1: Card application command set.
4. ANSI. American National Standard, X9.42-2001.
5. M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology-CRYPTO93*, pages 232–249. Springer, 1994.
6. S. Blake-Wilson and A. Menezes. Unknown key-share attacks on the station-to-station (STS) protocol. In *Public Key Cryptography*, pages 154–170. Springer, 1999.
7. 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. ACM, 2004.
8. R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *Advances in Cryptology-EUROCRYPT 2001*, pages 453–474. Springer, 2001.
9. R. Canetti and H. Krawczyk. Security analysis of IKE's signature-based key-exchange protocol. In *Advances in Cryptology-CRYPTO 2002*, pages 143–161. Springer, 2002.
10. L. Chen and B. Warinschi. Security of the TCG privacy-CA solution. In *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*, pages 609–616. IEEE, 2010.
11. K.-K. R. Choo, C. Boyd, and Y. Hitchcock. Examining indistinguishability-based proof models for key establishment protocols. In *Advances in Cryptology-ASIACRYPT 2005*, pages 585–604. Springer, 2005.
12. W. Diffie and M. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.
13. W. Diffie, P. C. Van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, 1992.
14. GlobalPlatform. GlobalPlatform Device Technology TEE Client API Specification. http://www.globalplatform.org/specificationsdevice.asp, 2010.
15. IEEE. 1363-2000: Standard specifications for public key cryptography.
16. Intel Corp. Intel Trusted Execution Technology. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf, 2012.
17. ISO/IEC. 11770-3:2008 information technology - security techniques - key management - part 3: Mechanisms using asymmetric techniques.
18. ISO/IEC. 24727-6:2010 Identification cards - integrated circuit card programming interfaces - part 6: Registration authority procedures for the authentication protocols for interoperability, 2011.
19. B. S. Kaliski Jr. An unknown key-share attack on the MQV key agreement protocol. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):275–288, 2001.
20. H. Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol. In *Advances in Cryptology-CRYPTO 2005*, pages 546–566. Springer, 2005.
21. B. LaMacchia, K. Lauter, and A. Mityagin. Stronger security of authenticated key exchange. In *Provable Security*, pages 1–16. Springer, 2007.

22. K. Lauter and A. Mityagin. Security analysis of KEA authenticated key exchange protocol. In *Public Key Cryptography-PKC 2006*, pages 378–394. Springer, 2006.
23. L. Law, A. Menezes, M. Qu, J. Solinas, and S. Vanstone. An efficient protocol for authenticated key agreement. *Designs, Codes and Cryptography*, 28(2):119–134, 2003.
24. A. Menezes, M. Qu, and S. Vanstone. Some new key agreement protocols providing mutual implicit authentication. In *Second Workshop on Selected Areas in Cryptography (SAC'95)*, 1995.
25. A. Menezes and B. Ustaoglu. On the importance of public-key validation in the MQV and HMQV key agreement protocols. In *Progress in Cryptology-INDOCRYPT 2006*, pages 133–147. Springer, 2006.
26. National Institute of Standards and Technology. SKIPJACK and KEA Algorithm Specifications Version 2.0. `http://csrc.nist.gov/groups/ST/toolkit/documents/skipjack/skipjack.pdf`, May 1998.
27. NIST. Special publication 800-56 (DRAFT): Recommendation on key establishment schemes, Draft 2, January 2003.
28. NIST. SP 800-56 (DRAFT): Special publication 800-56, recommendation for pairwise key establishment schemes using discrete logarithm cryptography, July 2005.
29. E. L. Saint, D. Fedronic, and S. Liu. Open Protocol for Access Control Identification and Ticketing with Privacy. `http://www.smartcardalliance.org/resources/pdf/OPACITY_Protocol_3.7.pdf.`, 2011.
30. V. Shoup. *On formal models for secure key exchange*. Citeseer, 1999.
31. State Password Administration Committee in China. Functionality and Interface Specification of Cryptographic Support Platform for Trusted Computing. `http://www.oscca.gov.cn/UpFile/File64.PDF`, December 2007(in Chinese).
32. Trusted Computing Group. TPM Main Specification Level 2 Version 1.2, Revision 116. `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`, March 2011.
33. Trusted Computing Group. Trusted Platform Module Library Specification - Part 1 Architecture, Family "2.0" Level 00, Revision 00.99. `http://www.trustedcomputinggroup.org/developers/trusted_platform_module`, August 2013.