Cryptanalysis of Self-Generated-Certificate Public Key Encryption without Pairing in PKC07

Xu an Wang[1], Xinyi Huang[2], Xiaoyuan Yang[1]

[1]Key Laboratory of Information and Network Security
Engneering College of Chinese Armed Police Force, P.R. China
wangxahq@yahoo.com.cn

[2]Centre for Information Security Research
School of Information Technology and Computer Science
University of Wollongong, Australia
xyh068@uow.edu.au

Abstract. In PKC07,Lai and Kou proposed a self-generated-certificate public key encryption scheme without pairing [1].In this paper, we show that this scheme is not secure.Our contribution are as following: First, we point out that the security model is not sufficient for their scheme and propose a new security model for CL-PKE and SGC-PKE.Second, we give a man-in-the-middle attack scheme to their scheme in the new security model and propose a rescue SGC-PKE scheme by giving little change to the original scheme .We further point out the reason for successfully attacking is binding the user's secret key with the *multiply* of partial public key from KGC and user's self-generated public key instead of binding with partial public key from KGC and user's self-generated public key *independently*. At last, Based on Baek et al's security proof for their CLPKE scheme without pairing in [6],we prove our new scheme's security in the random oracle model.

1 Introduction

In Asiacrypt'03, Al-Riyami and Paterson introduced the concept of Certificateless Public Key Cryptography (CL-PKC) .It is a new cryptographic paradigm which lies between Traditional Public Key Cryptography and Identity Based Cryptography. The idea is to eliminate the inherent key-escrow problem of Identity-Based Cryptography (IBC). At the same time, it preserves the attractive advantage of IBC which is the absence of digital certificates (issued by Certificate Authority) and their important management overhead. Different from IBC, the user's public key is no longer an arbitrary string. Rather, it is similar to the public key used in the traditional PKC generated by the user. A crucial difference between them is that the public key in CL-PKC does not need to be explicitly certified as it has been generated using some partial private key obtained from

the trusted authority called Key Generation Center (KGC). Note here that the KGC does not know the user's private keys since they contain secret information generated by the users themselves, thereby removing the escrow problem in IBC [6–10].

It seems that CL-PKC has successfully solved the explicit certification problem. Unfortunately, it suffers Denial-of-Decryption (DoD) Attack found by Liu and Au in AisaCCS07. Suppose a sender want to encrypt a message to a receiver, The adversary can replace the receiver's public key by any other' s public key. Although the adversary cannot decrypt the ciphertext, the receiver cannot decrypt too while the sender can not be aware of this. This is similar to Denial of Service (DoS) Attack in the way that the attacker cannot gain any secret information but precluding others from getting the normal service. Liu and Au [2, 3] propose a new paradigm called Self-Generated-Certificate Public Key Cryptography (SGC-PKC) to defend the above attack while preserving all advantages of Certificateless Public Key Cryptography. Similar to CL-PKC, every user is given a partial secret key by the KGC and generates his own secret key and corresponding public key. In addition, he also needs to generate a certificate using his own secret key. The purpose of this self-generated certificate [11] is to bind the identity (or personal information) and the public key together. The main difference is that, it can be verified by using the user's identity and public key only and does not require any trusted party. It is implicitly included in the user's public key. If Carol uses her public key to replace Alice's public key (or certificate), Bob can be aware of this and he may ask Alice to send him again her public key for the encryption.

Liu and Au proposed the first SGC-PKE scheme in [2, 3], which defends the DoD attack that exists in CL-PKE. In PKC07, Lai and Kou proposed a self-generated-certificate public key encryption without pairing scheme, to the best of our knowledge, this is the second SGC-PKE scheme[1]. In this paper, we show that this scheme cannot resist a man-in-the-middle attack. We further point out the reason for successfully attacking is binding the user's secret key with the multiply of partial public key from KGC and user's public key instead of binding with partial public key from KGC and user's public key independently.

We organize the paper as following. In section 2, we give BSS's definition for CL-PKE and LK's definition for CL-PKE. In section 3, we propose two security models–Type A model and Type B model–for CL-PKE (As an independent interesting for cryptographic community, we thoroughly revisit certificateless public cryptography keygeneration algorithm and security model in appendix). In section 4, we give LK's SGC-PKE scheme and their security model in [1]. In section 5, we point out this model falls in type A model but their scheme falls in type B model, so the original model is not sufficient, and we give an attack to the LK scheme in the type B model. In section 6, we propose a rescue scheme and prove its security in the random oracle model. We give our concluding remarks in section 7.

2 Definitions

In this section we first introduce BSS's definition for CL-PKE, Next, we recall the LK's definition for CL-PKE .Last we compare the two definitions.

2.1 BSS's Definition for Certifiateless Public Key Encryption

Definition 1. (BSS's Certificateless Public Key Encryption) *A generic CLPKE (Certificateless Public Key Encryption) scheme, consists of the following algorithms.*

- Setup: The Key Generation Center (KGC) runs this algorithm to generate a common parameter $params$ and a master key $masterKey$. Note that $params$ is given to all interested parties. We write $(params, masterKey) = Setup()$.
- PartialKeyExtract:Taking $params$, $masterKey$ and an identity ID received from a user as input, the KGC runs this algorithm to generate a par tial private key D_{ID} and a partial public key P_{ID}. We write $(P_{ID}, D_{ID}) = PartialKeyExtract(params, masterKey, ID)$.
- SetSecretValue: Taking $params$ and ID as input, the user runs this algorithm to generate a secret value s_{ID}. We write $s_{ID} = SetSecretValue(params, ID)$.
- SetPrivateKey: Taking $params$, D_{ID} and s_{ID} as input, the user runs this algo rithm to generate a private key SK_{ID}. We write $SK_{ID} = SetPrivateKey(pa rams, D_{ID}, s_{ID})$.
- SetPublicKey: Taking $params$, P_{ID}, $s_{ID}$and ID as input, the user runs this algorithm to generate a public key PK_{ID}. We write $PK_{ID} = SetPublicKey(params, P_{ID}, s_{ID}, ID)$.
- Encrypt: Taking $params$, ID, PK_{ID}, and a plaintext message M as input, a sender runs this algorithm to create a ciphertext C. We write $C = Encrypt(params, ID, PK_{ID}, M)$.
- Decrypt: Taking $params$, SK_{ID} and the ciphertext C as input, the user as a recipient runs this algorithm to get a decryption , which is either a plaintext message or a "Reject" message. We write $= Decrypt(params, SK_{ID}, C)$.

2.2 LK's Definition for Certifiateless Public Key Encryption

Definition 2. (LK's Certificateless Public Key Encryption) *A generic Certificateless Public Key Encryption scheme, denoted by CLPKE, consists of the following algorithms:*

- Setup: is a probabilistic polynomial time (PPT) algorithms run by a Key Generation Center (KGC), given a security parameter k as input, outputs a randomly chosen master secret mk and a list of public parameter param.We write $(mk, param) = Setup(k)$.
- UserKeyGeneration: is PPT algorithm, run by the user, given a list of public parameters $param$ as inputs, outputs a secret key sk and a public key pk. We write $(sk, pk) = UserKeyGeneration(param)$.

- PartialKeyExtract: Taking $param, mk$, a user's identity ID and pk received from the user, the KGC runs this PPT algorithm to generate a partial private key D_{ID} and a partial public key P_{ID}. We write $(P_{ID}, D_{ID}) = PartialKeyExtract(param, mk, ID, pk)$.
- SetPrivateKey: Taking $param$, D_{ID} and sk as input, the user runs this PPT algorithm to generate a private key SK_{ID}. We write $SK_{ID} = SetPrivateKey(param, D_{ID}, sk)$.
- SetPublicKey: Taking $param$, P_{ID} and pk as input, the user runs this PPT algorithm to generate a public key PK_{ID}. We write PK_{ID}=$SetPublicKey(param, P_{ID}, pk)$.
- Encrypt: Taking a plaintext M, list of parameters $param$, a receiver's identity ID and PK_{ID} as inputs, a sender runs this PPT algorithm to create a ciphertext C. We write $C = Encrypt(param, ID, PK_{ID}, M)$.
- Decrypt: Taking param, SK_{ID}, the ciphertext C as inputs, the user as a recipient runs this deterministic algorithm to get a decryption m, which is either a plaintext message a "Reject"message. We write m=$Decrypt\ (param, SK_{ID}, C)$.

Comparison: The Setup,SetPrivateKey, Encrypt and Decrypt algorithm are same in [6] as in [1].The SetSecretValue in [6] plays the same role as UserKeyGeneration in [1], the difference lies in the former just outputing a user's local secret value but the latter outputing also a user's local public key.In [6] SetSecretValue can run independently with PartialKeyExtract while in [1]UserKeyGeneration must run precede with PartialKeyExtract as PartialKeyExtract must include user's local public key pk as its input.In [6],SetPublicKey does not include user's local public key as input while In [1]this is required.

3 Two Kinds of Security Model for CL-PKE

In this section, we give two kinds of security model for CL-PKE, one for the BSS's CL-PKE ,we denote it as Type A model; the other for the LK's CL-PKE, we denote it as Type B model.According to the original scheme in [4], there are two types of adversaries. Type I adversary does not have the KGC's mater secret key but it can replace public keys of arbitrary identities with other public keys of its own choices. It can also obtain partial and full secret keys of arbitrary identities.Type II adversary know the master secret key (hence it can compute partial secret key by itself). It is still allowed to obtain full secret key for arbitrary identities but is not allowed to replace public keys at any time.So in both security models there are two kinds of adversaries: Type I adversary and Type II adversary.

3.1 Type A security model for BSS's CL-PKE

Definition 3. (Type A security model for IND-CCA Security for BSS's CL-PKE) *A Certificateless Public Key Encryption scheme CLPKE is IND-CCA secure if no PPT adversary A of Type I or Type II has anon-negligible advantage in the following game played against the challenger:*

1. The challenger takes a security parameter k and runs the Setup algorithm. It gives A the resulting system parameters $param$. If A is of Type I, the challenger keeps the master secret key mk to itself, otherwise, it gives mk to A.
2. A is given access to the following oracles:
   - Public-Key-Request-Oracle: on input a user's identity ID, it computes$(P_{ID}, D_{ID}) = PartialKeyExtract(params, masterKey, ID)$and $s_{ID} = SetSecretValue(params, ID)$. It then computes $PK_{ID} = SetPublicKey(params, P_{ID}, s_{ID}, ID)$and returns it to A.
   - Partial-Key-Extract-Oracle: on input a user's identity ID , it computes $(P_{ID}, D_{ID}) = PartialKeyExtract(params, masterKey, ID)$ and returns it to A. (Note that it is only useful to Type I adversary.)
   - Private-Key-Request-Oracle: on input a user's identity ID, it computes $s_{ID} = SetSecretValue(params, ID)$ and $(P_{ID}, D_{ID}) = PartialKeyExtract(params, masterKey, ID)$. It then computes $SK_{ID} = SetPrivateKey(params, D_{ID}, s_{ID})$and returns it to A. it outputs "Reject". if the user's public key has been replaced (in the case of Type I adversary.)
   - Public-Key-Replace-Oracle: (For Type I adversary only) on input identity and a valid public key, it replaces the associated user's public key with the new one.
   - Decryption-Oracle: on input a ciphertext and an identity, returns the decrypted plaintext using the private key corresponding to the current value of the public key associated with the identity of the user.
3. After making oracle queries a polynomial times, A outputs and submits two message (M_0, M_1), together with an identity ID^* of uncorrupted secret key to the challenger. The challenger picks a random bit $b \in 0, 1$ and computes C^*, the encryption of M_b under the current public key PK_{ID^*} for ID^*. If the output of the encryption is "Invalid ciphertext", then A immediately losses the game. Otherwise C^* is delivered to A.
4. A makes a new sequence of queries.
5. A outputs a bit b'. It wins if $b' = b$ and fulfills the following conditions:
   - At any time, ID^* has not been submitted to Private-Key-Request-Oracle.
   - In Step (4), C^* has not been submitted to Decryption-Oracle for the combination (ID^*, PK_{ID^*}) under which M_b was encrypted.
   - If it is Type I, ID^* has not been submitted to both Public-Key-Replace-Oracle before Step (3) and Partial-Key-Extract-Oracle at some step.

Define the guessing advantage of A as

$$Succ_{CLE}^{IND-CCA}(A) = \mid Pr(b = b') - \frac{1}{2} \mid$$

3.2 Type B security model for LK's CL-PKE

Definition 4. (Type B security model for IND-CCA Security for LK's CL-PKE) *A Certificateless Public Key Encryption scheme CLPKE is IND-CCA secure if no PPT adversary A of Type I or Type II has anon-negligible advantage in the following game played against the challenger:*

1. The challenger takes a security parameter k and runs the Setup algorithm. It gives A the resulting system parameters $param$. If A is of Type I, the challenger keeps the master secret key mk to itself, otherwise, it gives mk to A.
2. A is given access to the following oracles:
   - Public-Key-Request-Oracle: on input a user's identity ID, it computes $(sk, pk) = UserKeyGeneration(param)$ and $(P_{ID}, D_{ID}) = PartialKey Extract(param, mk, ID, pk)$. It then computes $PK_{ID} = SetPublicKey (param, P_{ID}, pk)$ and returns it to A.
   - Partial-Key-Extract-Oracle: on input a user's identity ID and pk, it computes $(P_{ID}, D_{ID}) = PartialKeyExtract\ (param, mk, ID, pk)$ and returns it to A. (Note that it is only useful to Type I adversary.)
   - Private-Key-Request-Oracle: on input a user's identity ID, it computes (sk, pk)=$UserKeyGeneration(param)$and $(P_{ID}, D_{ID}) = PartialKeyEx tract(param, mk, ID, pk)$. It then computes SK_{ID} =$SetPrivateKey(pa ram, D_{ID}, sk)$ and returns it to A. it outputs "Reject". if the user's public key has been replaced (in the case of Type I adversary.)
   - Public-Key-Replace-Oracle: (For Type I adversary only) on input identity and a valid public key, it replaces the associated user's public key with the new one.
   - Decryption-Oracle: on input a ciphertext and an identity, returns the decrypted plaintext using the private key corresponding to the current value of the public key associated with the identity of the user.
3. After making oracle queries a polynomial times, A outputs and submits two message (M_0, M_1), together with an identity ID^* of uncorrupted secret key to the challenger. The challenger picks a random bit $b \in 0, 1$ and computes C^*, the encryption of M_b under the current public key PK_{ID^*} for ID^*. If the output of the encryption is "Invalid ciphertext", then A immediately losses the game. Otherwise C^* is delivered to A.
4. A makes a new sequence of queries.
5. A outputs a bit b'. It wins if b' =b and fulfills the following conditions:
   - At any time, ID^* has not been submitted to Private-Key-Request-Oracle.
   - In Step (4), C^* has not been submitted to Decryption-Oracle for the combination $(ID*, PK_{ID^*)}$ under which M_b was encrypted.
   - If it is Type I, $(ID^*, PK_{ID^*)}$ has not been submitted to both Public-Key-Replace-Oracle before Step (3) and Partial-Key-Extract-Oracle at some step.

Define the guessing advantage of A as

$$Succ_{CLE}^{IND-CCA}(A) =\mid Pr(b = b') - \frac{1}{2} \mid$$

3.3 The difference Between Type A security model and Type B Security model

We note that Type A security model denies access to PartialKeyExtract -Oracle with any input containing ID^*. But in Type B security model for CL-PKE, the

adversary can access PartialKeyExtract-Oracle with input $(ID^*, PK_{ID^{"}})$ where $PK_{ID^{"}} \neq PK_{ID^*}$.

The LK's IND-CCA2 model [1] denied access to PartialKeyExtract -Oracle with any input containing ID^*, we consider this is unreasonable. This restriction is applied only to CL-PKE scheme with the property of PartialKeyExtract algorithm running independently with UserKeyGeneration algorithm. But LK's CL-PKE has no this property.This is the reason why the following attack can succeed.

Actually, there are many different definitions and security models for CL-PKC.We thoroughly revisiting the current literature [6-10,14-21]in CL-PKC (We suggest Dent's paper [15] as a good reference for current CL research) and we give a new framework for CL-PKC's definitions and security models, which maybe be interesting for the cryptographic community, we give it in appendix.

4 LK's Definition and Security Model for SGC-PKE and the Concrete Scheme

4.1 LK's Definition and Security Model for SGC-PKE

The definition of SGC Encryption is same as the definition of CL-encryption given in Definition 1, except for SetPublicKey in which the user generates a certificate using his own secret key.

For security, in addition to IND-CCA, we require the scheme to be DoD-Free, which is formally defined as follow as a game played between the challenger and a PPT adversary (DoD Adversary), which has the same power of a Type I adversary defined in CL-encryption.

Definition 5. (DoD-Free Security) *A SGC Encryption scheme is DoD-Free secure if no PPT adversary A has a non-negligible advantage in the following game played against the challenger:*

1. The challenger takes a security parameter k and runs the Setup algorithm. It gives A the resulting systems parameters $param$. The challenger keeps the master secret key mk to itself.
2. A is given access to Public-Key-Request-Oracle , Partial-Key-Extract-Oracle, Private-Key-Request-Oracle and Public-Key-Replace-Oracle.
3. After making oracle queries a polynomial times, A outputs a message M^*, together with an identity ID^* to the challenger. The challenger computes C^*, the encryption of M^* under the current public key PK_{ID^*} for ID^*. If the output of the encryption is "Invalid ciphertext", then A immediately losses the game. Otherwise it outputs C^*.
4. A wins if the following conditions are fulfilled:
   - The output of the encryption in Step (3) is not "Invalid ciphertext"
   - $Decrypt(param, SK_{ID^*}, C^*) = M^*$.
   - At any time, ID^* has not been submitted to Partial-Key-Extract-Oracle.

Define the advantage of A as

$$Succ_{SGC}^{DoD-Free}(A) = Pr(A\ Wins)$$

4.2 LK's SGC-PKE Scheme

1. Setup: Generate two large primes p and q such that $q|p-1$. Pick a generator g of $\mathbb{Z}_{q^*}$. Pick $x \in \mathbb{Z}_{q^*}$ uniformly at random and compute $y = g^x$. Choose hash functions $H_1 : \{0,1\}^* \times \mathbb{Z}_p^* \rightarrow \mathbb{Z}_q^*$, $H_2 : \{0,1\}^{l_0} \times \{0,1\}^{l_1} \rightarrow \mathbb{Z}_q^*$ and $H_3 : \mathbb{Z}_p^* \rightarrow \{0,1\}^l$, where $l = l_0 + l_1 \in N$. Return $param = (p,q,g,y,H_1, H_2, H_3)$ and $mk = x$.
2. UserKeyGeneration: Pick $z \in \mathbb{Z}_q^*$ at random and compute $u = g^z$, Return(sk,pk) $=(z,u)$.
3. PartialKeyExtract: Taking $param$, mk, ID, pk as input, it outputs (P_{ID}, D_{ID}) $= (w = g^s, t = s + xH_1(ID, w * pk) = s + xH_1(ID, wu))$.
4. SetPrivateKey: outputs $SK_{ID} = sk + D_{ID} = z + t$.
5. SetPublicKey: Except for taking $param$, P_{ID} and pk as input, it includes ID and SK_{ID} as inputs. Chooses a new hash function $H_0 : \{0,1\}^* \times \mathbb{Z}_p^* \times \mathbb{Z}_p^* \times \mathbb{Z}_p^* \rightarrow \mathbb{Z}_q^*$,then computes $PK_{ID}^1 = pk * P_{ID} = uw$ and $PK_{ID}^2 = pk * P_{ID} * y^{H_1(ID,pk*P_{ID})} = uwy^{H_1(ID,uw)} = g^{z+t} = g^{SK_{ID}}$. Next, it does the following performances to sign the user's identity ID and PK_{ID}^1,PK_{ID}^2 using the user's private key SK_{ID} and Schnnor's signature scheme [13]. (1) Choose a random $r \in \mathbb{Z}_{q^*}$, (2) compute $R = g^r mod\ p$;(3) set the signature to be (R, s), where $s = r + SK_{ID} * H_0(ID, PK_{ID}^1, PK_{ID}^2, R)$. Finally, returns $PK_{ID} = (PK_{ID}^1, PK_{ID}^2, (R, s))$.
6. Encrypt: let $PK_{ID} = (PK_{ID}^1, PK_{ID}^2, (R, s))$.If $PK_{ID}^2 \neq PK_{ID}^1 * y^{H_1(ID,PK_{ID}^1)}$ or $g^s \neq R * (PK_{ID}^2)^{H_0(ID,PK_{ID}^1,PK_{ID}^2,R)}$, it returns"Reject", else pick $o \in \{0,1\}^{l_1}$ at random, and compute $r = H_2(M, o)$.Compute $C = (C_1, C_2)$ such that $C_1 = g^r, C_2 = H_3((uwy^{H_1(ID,uw)})^r) \oplus s(M \parallel o) = H_3((PK_{ID}^2)^r) \oplus (M \parallel o)$.
7. Decrypt: Parse C as (C_1, C_2) and SK_{ID} as (z, t).Compute $M \parallel o = H_3((C_1)^{z+t}$ $\oplus C_2$. If $g^{H_1(M,o)} = C_1$, return$M$ .Else return "Reject".

5 Attack on LK's SGC-PKE Scheme

The LK's SGC-PKE definition and security model comes from the definition and security model for CL-PKE. So the IND-CCA2 security for LK's SGC-PKE suffering from not allowing PartialKeyExtract-Oracle querying with $(ID^*, PK_{ID^"})$ $\neq (ID^*, PK_{ID^*})$.Similarly the Dod-Free security for LK's SGC-PKE suffering from this too.

That is, LK's SGC-PKE scheme falls in Type B security model, but their security model falls in Type A security model and they ananlysis their scheme in Type A model, this is why their secuurity ananlysis is not sufficient and our attack can work.

Note that in Lai and Kou's SGC-PKE scheme, the SK_{ID} binds with the multiply of partial public key from KGC and user's self-generated public key instead of with partial public key from KGC and user's self-generated public key independently. We can explore this shortcoming to give a *man-in-the-middle attack*. We attack the target user when he generates his privatekey and public key.

First the attacker corrupts the target ID and gets his key $(sk,pk)=(z,u)$.Then the attacker pretends to be a user with identity ID for KGC and pretends to be the KGC for the target user. He can always control the target user's privatekey be equal to his privatekey which is definitely insecure.

1. The attacker gets the target ID's user's key $(sk,pk)=(z,u)$ where $u = g^z$ by corruption the target ID or via UserKeyGeneration Oracle.
2. The attacker generates his own key$(sk',pk')=(z',u')$ where $u' = g^{z'}$.
3. The attacker pretends to be the target ID to the KGC, and then he gets the partial key $(P_{ID}, D_{ID}) = (w = g^s, t = s + xH_1(ID, w * pk') = s + xH_1(ID, wu'))$ via the PartialKey Extract Oracle $(param, mk, ID, pk')$.
4. The attacker computes his private key $SK_{Attacker} = sk + D_{ID} = z' + t$.
5. The attacker pretends to be the KGC to the target ID, and he sets $(P^*_{ID}, D^*_{ID}) = (w^* = \frac{wu'}{g^z}, SK_{Attacker} - z)$. Send it as the result of PartialKeyExtract (param, mk, ID, pk) to the target ID.
6. The target ID checks whether equation $g^{D^*_{ID}} = w^* * y^{H_1(ID,w^**PK)}$ holds. If it holds, he computes his own private key $SK_{Target} = sk + D^*_{ID} = z + SK_{Attacker} - z = SK_{Attacker}$,else "reject".
7. Thus the attacker can control the target ID's Privatekey be equal to his Privatekey and decrypt all the ciphertext sends to the target ID.

First we verify the equation $g^{D^*_{ID}} = w^* * y^{H_1(ID,w^**PK)}$ always holds.

$$\begin{aligned} \mathrm{g}^{D^*_{ID}} &= g^{SK_{Attacker}-z} \\ &= g^{z'+t-z} \\ &= g^{z'+s+xH_1(ID,wu')-z} \\ &= g^{z'+s+xH_1(ID,w^**pk)-z} \\ &= \frac{wu'}{g^z} * y^{H_1(ID,w^**pk)} \\ &= w * y^{H_1(ID,w^**pk)} \end{aligned}$$

In our attack, the attacker can access two oracles: Usergeneration Oracle and PartialKey Extract Oracle. Assuming UserKeyGeneration Oracle's output is (sk, pk), we must note that query to thePartialKey Extract Oracle is $(param, mk, ID, pk')$ instead of $(param, mk, ID, pk)$. Otherwise our attack is a trivial attack.

6 A Rescue Scheme and Its Security Proof

6.1 A Rescue Scheme

Actually, we just need give little change to the Lai and Kou's scheme to resist this attack. In the new scheme, the SK_{ID} binds with partial public key from KGC and user's self-generated public key independently, so the man-in-the-middle attack can not work any more. Following is the rescue scheme.

1. Setup: Generate two large primes p and q such that $q|p-1$. Pick a generator g of $\mathbb{Z}_{q^*}$. Pick $x \in \mathbb{Z}_{q^*}$ uniformly at random and compute $y = g^x$. Choose hash functions $H_1 : \{0,1\}^* \times \mathbb{Z}_p^* \times \mathbb{Z}_p^* \rightarrow \mathbb{Z}_q^*$, $H_2 : \{0,1\}^{l_0} \times \{0,1\}^{l_1} \rightarrow \mathbb{Z}_q^*$ and $H_3 : \mathbb{Z}_p^* \rightarrow \{0,1\}^l$, where $l = l_0 + l_1 \in N$. Return $param = (p,q,g,y,H_1, H_2, H_3)$ and $mk = x$.
2. UserKeyGeneration: Pick $z \in \mathbb{Z}_q^*$ at random and compute $u = g^z$, Return$(sk,pk) =(z,u)$.
3. PartialKeyExtract: Taking $param, mk, ID, pk$ as input, it outputs $(P_{ID}, D_{ID}) = (w = g^s, t = s + xH_1(ID, w, pk) = s + xH_1(ID, w, u))$.
4. SetPrivateKey: outputs $SK_{ID} = sk + D_{ID} = z + t$.
5. SetPublicKey: Except for taking $param$, P_{ID} and pk as input, it includes ID and SK_{ID} as inputs. Chooses a new hash function $H_0 : \{0,1\}^* \times \mathbb{Z}_p^* \times \mathbb{Z}_p^* \times \mathbb{Z}_p^* \rightarrow \mathbb{Z}_q^*$,then computes $PK_{ID}^1 = (pk, P_{ID}) = (u, w)$ and $PK_{ID}^2 = pk * P_{ID} * y^{H_1(ID,pk,P_{ID})} = uwy^{H_1(ID,u,w)} = g^{z+t} = g^{SK_{ID}}$. Next, it does the following performances to sign the user's identity ID and PK_{ID}^1,PK_{ID}^2 using the user's private key SK_{ID} and Schnnor's signature scheme. (1) Choose a random $r \in \mathbb{Z}_{q^*}$, (2) compute $R = g^r mod\ p$;(3) set the signature to be (R, s), where $s = r + SK_{ID} * H_0(ID, PK_{ID}^1, PK_{ID}^2, R)$. Finally, returns $PK_{ID} = (PK_{ID}^1, PK_{ID}^2, (R, s))$.
6. Encrypt: let $PK_{ID} = (PK_{ID}^1, PK_{ID}^2, (R, s))$.parse PK_{ID}^1 as(u, w). If $PK_{ID}^2 \neq PK_{ID}^1 * y^{H_1(ID,u,w)}$ or $g^s \neq R*(PK_{ID}^2)^{H_0(ID,PK_{ID}^1,PK_{ID}^2,R)}$, it returns"Reject", else pick $o \in \{0,1\}^{l_1}$ at random, and compute $r = H_2(M, o)$.Compute $C = (C_1, C_2)$ such that $C_1 = g^r, C_2 = H_3((uwy^{H_1(ID,uw)})^r) \oplus (M \parallel o) = H_3((PK_{ID}^2)^r) \oplus (M \parallel o)$.
7. Decrypt: Parse C as (C_1, C_2) and SK_{ID} as (z, t).Compute $M \parallel o = H_3((C_1)^{z+t} \oplus C_2$. If $g^{H_1(M,o)} = C_1$, return$M$ .Else return "Reject".

6.2 Security Analysis

The security proofs of our scheme are similar to the BSS's CL-PKE [6]. Basically, the main idea of the security proofs given in this section is to have the CDH attacker B simulates the environment of the Type I and Type II attackers A_I and A_{II} in Type B security model respectively until it can compute a Diffie-Hellman key g^{ab} of g^a and g^b using the ability of A_I and A_{II} .

For the attacker A_I, B sets g^a as a part of the challenge ciphertext and g^b as a KGC's public key. On the other hand, for the attacker A_{II}, B sets g^a as a part of the challenge ciphertext but uses g^b to generate a public key associated with the challenge identity.

The following two theorems show that our scheme is IND-CCA secure in the random oracle, assuming that the CDH problem is intractable. We will give proofs of Theorm1 due to our attacker is a Type I attacker. We omit the proof of Theorm2 due to its similarity with Theorm2 in [1] .

Theorem 1. *The SGC-PKE scheme is* $(t, q_{H_1}, q_{H_2}, q_{H_3}, q_{par}, q_{pub}, q_{prv}, q_D, \epsilon)$ *- IND-CCA secure against the Type I attacker* A_I *in the random oracle assuming the CDH problem is* (t', ϵ')*-intractable and the Schnorr's signature scheme is*

$(t^{''}, \epsilon^{''})$ *secure against the adaptively chosen message attack in the random oracle model , where* $\epsilon^{'} > \frac{1}{q_{H_2}}(\frac{2(\epsilon-\epsilon^{''})}{e(q_{prv}+1)} - \frac{q_{H_2}}{2^{l_1}} - \frac{q_D q_{H_2}}{2^{l_1}} - \frac{q_D}{q})$ *and* $t^{'} > t + t^{''} + 2(q_{par} + 2q_{pub} + q_{prv})t_{ex} + 10q_D q_{H_2} q_{H_3} t_{ex} + 3t_{ex}$ *where* t_{ex} *denotes the time for computing exponentiation in* Z_{p^*}.

Proof. Let A_I be an IND-SGCPKE-CCA Type I attacker. We show that using A_I, one can construct an attacker B that can solve the CDH problem. Suppose that B is given $(p, q, g, g^a, g^b$ as an instance of the CDH problem. B can simulate the Challenger's execution of each phase of IND-SGCPKE-CCA game for A_I as follows.

[**Simulation before challenge**] B sets $y = g^b$ and gives A_I $(p, q, g, y, H_0, H_1, H_2, H_3)$ as params, where $(H_0, H_1, H_2, H_3)$are random oracles controlled by B as follows.

On receiving a query (ID, ω, u) *to* H_1:

1. If (ID, ω, u, e) exists in H_1List, return e as answer.

2. Otherwise, pick $e \in Z_q^*$at random, add (ID, ω, u, e) to H_1List and return e as answer.

On receiving a query $(M,)$ *to* H_2 :

1. If $(M, , r)$ exists in H_2List, return r as answer.

2.Otherwise, pick $r \in Z_q^*$ at random, add $(M, , r)$ to H_2List and return r as answer.

On receiving a query k *to* H_3:

1. If (k, R) exists in H_3List, return R as answer.

2. Otherwise, pick $R \in \{0,1\}^l$ at random, add (k, R) to H_3List and return R as answer.

On receiving a partial key extraction query ((ID, u) *, "partial key extract")*

1. If $((ID, u), \omega, t)$ exists in $PartialKeyList$, return (ω, t) as answer.

2. Otherwise, do the following:

(a) Pick $t, e \in Z_q^*$ at random and compute $\omega = \frac{g^t y^{-e}}{u}$; add to H_1List (That is, e is defined to be $H_1(ID, \omega, u)$.) and $((ID, u), \omega, t)$ to $PartialKeyList$; return (ω, t) as answer.

Note from the above simulation that we have $u\omega y^{H_1(ID,PK,P_{ID}} = u * \frac{g^t y^{-e}}{u} * y^e = g^t$, which holds in the real attack too.

On receiving a public key request query (ID,"public key request")

1. If $(ID, PK_{ID}^1, PK_{ID}^2, (R, \sigma))$ exists in $PublicKeyList$, return $(PK_{ID}^1, PK_{ID}^2, (R, \sigma))$ as answer.

2. Otherwise, pick $coin \in \{0, 1\}$, so that $Pr[coin = 0] = \delta$ (δ will be determined later).

3. If $coin = 0$, do the following:

(a) If $((ID, u), \omega, t)$ exists in $PartialKeyList$, pick $z \in Z_q^*$ at random and compute $u = g^z$; compute $sk_{ID} = z + t$ add $(ID, sk_{ID}$ to $PrivateKeyList$; Choose a random $r^{'} \in Z_q^*$, compute $R = g^r mod p$,set the signature to be (R, σ), where $\sigma = r^{'} + sk_{ID} * H_0(ID, PK_{ID}^1, PK_{ID}^2, R)$(On receiving a query$(ID, PK_{ID}^1, PK_{ID}^2, R)$ to H_0: 1. If $(coin = 0, (ID, PK_{ID}^1, PK_{ID}^2, R), h)$exists in H_0List, return h as answer. 2. Otherwise, pick $h \in Z_q^*$

at random, add $(coin = 0, (ID, PK^1_{ID}, PK^2_{ID}, R), h)$ to H_0List and return h as answer.). Let $PK^1_{ID} = (u, \omega)$, $PK^2_{ID} = g^{sk_{ID}}$. Finally, returns $PK_{ID} = (ID, PK^1_{ID}, PK^2_{ID}, R)$ as answer.

(b) Otherwise, run the above simulation algorithm for partial key extraction taking $r(ID, u)$ as input to get a partial key (w, t); pick $z \in Z^*_q$ at random and compute $u = g^z$; compute $sk_{ID} = z + t$ add $(ID, sk_{ID}$ to $PrivateKeyList$; Choose a random $r^{'} \in Z^*_q$, compute $R = g^r modp$,set the signature to be (R, σ), where $\sigma = r^{'} + sk_{ID} * H_0(ID, PK^1_{ID}, PK^2_{ID}, R)$($B$ simulate the H_0 as the above simulation). Let $PK^1_{ID} = (u, \omega)$, $PK^2_{ID} = g^{sk_{ID}}$. Finally, returns $PK_{ID} = (ID, PK^1_{ID}, PK^2_{ID}, R)$ as answer.

4. Otherwise (if $coin = 1$), pick $s, z \in Z^*_q$ and compute $w = g^s$ and $u = g^z$; compute $PK^1_{ID} = (u, \omega)$ and $PK^2_{ID} = u\omega y^{H_1(ID,PK,P_{ID}}$; pick $\sigma, j \in Z^*_q$, compute $R = \frac{g^\sigma}{(PK^2_{ID}}{}^{j)}$(Let $j = H_0(ID, PK^1_{ID}, PK^2_{ID}, R)$, add $(coin = 1, H_0(ID, PK^1_{ID}, PK^2_{ID}, R), j)$to H_0List), add $(ID, ?, z, s)$ to $PrivateKeyList$ and $((ID, PK^1_{ID}, PK^2_{ID}, (R, \sigma)), coin)$ to $PublicKeyList$; return $(PK^1_{ID}, PK^2_{ID}$, $(R, \sigma))$as answer.

On receiving a private key extraction query (ID," privatekeyextract")

1. Run the above simulation algorithm for public key request taking ID as input to get a tuple $((ID, PK^1_{ID}, PK^2_{ID}, (R, \sigma)), coin) \in PublicKeyList$.

2. If $coin = 0$, search $PrivateKeyList$ for a tuple $(ID, sk_{ID})$and return $SK_{ID} = sk_{ID}$ as answer.

3. Otherwise, return "Abort" and terminate.

On receiving a decryption query $((ID, PK^1_{ID}, PK^2_{ID}, (R, \sigma)), C,$"$decryption$"), where $C = (C_1, C_2)$

1. Search $PublicKeyList$ for a tuple $((ID, PK^1_{ID}, PK^2_{ID}, (R, \sigma)), coin)$.

2. If such a tuple exists and $coin = 0$

(a) Search $PrivateKeyList$ for a tuple (ID, sk_{ID}). (Note that from the simulation of public key request, (ID, sk_{ID}) must exist in $PrivateKeyList$ as long as one can find $(ID, (ID, PK^1_{ID}, PK^2_{ID}, (R, \sigma)), coin)$ with coin =0 in PublicKeyList.

(b) Parses PK_{ID} as $(PK^1_{ID}, PK^2_{ID}, (R, \sigma))$, parse $\mathrm{PK}^1_{ID} as$(u,ω) . If $PK^2_{ID} \neq PK^1_{ID} * y^{H_1(ID,u,w)}$ or $g^s \neq R * (PK^2_{ID})^{H_0(ID,PK^1_{ID},PK^2_{ID},R)}$, it returns"Reject", Parse C as (C_1, C_2) and SK_{ID} as (z, t).Compute $M \parallel = H_3((C_1)^{z+t} \oplus C_2$. If $g^{H_1(M,)} = C_1$, return$M$ .Else return "Reject".

3. Else if such a tuple exists and $coin = 1$

(a) Run the above simulation algorithm for H_1 to get a tuple (ID, ω, u, e).

(b) If there exist$(M, , r) \in H_2List$ and $(k, R) \in H_3List$ such that $c_1 = g^r, c_2 = R \oplus M \parallel, k = u\omega y^{er}$ Return M and Reject otherwise. We remark that the pair$(M, , r)$ that satisfies the above condition uniquely exists in H_2List as the encryption function is injective with respect to (ID, ω, u, e).

4. Else if such a tuple does not exist (This is the case when the public key of a target user is replaced by A_I, but from the SGC-PKE definition, this means breaking Schnnor's signature scheme).

[**Simulation on challenge**] B answers A_I's queries as follows:

On receiving a challenge query $(ID^*, (M_0, M_1))$:

1. Run the above simulation algorithm for public key request taking ID as input to get a tuple $((ID^*, {PK^1_{ID}}^*, {PK^2_{ID}}^*, (R^*, \sigma^*)), coin) \in PublicKeyList$.

2. If $coin = 0$ return "Abort" and terminate

3. Otherwise, parse ${PK^1_{ID}}^*$ as (u^*, ω^*) . If $PK^2_{ID}* \neq {PK^1_{ID}}^* * y^{H_1(ID^*, u^*, w^*)}$ or $g^{\sigma *} \neq R^* * (PK^2_{ID}{}^*)^{H_0(ID^*, PK^1_{ID}{}^*, PK^2_{ID}{}^*, R^*)}$, it returns"Reject"(This case can only happen with negligible probability).else do the following:

(a)Search $PrivateKeyList$ for a tuple $(ID^*, ?, z^*, s^*)$. In this case, we know that $\omega^* = g^{s^*}$ and $u^* = g^{z*}$.

(b) Pick $^* \in \{0,1\}^{l_1}$,$c_2{}^* \in \{0,1\}^l$ and $\in \{0,1\}$random.

(c) Set $c_1^* = g^a$, $\gamma_{ID^*} = u^* \omega^* y^{e^*}$ and $e^* = H_1(ID^*, u^*, \omega^*)$

(d) Define $a = H_2(M,^*)$ and $H_3(\gamma^a_{ID^*}) = c_2^* \oplus (M^*)$ Note that B does not know "a")

4. Return $c^* = (c_1^*, c_2^*)$ as a target ciphertext.

Note that by the construction given above, $\mathrm{c}_2^* = \mathrm{H}_3(\gamma^a_{ID^*}) \oplus (M^*)$.

[**Simulation after challenge**] In this phase, B answers A_I's queries in the same way as before except the natural constraints.

[**Guess**]When A outputs its , B returns the set $S = \{(\frac{k_i}{g^{as^*} g^{az^*}})^{\frac{1}{e^*}} | \mathrm{k}_i$ are the queries toH_3 for $i \in [1, q_{H_3}]$such that $e^* = H_1(ID^*, u^*, \omega^*).\}$

[**Analysis**]We first evaluate the simulations of the random oracles given above. From the construction of H_1, it is clear that the simulation of H_1 is perfect. As long as A_I does not query $(M, ^*)$ to H_2 nor $\left(u^* \omega^* y^{e^*}\right)^a$ to H_3, the simulations of H_2 and H_3 are perfect. By $Ask^*_{H_3}$we denote the event $\left(u^* \omega^* y^{e^*}\right)^a$ has been queried to H_3. Also, by $Ask^*_{H_2}$ we denote the event$(M, ^*)$ has queried to H_2.

Next, one can notice that the simulated target ciphertext is identically distributed as the real one from the construction.

Now, we evaluate the simulation of the decryption oracle. If a public key PK_{ID} has not been produced under $coin = 1$, the simulation is perfect as B knows the private key SK_{ID} corresponding to PK .Otherwise, simulation errors may occur while B running the decryption oracle simulator specified above. However, these errors are not significant as shown below: Suppose that $(ID, {PK^1_{ID}}^*, {PK^2_{ID}}^*, (R^*, \sigma^*), C)$ has been issued as a valid decryption que ry. Even if C is valid, there is a possibility C can be produced without querying $(u \omega y^e)^r$ toH_3, where $e = H_1(ID, \omega, u)$ and $r = H_2(M,)$ Let **Valid** be an event that C is valid. Let $AskH_3$and $AskH_2$ respectively be events that $(u \omega y^e)^r$ has been queried to H_3 and $(M,)$has been queried to H_2. We then have

$$
\begin{aligned}
Pr[Valid \mid \neg AskH_3] &\leq Pr[Valid \wedge AskH_2 \mid \neg AskH_3] + Pr[Valid \wedge \neg AskH_2 \mid \neg AskH_3] \\
&\leq Pr[AskH_2 \mid \neg AskH_3] + Pr[Valid \mid \neg AskH_2 \wedge \neg AskH_3] \\
&\leq \frac{q_{H_2}}{2^{l_1}} + \frac{1}{q}
\end{aligned}
$$

Let **DecErr** be an event that $Valid \mid \neg AskH_3$ happens during the entire simulation. Then, since q_D decryption oracle queries are made, we have

$$Pr[DecErr] \leq \frac{q_D q_{H_2}}{2^{l_1}} + \frac{q_D}{q}.$$

Now we define an event **E** to be $(AskH_3^* \vee (AskH_2^* \mid \neg AskH_3^*) \vee DecErr) \mid \neg Abort$ where **Abort** denotes an event that B aborts during the simulation. By definition of $\varepsilon^{'}$ and $\varepsilon^{''}$, we then have

$$\begin{aligned}(\varepsilon - \varepsilon^{'}) &\leq \mid Pr[\beta^{'} - \beta] - \frac{1}{2} \\ &\leq \frac{1}{2} Pr[E] \\ &\leq \frac{1}{2Pr[\neg Abort]}(Pr[AskH_3^*] + Pr[AskH_2^* \mid \neg AskH_3^*] + Pr[DecErr])\end{aligned}$$

First, notice that the probability that B does not abort during the simulation is given by $\delta^{q_{prv}(1-\delta)}$ which is maximized at $\delta = 1 - \frac{1}{q_{prv}+1}$.Hence we have $Pr[\neg Abort] \leq \frac{1}{e(q_{prv}+1)}$ where e denotes the base of the natural logarithm.

Since $Pr[AskH_2^* \mid \neg AskH_3^*] \leq \frac{q_{H_2}}{2^{l_1}}$ and $Pr[DecErr] \leq \frac{q_D q_{H_2}}{2^{l_1}} + \frac{q_D}{q}$, we obtain

$$Pr[AskH_3^*] \geq \frac{2(\varepsilon - \varepsilon^{''})}{e(q_{prv}+1)} - \frac{q_{H_2}}{2^{l_1}} - \frac{q_D q_{H_2}}{2^{l_1}} - \frac{q_D}{q}$$

Meanwhile, if $AskH_3^*$ happens then B will be able to solve the CDH problem by picking $(\frac{k_i}{g^{as^*} g^{az^*}})^{\frac{1}{c^*}}$ from the set S defined in the simulation.Consequently, we obtain

$$\varepsilon^{'} \geq \frac{1}{q_{H_3}}(\frac{2(\varepsilon - \varepsilon^{''})}{e(q_{prv}+1)} - \frac{q_{H_2}}{2^{l_1}} - \frac{q_D q_{H_2}}{2^{l_1}} - \frac{q_D}{q}),$$

The running time of the CDH attacker B is

$$t^{'} \geq t + t^{''} + 2(q_{par} + 2q_{pub} + q_{prv})t_{ex} + 10 q_D q_{H_2} q_{H_3} t_{ex} + 3t_{ex}$$

where t_{ex} denotes the time for computing exponentiation in Z_{p^*}.

[**Remark**]Our security proof follows the Baek et al's security proof for their CLPKE scheme without pairing in [6], but interestingly, their security proof was not correct in the Simulation on challenge phase.When the Type I attacker $A_I$possibly replaces the public key $(\omega^*, u^*) = (g^{s*}, g^{z*})$ associated with the target identity ID^* with its own $(\omega, u) = (g^s, g^z)$. When $s \neq s^*$, the CDH attack algorithm B simulating the environment of A_I may not know s and hence has no way to find the Diffie-Hellman key at the end of the simulation and hence fails to solve the CDH problem. but our security proof can avoid this case, because our scheme is a self-generated certificate public key encryption, the replacing public key attack can not happen unless the schnnor signature scheme is not secure.

Theorem 2. *The SGC-PKE scheme is* $(t, q_{H_1}, q_{H_2}, q_{H_3}, q_{par}, q_{pub}, q_{prv}, q_D, \epsilon)$ *- IND-CCA secure against the Type II attacker* A_{II} *in the random oracle assuming the CDH problem is* $(t^{'}, \epsilon^{'})$*-intractable and the Schnorr's signature scheme is* $(t^{''}, \epsilon^{''})$ *secure against the adaptively chosen message attack in the random oracle model , where* $\epsilon^{'} > \frac{1}{q_{H_2}}(\frac{2(\epsilon-\epsilon^{''})}{e(q_{prv}+1)} - \frac{q_{H_2}}{2^{l_1}} - \frac{q_D q_{H_2}}{2^{l_1}} - \frac{q_D}{q})$ *and* $t^{'} > t + t^{''} + 2(2q_{pub} + q_{prv})t_{ex} + 10q_D q_{H_2} q_{H_3} t_{ex} + 3t_{ex}$ *where* t_{ex} *denotes the time for computing exponentiation in* Z_{p^*}*.*

Theorem 3. *The SGC-PKE scheme proposed in this paper is secure against DOD adversary, assuming that the Schnorr's signature scheme is secure against the adaptively chosen message attack in the random oracle model.*

The proof of the above two theorems is similar as [1]. Due to lack of space, we omit it.

7 Concluding Remarks

In this paper, we show that Lai and Kou's SGC-PKE scheme cannot resist man-in-the-middle attack. First, we propose two security models–Type A model and Type B model–for CL-PKE we point out that the security model in their paper does not fit for their scheme.(We thoroughly revisit certificateless public cryptography definition and security model in appendix,which maybe be interesting for cryptographic community). Second, we give a man-in-the-middle attack to their scheme in the Type B model and a rescue SGC-PKE scheme by giving a little change to the original scheme. We further point out the reason for successfully attacking is binding the user's secret key with the multiply of partial public key from KGC and user's self-generated public key instead of binding with partial public key from KGC and user's self-generated public key independently. At last, Based on Baek's security proof for their CLPKE scheme without pairing in [6],we prove our new scheme's security in the random oracle model.

References

1. J.Lai and W.Kou. Self-Generated-Certificate Public Key Encryption Without Pairing. In *Public Key Cryptography (PKC'07),* LNCS 4450, pages 476–489. Springer–Verlag, 2007.
2. J. K. Liu and M. H. Au. Self-Generated-Certificate Public Key Cryptosystem. Cryptology ePrint Archive, Report 2006/194, 2006.
3. J. K. Liu and M. H. Au.Self-Generated-Certificate Public Key Cryptography and Certificateless Signature/Encryption Scheme in the standard Model. In *AisaCCS 2007,* , pages 273–283, 2007.
4. S. S. Al-Riyami and K. Paterson. Certificateless public key cryptography. In *Advances in Cryptology, Proc. ASIACRYPT 2003,* LNCS 2894, pages 452–473. Springer–Verlag, 2003.
5. S. S. Al-Riyami and K. Paterson. Certificateless public key cryptography. Cryptology ePrint Archive, Report 2003/126, 2003.

6. J. Baek, R. Safavi-Naini, and W. Susilo. Certificateless public key encryption without pairing. In *ISC 2005,* LNCS 3650, pages. 134–148. Springer–Verlag, 2005.
7. B. Libert and J. Quisquater. On constructing certificateless cryptosystems from identity based encryption. In *Public Key Cryptography (PKC'06),* LNCS 3958, pages 474–490. Springer–Verlag, 2006.
8. D. H. Yum and P. J. Lee. Generic construction of certificateless encryption. In *ICCSA'04,* LNCS 3040, pages. 802–811. Springer–Verlag, 2004.
9. Y. Shi and J. Li. Provable efficient certificateless public key encryption. Cryptology ePrint Archive, Report 2005/287, 2005.
10. Z. Cheng and R. Comley. Efficient certificateless public key encryption. Cryptology ePrint Archive, Report 2005/012, 2005.
11. M. Girault. Self-certified public keys. In *Advances in Cryptology, Proc. EUROCRYPT 1991,* LNCS 547, pages 490–497. Springer–Verlag, 1992.
12. A. Shamir. Identity-based Cryptosystems and Signature Schemes. In *Advances in Cryptology, Proc. CRYPTO 1984,* LNCS 196, pages 47–53. Springer–Verlag, 1984.
13. C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology, Vol. 4, No. 3*, pages. 161–174, 1991.
14. Au,M.H.,Chen,J.,Liu,J.K.,Mu,Y.,Wong,D.S.,Yang,G. Malicious KGC attack in certificateless cryptography. In *Proc.ACM Symposium on Information, Computer and Communications Security, CCS'07* ACM Press,New-York,2007.
15. Dent,A.W. A survey of certificateless encryption schemes and security models.Cryptology ePrint Archive, Report 2006/211, 2006.
16. Dent,A.W., Libert,B., Paterson,K.G. Certificateless encryption schemes strongly secure in the standard model. Cryptology ePrint Archive, Report 2007/121, 2007.
17. Galindo,D., Morillo,P., Rafols,C. Breaking Yum and Lee generic constructions of certificateless and certificate-based encryption schemes. In *EuroPKI2006,* LNCS 4043, pages 81–91. Springer–Verlag, 2006.
18. Hu,B.C.,Wong,D.S.,Zhang,Z.,Deng,X. Key replacement attack against a generic construction of certificateless signature.In *ACISP2006* LNCS 4058, pages 235–246. Springer–Verlag, 2006.
19. Huang,Q.,Wong,D.S.Genericc ertificateless encryption in the standard model. In *IWSEC2007,* LNCS 4752, pages 278–291. Springer–Verlag, 2007.
20. Huang,X., Susilo,W., Mu,Y., Zhang,F. On the security of certificateless signature schemes from Asiacrypt2003. In *CANS'05,* LNCS 3810, pages 130–145. Springer–Verlag,2005.
21. Zhang,Z.,Wong,D.S.,Xu,J.,Feng,D.,Zhang,F. Certificateless public-key signature: Security model and efficient construction. In *ACNS'06,* LNCS 3989, pages 293–308. Springer–Verlag,2006.

A Certificateless Public Cryptography KeyGeneration Algorithm and Security Model Revisited

In the current literature [1-10, 14-21], LK's CL-PKE is different from the other's CL-PKE by LK's $PartialKeyExtract$ algorithm including user's public key as input while other's not. Also, LK's $PartialKeyExtract$ algorithm must run after $UserKeyGeneration$ algorithm while other's $PartialKeyExtract$ algorithm can run independently with $UserKeyGeneration$ algorithm. We thoroughly revisiting the current literature in CL-PKC (We suggest Dent's paper [15] as a good reference for current CL research) and we give a new framework for CL-PKC's KeyGeneration Algorithm and reconsider its security model.

A.1 KeyGeneration Algorithm

Syntax: {.} means the variant is a choice, not a necessary input.Variant with $*$ means this variant is a malicious variant controlled by the adversary.

1.KGC and USER thoroughly independently (User's setup and KGC's setup independently)

-*Setup*: takes as input a security parameter 1^k and returns the master private key msk and the master public key mpk.This algorithm is run by a KGC to initially setup a certicateless system.

$$(mpk, msk) = Setup(1^k)$$

-*PartialprivatekeyGen*: takes as input the master public key mpk,the master private key msk, and an identifier $ID \in \{0,1\}^*$.It outputs a partial private key d_{ID}.This algorithm is run by a KGC once for each user, and the corresponding partial private key is distributed to that user in a suitably secure manner.

$$d_{ID} = Extract(ID, mpk, msk)$$

-*UserkeyGen*: takes as input a security parameter 1^k and generates user's public key upk and private key usk.

$$(upk, usk) = UserkeyGen(1^k, \{ID\})$$

-*Setprivatekey*: takes as input the master public key mpk, an entity's partial private key d_{ID} an entity's public value upk and an entity's secret value usk .It outputs the full private key sk for that user. This algorithm is run once by the user.

$$sk = Setprivatekey(mpk, d_{ID}, upk, usk)$$

-*SetPublickey*: given the master public key mpk, a user's public key upk and a user's secret value sk, this algorithm outputs a public key $pk_{ID} \in PK$ for that user. This algorithm is run once by the user and the resulting public key is widely and freely distributed.

$$pk_{ID} = Setpublickey(mpk, upk, sk)$$

[**Remark**]: this is the algorithm considered in a few existing literature papers, including HU and HUANG's works[18,19]. Most of these work are constructed just in generic, no efficient construction is proposed.

2. KGC's setup influence User's setup:

2.1 KGC's partialprivatekey and user's localkey generate independently

-*Setup*: Takes as input a security parameter $1k$ and returns the master private key msk and the master public key mpk.This algorithm is run by a KGC to initially setup a certicateless system.

$$(mpk^*, msk) = Setup(1^k)$$

-*PartialprivatekeyGen*: takes as input the master public key mpk^*,the master private key msk, and an identifier ID .It out puts a partial private key d_{ID}.This algorithm is run by a KGC once for each user, and the corresponding partial private key is distributed to that user in a suitably secure manner.

$$d_{ID} = Extract(ID, mpk^*, msk)$$

-*UserkeyGen*: given the master public key mpk^* and an entity's identity ID as input, and outputs user's public key upk and private key usk. This algorithm is run once by the user.

$$(upk, usk) = UserkeyGen(mpk^*, ID)$$

-*Setprivatekey*: takes as input the master public key mpk^*,an entity's partial private key d_{ID} an entity's public value upk and an entity's secret value usk .It outputs the full private key sk for that user. This algorithm is run once by the user. equation* sk=Setprivatekey (mpk,$\mathrm{d}_{ID}, \{upk\}, usk$)
-*SetPublickey*: given the master public key mpk^*, a user's public key upk and a user's secret value sk, this algorithm outputs a public key $pk_{ID} \in PK$ for that user. This algorithm is run once by the user and the resulting public key is widely and freely distributed.

$$pk_{ID} = Setpublickey(mpk^*, \{upk\}, sk)$$

[**Remark**]: this is the algorithm considered in most existing literature papers, including AP and most other works [4,5]. UserkeyGen and PartialkeyGen can work independently. But it can reach only Trust Level 2.

2.2 KGC's partialprivatekey generation influenced by User's localkey generation

-*Setup*: Takes as input a security parameter 1^k and returns the master private key msk and the master public key mpk^*.This algorithm is run by a KGC to initially setup a certicateless system. equation* $(\mathrm{mpk}^*, msk) = Setup(1^k)$
-*UserkeyGen*: given the master public key mpk^* and an entity's identity ID as input, and outputs user's public key upk and private key usk. This algorithm is run once by the user.

$$(upk, usk) = UserkeyGen(mpk^*, ID)$$

-*PartialprivatekeyGen*: takes as input the master public key mpk^*,the master private key msk, a user's public key upk and an identifier ID .It out puts a partial private key d_{ID}.This algorithm is run by a KGC once for each user, and the corresponding partial private key is distributed to that user in a suitably secure manner.

$$d_{ID} = Extract(ID, mpk^*, msk, upk)$$

-*Setprivatekey*: takes as input the master public key mpk^*, an entity's partial private key d_{ID}, an entity's public value upk and an entity's secret value usk .It

outputs the full private key sk for that user. This algorithm is run once by the user.

$$sk = Setprivatekey(mpk^*, d_{ID}, \{upk\}, usk)$$

-*SetPublickey*: given the master public key mpk, a user's public key upk and a user's secret value sk, this algorithm outputs a public key for that user. This algorithm is run once by the user and the resulting public key is widely and freely distributed.

$$pk_{ID} = Setpublickey(mpk^*, \{upk\}, sk)$$

[**Remark**]: this is the algorithm considered in LK's paper[1]and our paper. PartialkeyGen must work before UserkeyGen. But it can reach Trust Level 3.

2.3 User's localkey generation influenced by KGC's partialprivatekey generation

-*Setup*: Takes as input a security parameter 1^k and returns the master private key msk and the master public key mpk^*.This algorithm is run by a KGC to initially setup a certicateless system.

$$(mpk^*, msk) = Setup(1^k)$$

-*PartialprivatekeyGen*: takes as input the master public key mpk^*,the master private key msk, and an identifier ID .It outputs a partial private key d_{ID}.This algorithm is run by a KGC once for each user, and the corresponding partial private key is distributed to that user in a suitably secure manner.

$$d_{ID} = Extract(ID, mpk^*, msk)$$

-*UserkeyGen*: given the master public key mpk^* and an entity's identity ID as input, and outputs user's public key upk and private key usk. This algorithm is run once by the user.

$$(upk, usk) = UserkeyGen(mpk^*, ID, d_{ID})$$

-*Setprivatekey*: takes as input the master public key mpk^*, an entity's partial private key d_{ID} an entity's public value upk and an entity's secret value usk .It outputs the full private key sk for that user. This algorithm is run once by the user.

$$sk = Setprivatekey(mpk^*, d_{ID}, \{upk\}, usk)$$

-*SetPublickey*: given the master public key mpk^*, a user's public key upk and a user's secret value sk, this algorithm outputs a public key pk_{ID} for that user. This algorithm is run once by the user and the resulting public key is widely and freely distributed.

$$pk_{ID} = Setpublickey(mpk^*, \{upk\}, sk)$$

[**Remark**]: this is the algorithm not considered in existing literature papers, but we cannot ignore this case.

3. User's setup influence KGC's setup

[**Remark**]: this is the algorithm not considered in existing literature papers, but we cannot ignore this case.

B Security Model

B.1 Adversary Types

There are at lest five kinds of adversaries we must consider:
-Type 1 adversary: Original Type 1 adversary can replace the target user's public key by other user's public key or own user's public key. But the adversary denied accessing to the KGC's master key.
-Type 2 adversary: Original Type 2 adversary can know the KGC's master key but cannot replace the target user's public key before challenge phase.
-Type 3 adversary: Malicious but passive KGC can plant trapdoor in the public parameters
-Type 4 adversary*: Malicious User can plant trapdoor in the upk which can happen in the standard model
-Type 5 adversary*: Man-in-the-middle adversary can be pretend as KGC or User.

The adverarys with symbol * means these adversaries have not been considered in the litreture until now.

B.2 Concurrent Environment

Furthermore, we must consider multi-public-key one ID model, this model maybe be different from one-public-key one ID model.We must also consider UC security for CL cryptography, this maybe gives important impact on CL cryptography for it is adapted in practical application.