Privacy-Preserving DRM for Cloud Computing

Ronald Petrlík
Department of Computer Science
University of Paderborn
33098 Paderborn, Germany
ronald.petrlic@upb.de

Christoph Sorge
Department of Computer Science
University of Paderborn
33098 Paderborn, Germany
christoph.sorge@upb.de

Abstract—We come up with a digital rights management (DRM) concept for cloud computing and show how license management for software within the cloud can be achieved in a privacy-friendly manner. In our scenario, users who buy software from software providers stay anonymous. At the same time, our approach guarantees that software licenses are bound to users and their validity is checked before execution. We employ a software re-encryption scheme so that computing centers which execute users’ software are not able to build user profiles—not even under pseudonym—of their users. We combine secret sharing and homomorphic encryption. We make sure that malicious users are unable to relay software to others. DRM constitutes an incentive for software providers to take part in a future cloud computing scenario. We make this scenario more attractive for users by preserving their privacy.

I. INTRODUCTION

Digital rights management (DRM) is about controlling the usage of digital media, supporting license models that allow media providers the charging of content usage. Cloud computing denotes the on-demand computing power and software provision by (large) computing centers.

We come up with a privacy-preserving DRM scheme for cloud computing. Our main contributions in this paper are:

- We are the first to propose a DRM scheme for the cloud.
- The software license is checked each time the software is executed without allowing any party to build user profiles.
- We employ a secret sharing scheme based on homomorphic encryption and further combine it with a re-encryption scheme to achieve privacy protection.

In (future) cloud computing, we can identify software providers that provide software in return for money and computing centers that execute the users’ software in return for money. This scenario is different from traditional cloud computing where providers typically provide software and its execution—which does not fit the needs of users asking for an execution of software not provided by the providers. The separation between the provision of software and its execution facilitates the market entrance of software providers and computing centers as they can focus on their core business. We assume that in such a future cloud computing scenario, there need to be some dedicated parties—we call them service providers—that act as agents for the users to facilitate the use of cloud computing. Those service providers handle the users’ payments for software bought from software providers and software executions carried out at computing centers. Moreover, the service providers handle the storage of the users’ software within the cloud. They are also responsible for checking the licenses before allowing the software execution.

The remainder of this paper is structured as follows. Related work is covered in Sect. II. In Sect. III we come up with the requirements for a cloud computing DRM system. We present the architecture and the corresponding protocol of our proposed concept in Sect. IV. We discuss the concept in Sect. V before we conclude and give an outlook in Sect. VI.

II. RELATED WORK

PerLMAN et al. [1] present a privacy-preserving DRM concept that allows users buying content without revealing which content it is about. Their first scheme is based on anonymous cash. The distributor blindly signs cash that is bought by users. Users spend the cash to buy content from the same distributor. Their second scheme is based on blind decryption. Users ask the distributor to decrypt an encrypted decryption key to decrypt the content. The decryption key is encrypted with the distributor’s public key, so only the distributor is able to decrypt the key when the user pays for the content. To prevent the distributor from learning which content the user will decrypt with the key—as each content is encrypted with a different key—the user blinds the encrypted key during the request. The authors conclude that the second scheme is more efficient than the one based on anonymous cash. A shortcoming of their concept is that it does not allow for different license models, e.g. models that limit the number of executions of the content are not supported, as the license cannot be checked each time the content is executed.

Conrado et al. [2] present a privacy-preserving DRM concept that allows users to anonymously buy content from a provider and that prevents the tracking of users while the content is accessed. Users’ identities are disassociated from their content. They introduce temporary pseudonyms which are managed by users’ smart cards. Their main assumptions are that users contact the provider by means of anonymous channels, that the smart cards do not reveal any personally identifiable information (PII), and that content can only be accessed by compliant devices—devices that behave according to the DRM rules. The provider may be able to build user profiles by learning the association between the smart
card’s public key and the content ID during the buying phase. Compliant devices under an attacker’s control may be able to identify a user and learn which content is accessed. Tracking is possible for a limited number of transactions as the user's pseudonyms change periodically. The authors also present a solution to support countable rights licenses, i.e. where the content can be accessed only a limited number of times. For that purpose, they introduce an additional domain manager device which creates personalized user rights.

In contrast to the approach by PERLMAN ET AL. [1] our solution allows for a license checking before every software execution and thus enabling different license models, which is of importance in (future) cloud computing. Users do not need to possess any trusted devices, as needed in the approach by CONRADO ET AL. [2]. Our solution protects users from profile building — in contrast to the solution by CONRADO ET AL.

III. REQUIREMENTS

A. Economics

Software providers typically distribute software over the Internet today. To make users pay for the software use, providers sell the licenses needed to run the software. In cloud computing, users ask for software renting models rather than buying software. To support this flexibility, our first requirement is as follows:

- **Req. I: Different license models have to be supported.**

As cloud computing abstracts from computing centers, users must not be limited in terms of software execution:

- **Req. II: The software license must not bind the software execution to a certain computing center.**

The only exception is that a user is not allowed to execute the software on its own platform. This would allow for a bypassing of the DRM scheme in place. Thus, only certified computing centers are supported.

B. Security

Users must not be able to relay software to others:

- **Req. III: The software must be protected so that it cannot be relayed by a user.**

Another requirement is that no data must leak to unauthorized parties and actions must be assignable to users:

- **Req. IV: Data confidentiality against unauthorized parties and non-repudiation of users’ actions must be achieved.**

C. Privacy

Cloud computing is problematical as personal data are stored and processed at any computing center in the cloud1. In most cases, it is not clear in which countries the computing centers are, which privacy regulations apply, etc. The problems mostly have not been solved yet. We are addressing requirements concerning privacy protection within the cloud for our scenario of software buying and execution.

Privacy protection is the protection of personally identifiable information (PII), i.e. “information which can be used to distinguish or trace an individual’s identity” [3]. We do not only address PII but we include the protection of personal data as well. Personal data are “information relating to an identified or identifiable natural person” [4]. There may exist certain information that do not (directly) allow for an identification of individuals but that relate to natural persons and thus have to be handled with care. In our scenario the PII are identities that reveal who buys and executes software. Personal information are information like which software is bought and how often it is executed. To prevent the revealing of PII we have to look for anonymity and to prevent the revealing of personal data we have to look for prevention of profile building.

1) **Anonymity:** Anonymity means that there is no way to be able to relate individual information to an identified or identifiable natural person. In our scenario, neither the software provider that sells software nor the computing center that executes software must be able to find out who buys and executes software. The users should also stay anonymous towards the service provider that handles the user interactions with the software provider and the computing center:

- **Req. V: The user should stay anonymous towards service provider, software provider, and computing center.**

2) **Profile building prevention:** Profile building means that a party is able to relate actions to certain users of the system. There is a special form of profile building called profile building under pseudonym. This form of profile building does not relate actions to certain users but relates them to certain pseudonyms. A pseudonym is an identifier other than the real name which is relatable to a certain person if the mapping is known. Even though an attacker does not know the mapping and thus, is not able to directly map the pseudonym to a certain person, having a profile under pseudonym may suffice to differentiate between users and indirectly identify a certain person again. Such profiles under pseudonym may be built if a user appears towards a party under the same pseudonym more than once. Note that preventing profile building under pseudonym implies preventing profile building (under real identity) and thus, we cover the stronger requirement of profile building under pseudonym in this work. For our scenario, preventing profile building means that no party is able to see who buys which software and how often it is being executed:

- **Req. VI: Profile building (under pseudonym) must not be possible for any party.**

In practical DRM schemes, where licenses are checked online by the content provider each time the content is executed, such profile building (under pseudonym) is possible — this surveillance of information about intellectual consumption threatens the rights of personal integrity. [5] This should highlight the importance of Req. VI in our scenario.

IV. CONCEPT

As we have already pointed out in Sect. I, we assume that in a future cloud computing scenario there exists a number

---

1Note that this applies for companies as well — however, in this case we rather talk about business secrets than about privacy.
of service providers that act as proxies between users and software providers and between users and computing centers. Users who want to buy software from a software provider and execute it at a computing center register with any service provider2. The service provider manages its users’ registration data, anonymous payments, bought software, etc., and checks whether the license is valid before allowing the software execution. Software providers rely on the service providers to perform the checking — service level agreements are in place.

A. Assumptions

1) A public key infrastructure (PKI) is available: The PKI is needed to issue certificates to the involved parties. We denote the user’s public RSA key by \((e_U, N_U)\) and the corresponding private key by \((d_U, N_U)\). The service provider possesses the key pair \((e_{SP}, N_{SP}), (d_{SP}, N_{SP})\), the software provider the pair \((e_{SWP}, N_{SWP}), (d_{SWP}, N_{SWP})\), and the computing center the pair \((e_{CC}, N_{CC}), (d_{CC}, N_{CC})\). As the user needs to be authenticated by the service provider to initiate the buying, payment, and execution of software, the service provider needs to know the (registered) users’ public keys. The service provider must not be able to trace back and identify users based on their public keys — only in case of fraud the relation between public key and user identity is disclosed by the PKI. The user needs to know the public keys of the software providers he buys software from and those of the computing centers that should execute the software.

2) The software provider offers a number of products: If a software provider offered only a single product, it would be easy for the service provider to learn which product is bought by its users from that software provider.

3) The parties do not cooperate to break user privacy: The service provider must not share its knowledge about users with the software provider or the computing center. Otherwise they would be able to build user profiles under pseudonym.

4) DRM is not applied for the computing center: We do not cover mechanisms that restrict the computing center.

B. Preliminaries

We employ an additively homomorphic encryption scheme as presented by Castelluccia et al. [6] for the software encryption and secret sharing. The scheme, shown in Tab. I, is based on modular addition (+). It specifies encryption, decryption, and computation performed on cipher texts (aggregation). The authors suggest to generate the keystream \(k\) by using a stream cipher such as RC4. They prove that their scheme is provably secure if the keystream is not reused.

C. Software buying

Fig. 1 shows the software buying protocol. The user constructs message \(MSG_1 = (\text{buy } SW_x, \text{payment token})\) that is sent to the software provider to buy software \(x\). We do not cover the details of the payment token.3 The user randomly chooses a symmetric key \(K_{USWP}\) later used between user and software provider. The message and the key are encrypted with the software provider’s public key \(e_{SWP}\). The result \(MSG_1\) and the signature are sent towards the service provider in step (1). In step (2) the service provider verifies the signature of the encrypted message \(MSG_1\). The encrypted message — without the signature — is forwarded towards the software provider via a secure channel5 in step (3). In step (4) the software provider extracts the message \(MSG_1\) and the symmetric key \(K_{USWP}\).

D. Software and license retrieval

Fig. 2 shows the software and license retrieval protocol.

1) Software encryption: To prevent the encrypted software from being related to a specific software by the service provider by analyzing its length, padding is employed to the software before encryption. In step (5) the software provider provides the service provider with the encrypted software \(SW_x\) — denoted by \(Enc_{K_x}(SW_x)\). The employed encryption scheme, based on the scheme in Tab. I, is as follows:

\[
Enc_{K_x}(SW_x) = SW_x + K_x \mod M,
\]

where \(M\) is a large integer, \(SW_x \in [0, M - 1]\) and \(K_x \in [0, M - 1]\) has to apply. The key \(K_x\) is unique, i.e. whenever a software is bought the software provider generates a new key with the RC4 stream cipher. We employ the scheme for the software encryption without making use of the homomorphic property — the reason why we still employ this scheme will become clear when we cover the software re-encryption in Sect. IV-E2.

2) Secret sharing: The idea behind the encryption of the software is that neither the user nor the service provider get the (symmetric) decryption key \(K_x\) needed to decrypt \(SW_x\). This is because the user is not trusted by the software provider. If the user had the key, he would be able to decrypt the software and share it with others. The service provider does not get the key because otherwise it would know which software has been bought by the user and how often it is executed. We propose a scheme that is based on secret sharing — the decryption key is subdivided into two parts, one for the user and one for the service provider, \(ShareU_{K_x}\) and \(ShareSP_{K_x}\). We employ an additive secret sharing scheme, i.e. \(K_x = ShareU_{K_x} + ShareSP_{K_x}\), where \(ShareU_{K_x} \in [0, M - 1]\) is chosen randomly and \(ShareSP_{K_x}\) is chosen accordingly.

5Secure channel means that the communication is encrypted, authenticated, and integrity-protected. TLS may be used to achieve those protection goals.

---

**TABLE I**

| Encryption: | - Message \(m \in [0, M - 1]\), where \(M\) is a large integer |
| - randomly generated keystream \(k \in [0, M - 1]\) |
| - \(c = (m + k) \mod M\) |

**Decryption: | Dec(c, k, M) = c - k (mod M)**

**Aggregation: |**
| - Let \(c_1 = Enc(m_1, k_1, M)\) and \(c_2 = Enc(m_2, k_2, M)\) |
| - For \(k = k_1 + k_2\), \(Dec(c_1 + c_2, k, M) = m_1 + m_2\) |
3) Software license: The encrypted software comes with the corresponding license \(license_x\) in step (5). The \(license_x\) is a data structure that contains the following fields: encrypted software identification, validity period, pricing model, digital signature, and \(ShareSP_{K_x}\). The encrypted software identification does not contain information about the software but rather helps the service provider identifying the user’s encrypted software. The validity period and the pricing model, e.g. number of allowed software executions or flat rate, are checked by the service provider each time the software is intended to be executed. The license is digitally signed by the software provider. The \(ShareSP_{K_x}\) is the service provider’s share for the software decryption key \(K_x\). The relation between user and license is managed by the service provider — the software provider is not aware of the relation between user and license and thus, it is not stored within the license.

4) Retrieval: The encrypted software and the license are not forwarded to the user by the service provider but stored for later usage. A secure channel between the software provider and the user, whose communication is relayed via the service provider, is needed so that the software provider can securely communicate the user’s share \(ShareU_{K_x}\) of the decryption key \(K_x\) towards the user in step (6)— without allowing the service provider to get the share and thus being able to decrypt the software. The software provider employs a symmetric encryption scheme, e.g. AES [7], and uses the symmetric key \(K_{USWP}\) that it received by the user in protocol step (1).

E. Software execution

Fig. 3 shows the software execution protocol. The user encrypts \(Msg_2 = \text{(command, } mv, K_{HU}, \text{ payment token)}\) in step (7) and sends it to the service provider. The command tells the service provider what to do, e.g. execute the software at a certain computing center. The \textit{modification vector} \(mv\) tells the service provider how to re-encrypt the encrypted software, as covered in Sect. IV-E2. The \textit{homomorphic user key} \(K_{HU}\), randomly generated by the user and never reused, is used for the reassembling of the software decryption key, i.e. the \textit{secret combination}, as covered next. The \textit{payment token} denotes the payment for the software execution.\(^6\)

1) Secret combination: The user and the service provider send their share values towards the computing center for the computing center to be able to reconstruct the software decryption key. If the user executes a software more than once at a certain computing center, the computing center would be able to build a profile (under pseudonym) if it received the user’s or service provider’s share directly. We employ a secret sharing scheme that is based on the homomorphic encryption scheme covered in Sect. IV-B. The computing center will only get the \textit{encrypted} share values of the user and the service provider and will be provided with a key that allows the decryption of the software decryption key \(K_x\) only — but not the individual encrypted share values.

In step (8) the user sends the computing center its \textit{modified} share value \(msg_u\) — encrypted with the user’s homomorphic key \(K_{HU}\). The \(msg_u\) is computed as: \(msg_u = ShareU_{K_x} + mv\), where \(msg_u \in [0, M - 1]\) has to apply. The whole message, incorporating a common key \(K_{UCC}\) between the user and the computing center — chosen randomly by the user—as well, is encrypted with the computing center’s public key so that the service provider, acting as relaying node again, does not get into possession of the encrypted \(msg_u\) and the common key.

The service provider generates a homomorphic key \(K_{HSP}\) and computes the aggregated homomorphic key in step (11): \(K_{aggrh} = K_{HU} + K_{HSP}\), where \(K_{aggrh} \in [0, M - 1]\). \(K_{aggrh}\) is unique for each transaction as both \(K_{HU}\) and \(K_{HSP}\) are

\(^6\)As already pointed out in the software buying phase in Sect. IV-C, the payment token might have been issued by the service provider or a third party in advance — after the payment of a certain amount of money.
randomly chosen each time. $K_{aggrh}$ allows the computing center to decrypt the (new) software decryption key $K_y$. In step (12) the service provider sends the computing center $K_{aggrh}$ along with the re-encrypted service, the service provider’s $ShareSP_{K_s}$ — encrypted with $K_{HSP}$ —, as well as the payment token to pay for the software execution. The message is encrypted with the computing center’s public key.

The computing center retrieves the software key $K_y$ by decrypting the aggregation of the user’s and the service provider’s encrypted shares with $K_{aggrh}$ in step (13). According to the specification presented in Tab. I, this computation is performed as: $K_y = (Enc_{K_{HSP}}(msu) + Enc_{K_{aggrh}}(ShareSP_{K_s})) \mod M$. In step (14) the computing center decrypts the software as: $SW_x = (Enc_{K_y}(SW_x) - K_y) \mod M$. The computing center executes the software, receives input data from the user (step (15)), and returns the computation result to the user in step (16). The communication between the user and the computing center is encrypted with the common key $K_{UCC}$ between both parties.

2) Software re-encryption: The re-encryption is performed by the service provider after it has checked the corresponding license (step (9)). As explained above, the user’s $ShareU_{K_s}$ is modified by the user, resulting in the modified share value $msu$, before it is transmitted to the computing center. This modification is needed as we propose that the service provider modifies the encrypted software each time it is sent to the computing center. Remember that the software encryption key $K_x$ is changed for every software purchase by the software provider. If the service provider did not re-encrypt the software, the key would stay static as well — allowing the computing center to build a user profile under pseudonym.

To allow for such a software re-encryption, the user’s share value has to be modified each time the software is executed. This is why we introduced the modification vector $mv$ in step (7). The $mv$ is randomly chosen by the user for each execution. The user’s modified share value $msu$ is computed by adding the modification vector to the share value. Accordingly, the software is modified, i.e. re-encrypted, by adding the same modification vector to the software: $Enc_{K_y}(SW_x) = Enc_{K_y}(SW_x) + mv$ (step (10)). For the decryption to succeed, $(SW_x + mv) \in [0, M - 1]$ must hold.

F. Improvement of the scheme

Our proposed scheme can be improved in terms of communication overhead. Instead of transmitting $ShareU_{K_s}$ from the software provider to the user, we may transmit a shorter seed instead. The seed is used as input to a pseudorandom number generator like RC4 by the user in order to get $ShareU_{K_s}$. The same approach can be taken for the transmission of the modification vector $mv$ and the homomorphic user key $K_{HU}$ from the user to the service provider.

V. DISCUSSION

A. Economics

The service provider checks the license before allowing the software execution. Depending on the license model the service provider decides whether the permission on the part of the software provider, as stated in the license, is still given. Our concept meets Req. I asking for different license models.

The user can choose the computing center where the software is executed. We meet Req. II, the very requirement of flexibility in cloud computing, with our approach.

We denote the fact that the software is now encrypted with the — still unknown — key $K_y$ by the new index “y”.

7 The details of the software decryption key change and the software re-encryption are explained in Sect. IV-E2.

8 We denote the fact that the software is now encrypted with the — still unknown — key $K_y$ by the new index “y”.

Fig. 3. Message sequence chart showing the software execution protocol executed between user, service provider, and computing center.
B. Security

The software provider provides the service provider with the encrypted software. The user does not get in possession of the software and is not able to relay it to anyone else. The user needs to contribute its software decryption key share for each software execution. Relaying software would mean for a user to reveal its authentication credentials for the service provider as well as its share value. Summing up, we meet Req. III. Any communication between the involved parties is encrypted and particularly the user’s software buying and execution commands are digitally signed to achieve non-repudiation for those actions. Thus, we comply with Req. IV as well.

C. Privacy

1) Anonymity: The user only directly communicates with the service provider—the other parties do not get identifying information like user data, network addresses, etc. An anonymous payment scheme is used to pay for the software and its execution. We meet Req. V asking for anonymity.

2) Profile building prevention: The software provider takes part only during the software buying and the software and license retrieval. There is no way for the software provider to build a profile of its (unknown) users under pseudonym.

We prevent the computing center from profile building under pseudonym by changing the user’s software decryption key share and the associated software re-encryption each time the software is sent to the computing center. If we did not change the share, the computing center would know that the software is executed by a user that executed it before.

The modified user’s share value and the service provider’s share value are sent towards the computing center encrypted employing a homomorphic encryption scheme. The computing center must not learn any single (modified) share value. The homomorphic keys \( K_{HU} \), \( K_{HSP} \), and the aggregated key \( K_{agg} \) are also changed for every transaction. The computing center is not able to re-identify previously used keys. The key renewal guarantees that the homomorphic encryption scheme is provably secure, as proven by CASTELLUCCIA ET AL. [6].

The service provider must not learn which software is executed by a certain user to prevent profile building under pseudonym. The security of \( \text{Msg}_1 \)—including \( SW_x \)—is based on the security of the RSA algorithm [8]. \( SW_x \) is encrypted with \( K_x \) by the software provider and sent to the service provider. As the service provider only gets its own share value of the key, it is not able to reconstruct the key \( K_x \) and decrypt the software on its own. The service provider does not learn the user’s (modified) share value as it is transmitted encrypted. The secrecy of the software encryption scheme is based on the secrecy of [6]. The scheme is proven to be provably secure if the employed key is not reused. The key \( K_y \) to encrypt \( SW_x \) is unique—we employ the RC4 stream cipher to generate the keystream. The re-encryption employs a new modification vector \( mv \) each time. The service provider does not learn the result of the software execution as the computing center encrypts the message with \( K_{UCC} \) (known by user and computing center). The service provider cannot act as MITM between user and software provider during the software buying as the user obtains the software provider’s public key via an independent channel, e.g. the PKI, and encrypts the message towards the software provider with this key. This is true for the communication between the user and the computing center as well. The service provider is not able to fool the user into executing its software within its own computing center and thereby learning the software as the user independently obtains the (intended) computing center’s public key. To sum it up, we meet Req. VI asking for the prevention of profile building (under pseudonym) for any party.

D. Correctness of the protocol

The scheme by CASTELLUCCIA ET AL. [6] is additively homomorphic due to the commutative property of addition. The software provider provides the service provider with \( (SW_x + K_x) \mod M \). The service provider additionally adds \( mv \) to the encrypted software. We required \( SW_x \in [0, M−1], K_x \in [0, M−1] \) as well as \( (SW_x+mv) \in [0, M−1] \). This last requirement is necessary as the modification of the software must not lead to an overflow. \( K_y \) is computed by the computing center as: \( K_y = ((\text{Share}_{SP} K_x + K_{HSP} + \text{Share}_{U} K_x + mv + K_{HU}) \mod M \). The decryption of the software is performed as: \( SW_x = (SW_x + K_x + mv - K_y) \mod M \).

VI. CONCLUSION AND OUTLOOK

We have pointed out that a DRM system is crucial for software providers to take part in future cloud computing. One major design goal of our concept is the protection of users’ privacy. The homomorphic encryption-based secret sharing scheme, combined with the software re-encryption scheme makes sure that users stay anonymous towards software providers and computing centers and profile building is not possible—not even under pseudonym—for any party.

REFERENCES


