A Distributed Key Management Approach

Rafael Dowsley*, Matthias Gabel*, Gerald Hübsch†, Gunther Schiefer* and Antonia Schwichtenberg†

* Karlsruhe Institute of Technology
Emails: {rafael.dowsley,matthias.gabel,gunther.schiefer}@kit.edu

† CAS Software AG
Emails: {gerald.huebsch,antonia.schwichtenberg}@cas.de

***Abstract*—Cloud computing provides reliable and highly-scalable access to resources over the internet. But outsourcing sensitive data to an probably untrusted cloud provider (third party) requires cryptographic methods like encryption. This paper presents a novel approach to a distributed cloud key management scheme. In a setting with a public cloud application, data is encrypted by a separate trusted adapter before storing somewhere else. The encryption key is not persistently stored at the adapter. Several entities share parts of the key that is computed and temporarily stored at the adapter if needed. This work describes how the key management is working during bootstrapping and runtime as well as how key recovery can be performed.**

***Index Terms*—Key Management, Key Distribution, Cloud Computing**

I. Introduction

Wherever sensitive personal data is stored and processed, security requirements are high. One good example are CRM systems that hold information like addresses, dates of birth, personal preferences and relations of any kind. These data must be protected from theft and illegal processing.

The General Data Protection Regulation of the European Union [6] names a number of abstract, negative effects that a lack of appropriate technical and organisational measures to ensure data protection can have on individuals. To make them more concrete, consider the vast impact that illegal manipulation of, for example, university applications, exam registrations and exam results, will have on CVs and career paths. The additional security requirements imposed on CRM solutions for the management of students are therefore even more drastic than those on classical CRM data. Only users with the right permission should be allowed to gather and process crucial personal data on behalf of the data owners. In the example, such users are the members of the board of examiners. Only they assign marks and store them in the university's CRM.

However, the main actors involved in any cloud-based solution are administrators of the public data centre, the solution provider, and all users with accounts to access the application. Encryption as a way to protect personal data against theft and illegal manipulation is only sufficient in case all actors are completely trustworthy and the cryptographic key cannot be compromised. In case only a single persistently stored key is required to read the data, that key can be stolen in a single attack against the data center, together with the encrypted data, without any possibility of intervention after the theft is recognized. As one possibility to increase the level of protection in such scenarios, we propose a shared key ownership model, in which the single key is split into different parts that are distributed to all involved actors and combined in volatile memory only when and as long as it is necessary.

Doing so has multiple advantages. No party is able to process encrypted data without the others' participation. This includes the possibility to enforce data ownership through the ownership of key parts. In an attack to compromise the persistently stored key, all instead of one actor would need to be attacked, increasing the chance of detection and interim intervention. An attack to obtain the combined key is aggravated by a limited time window that is difficult to predict for the attacker, again increasing the chance of detecting the attack before it is successful.

The remainder of this paper is structured as follows. Relevant related work in key management for cloud scenarios is presented in section II. The novel distributed key management approach is described in detail in section III. Section IV concludes this paper by discussing the applicability of the distributed key management for Software-as-a-Service and a brief outlook to future work.

II. Related Work

To process data, today, it is still necessary to have it unencrypted. But for security reasons it should be stored encrypted to prevent unauthorized access (all data connections must be encrypted as well). In addition, authorized access should only be possible on behalf of the data owner, the tenant. To share this data between the employees of this tenant, it is necessary to encrypt the database with one tenant key that should be only under the control of the tenant.

The most common place to store this tenant key is in a cloud system itself (e.g. a cloud provider) where the database is accessed [2]. This gives the operator of this system full data access at all times without control by the tenant. To avoid this, the sealed cloud approach [8] isolates the whole system logically and physically from every administrative access during runtime. Before administrative access is granted, all data is stored encrypted with a key which is only known to the tenant and deleted from memory. To restart the sealed

cloud, the tenant has to provide the key for data access again. Another way to give the tenant more control of data access is to have a specific key server operated by the tenant itself [10], [7]. Every access to the data needs to fetch the key from the tenant's key server, which requires an authorization before revealing the key. To avoid the necessity of operating a key server, the MimoSecco project [9] secured the key by storing it inside a secure hardware, which revealed the key only upon tenant authorization (more precisely: by an authenticated user which is authorized by the tenant to access the data).

Some cloud providers offer full data encryption for their storage back-ends. Amazon S3 [1], [3] for instance provides transparent data encryption. For encryption and decryption a symmetric algorithm (currently AES-256) is used. This key, however, is either managed and persistently stored by Amazon itself that is then able to decrypt all data. Or – in case the user manages the key and only uploads encrypted data – query processing in the cloud is not possible. The only application for this set-up is simple data storage with no queries being performed.

Another approach is chosen by Damgård et. al [4]. The goal is to have the keys available during busy times but to securely store it during idle times.

In this work we chose a different approach. The tenant key is split into multiple parts that are distributed and stored separately by their owners. Owners are entities that are physically or logically separated. They only reveal their key part to one and the same trusted entity when data needs to be processed.

For three-tier web applications, there are three owners: application users, the application itself and the DB proxy. The DB proxy is the trusted entity. It composes the tenant key from these parts for a limited time. When the proxy is idle, it does not have access to this tenant key and only keeps a hash value of it.

III. Key Management Mechanism

The key management mechanism describes the process of how the keys for database access are prepared, distributed and used during runtime. The cryptographic operations for the creation of the database keys is not part of the key management mechanism and is taken for granted. Also the access control, performed by the cloud application, is not part of this mechanism. In the PaaSword project[1], where this key management mechanism is used, exists more detailed work about a context-sensitive access control.

A. Key Management Model

The PaaSword approach is based on an architecture that separates the application A where the data is processed from the DB proxy P whose task is to store and access the data in a cloud database (see Figure 1)[2]. The key management for data access shall avoid specific secure hardware. For security reasons, the tenant key TK to access the database shall not be stored at the DB proxy P where it would be beyond the tenant's control. On the other hand, the approach shall avoid the necessity of running a key server at the tenant side. Furthermore the tenant key TK shall not be available at the application A, so the application or its administrator cannot access the data at all times. In addition, not every individual user should have the tenant key TK, due to the high risk of losing the key or theft, especially if mobile devices (smartphone, tablet, laptop) are used. If a user would have the tenant key TK, he/she would be able to directly access the database, bypassing the access control mechanism of the application. In addition, we need one key per tenant, to share access to the data to all users authorized by the tenant.

To fulfill all the requirements mentioned above, the approach of PaaSword is to split up the key in three parts and give one part to the user U, one part to the application A and one part to the DB proxy P. Therefore the tenant key TK is split up in three parts TK_U, TK_A and TK_P such that

$$\mathsf{TK} = \mathsf{TK}_U \oplus \mathsf{TK}_A \oplus \mathsf{TK}_P$$

where $\oplus$ is the bit-wise XOR function. The user gets TK_U, the application TK_A and the DB proxy TK_P. Only if they work together they can reconstruct the tenant key TK to access the database.

To have the ability to withdraw the access possibility for an individual user or to change the user part (TK_U) of the tenant key (e.g. it is lost or stolen) without affecting any other user it is necessary to have user individual triple sets of the tenant key TK. Therefore the tenant key TK is split up user individual for every user U_i in a way that:

$$\begin{aligned} \mathsf{TK} &= \mathsf{TK}_{U_1} \oplus \mathsf{TK}_{A_1} \oplus \mathsf{TK}_{P_1} \\ &= \mathsf{TK}_{U_2} \oplus \mathsf{TK}_{A_2} \oplus \mathsf{TK}_{P_2} \\ &= \mathsf{TK}_{U_3} \oplus \mathsf{TK}_{A_3} \oplus \mathsf{TK}_{P_3} \\ &= \dots \end{aligned}$$

To create those user individual key triples, the tenant key TK is handled as a bit-string, the length of TK is ℓ. For each user U_i, initially two uniformly random bit-strings, e.g., TK_{A_i} and TK_{P_i}, of length ℓ are chosen. Then the third key, e.g., TK_{U_i} is computed as

$$\mathsf{TK}_{U_i} = \mathsf{TK} \oplus \mathsf{TK}_{A_i} \oplus \mathsf{TK}_{P_i}.$$

It is irrelevant which two keys are random and which is computed from the other two and TK.

B. Key Management Setup

To set up this scenario, the process shown in Figure 2 is used during bootstrapping time. The DB proxy P creates the encrypted database (with key TK), stores the hash $H(\mathsf{TK})$ of TK for validation purpose, hands the tenant key TK to the tenant admin and deletes TK from its memory. The tenant admin splits up the tenant key TK for every user as described above. He keeps TK and all TK_{U_i} in a safe place in case a recovery is necessary. Afterwards the tenant admin distributes

[1] https://www.paasword.eu

[2] The details of the database proxy P are not the focus of this work, more details about it can be found in [5].

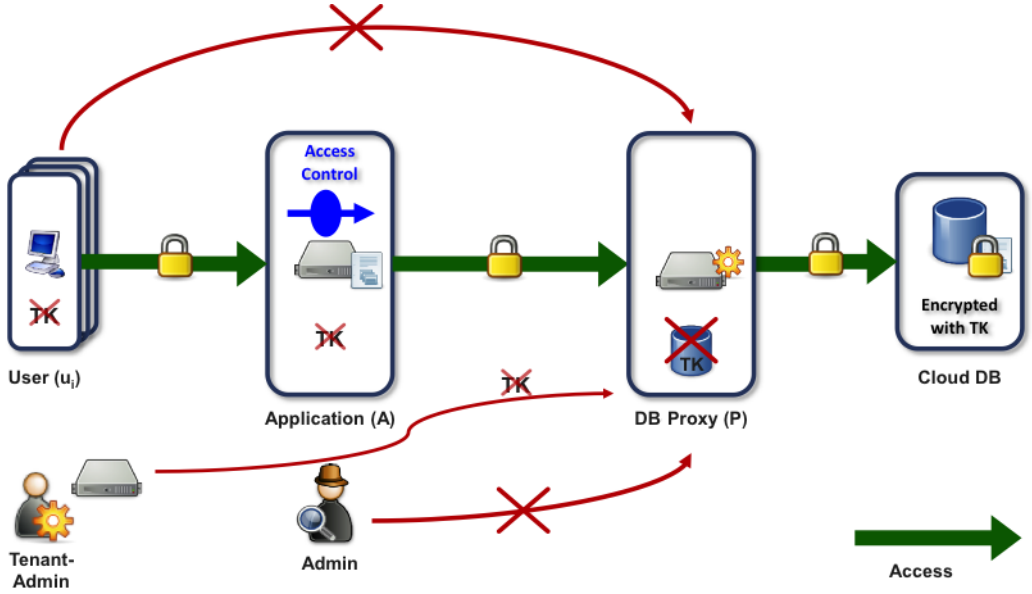

Fig. 1. Overview of Data Access in the PaaSword System.

the individual TK_{U_i} to every user (U_i), all the TK_{A_i} to the application A and all the TK_{P_i} to the DB proxy P. The distribution to application A and DB proxy P is secured with transport encryption and two-sided authorization. When the setup is finished, the tenant admin could go offline to prevent him from online attacks. He is only needed again in case of recovery or to add new user.

C. Key Management during Runtime

Figure 3 shows the PaaSword key management mechanism during runtime. The user (U_i) encrypts his individual part of the key (TK_{U_i}) together with a timestamp using the public key of the DB proxy P (referred to as $\mathsf{Enc}_P(\mathsf{TK}_{U_i}\|\mathsf{timestamp})$) and adds it to his/her request to the application for processing data. The application A controls the permission of the user to process the data and if it is granted the application A can use $\mathsf{Enc}_P(\mathsf{TK}_{U_i}\|\mathsf{timestamp})$ and its own user specific part of the application key TK_{A_i} to request the necessary database operations from the DB proxy P. The DB proxy P decrypts $\mathsf{Enc}_P(\mathsf{TK}_{U_i}\|\mathsf{timestamp})$ and controls the timestamp. If the timestamp is within the validity period, it reconstructs the tenant key $\mathsf{TK} = \mathsf{TK}_{U_i} \oplus \mathsf{TK}_{A_i} \oplus \mathsf{TK}_{P_i}$ and does the requested database operation. Afterwards P wipes TK from its memory.

D. Security assumptions

The confidentiality of the data in this process is based on the assumption that the DB proxy P is secure during execution time. Compromising the DB proxy P compromises the tenant key TK as soon as the first query is made. To restore the confidentiality, the database needs to be re-encrypted with a new tenant key.

The access control cannot be bypassed by a user U_i alone as TK_{A_i} would be missing to recover TK, but collaboration between a user U_i and an administrator of the application A allows temporary bypassing the access control. Together they have the necessary key parts to create any desired database request, but this does not compromise the tenant key TK.

To avoid replay attacks by the administrator of application A, timestamps are used to restrict the validity period of TK_{U_i}, but the administrator of application A is able to modify a user request to access other data than intended.

An external adversary in possession of the user credentials (UID_i, Login_i, TK_{U_i}) would be able to access whatever information the access control mechanism allows until the user is withdrawn or the tenant admin set up new user-individual keys.

To revoke a user from the system it's only necessary to delete one key part. This could be, for example, the part TK_{A_i} at the Application A where also the access rights are set. It is not necessary to destroy all copies of the user part.

E. Components of the Key Management Mechanism

The UML Component Diagram of Figure 4 depicts the main components that comprise the Key Management Mechanism. These components are further elaborated below.

As described above, the Tenant App is responsible for bootstrapping the key management. The first step is to create tenant certificates for signature and asymmetric encryption, which are validated by the Tenant Certificate Management component within the DB proxy P. The identity of the Tenant App should be verified by using a one-time password that is submitted separately (e.g. offline via phone, during a real-life meeting, etc.). Afterwards, the Tenant App can request the creation of a tenant specific encrypted database from the DB Access component. The DB Access component hands the tenant specific database key TK to the Tenant App, stores a hash of TK for validation during runtime and deletes its own copy of TK. Those two interfaces between Tenant App and

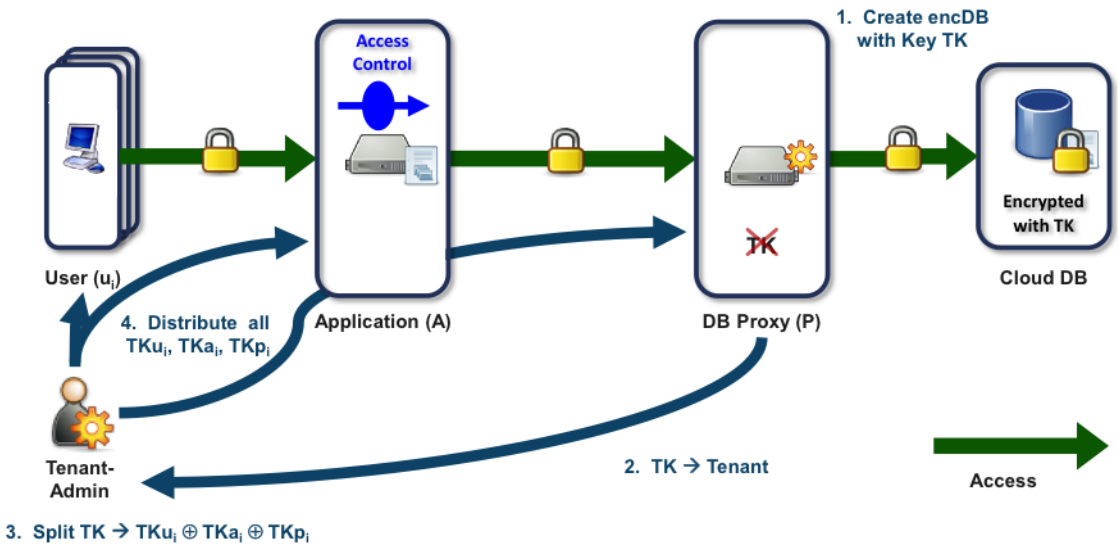

Fig. 2. Bootstrapping of the Key Management Mechanism.

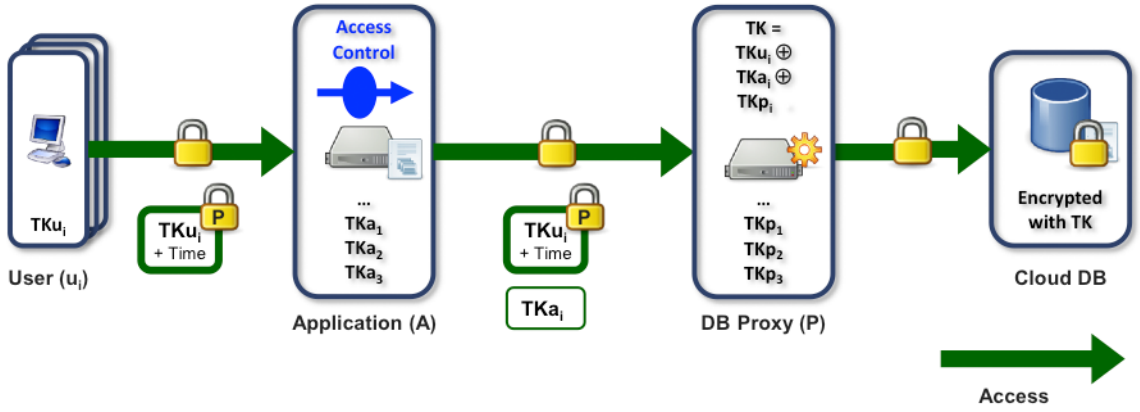

Fig. 3. Key Management Mechanism during Runtime.

DB proxy P are only needed once for bootstrapping and can be switched off afterwards.

The tenant administrator uses the Tenant App to create a User ID UID_i, user individual keys TK_{U_i}, TK_{A_i}, TK_{P_i} and stores TK and all TK_{U_i} as described above. The user keys TK_{U_i} are distributed to every individual user and all TK_{A_i}, $\mathsf{Enc}_P(\mathsf{TK}_{P_i}\|\mathsf{timestamp})$ are handed to the application A. Therefore, the Application User Key Management component of A offers an interface for the Tenant App to add, change or withdraw a user and its keys:

- $\mathsf{AddUser}(\mathsf{UID}_i, \mathsf{TK}_{A_i}, \mathsf{Enc}_P(\mathsf{TK}_{P_i}\|\mathsf{timestamp}))$
- $\mathsf{ChangeKey}(\mathsf{UID}_i, \mathsf{TK}_{A_i}, \mathsf{Enc}_P(\mathsf{TK}_{P_i}\|\mathsf{timestamp}))$
- $\mathsf{WithdrawUser}(\mathsf{UID}_i)$

To avoid manipulation by the administrator of application A, the encrypted keys TK_{P_i} for the DB proxy P are complemented by a timestamp and are signed by the Tenant App. UID_i, TK_{A_i} are stored at the application A and UID_i, $\mathsf{Enc}_P(\mathsf{TK}_{P_i}\|\mathsf{timestamp})$ are forwarded to the User Key Management of the DB proxy P. P decrypts $\mathsf{Enc}_P(\mathsf{TK}_{P_i}\|\mathsf{timestamp})$ and validates the timestamp and signature before TK_{P_i} is stored.

During runtime, every request to process data issued by a user U_i to the application A has added UID_i, $\mathsf{Enc}_P(\mathsf{TK}_{U_i}\|\mathsf{timestamp})$. To execute one user request, it can be necessary for application A to perform more than one database request. This makes it necessary that the validation of every request is valid for a short timeframe. The application A adds UID_i, $\mathsf{Enc}_P(\mathsf{TK}_{U_i}\|\mathsf{timestamp})$ and its part of the key TK_{A_i} to every database request which is needed to perform the user request. The DB Access component decrypts $\mathsf{Enc}_P(\mathsf{TK}_{U_i}\|\mathsf{timestamp})$ and controls the timestamp. If the timestamp is within the validity period, it reconstructs the tenant key $\mathsf{TK} = \mathsf{TK}_{U_i} \oplus \mathsf{TK}_{A_i} \oplus \mathsf{TK}_{P_i}$, validates it against the stored hash and does the requested database operation. Afterwards it wipes TK from its memory.

F. Key Recovery and Renewal

In a distributed architecture several entities can lose keys or be corrupted. Our scheme can cope with such data losses and

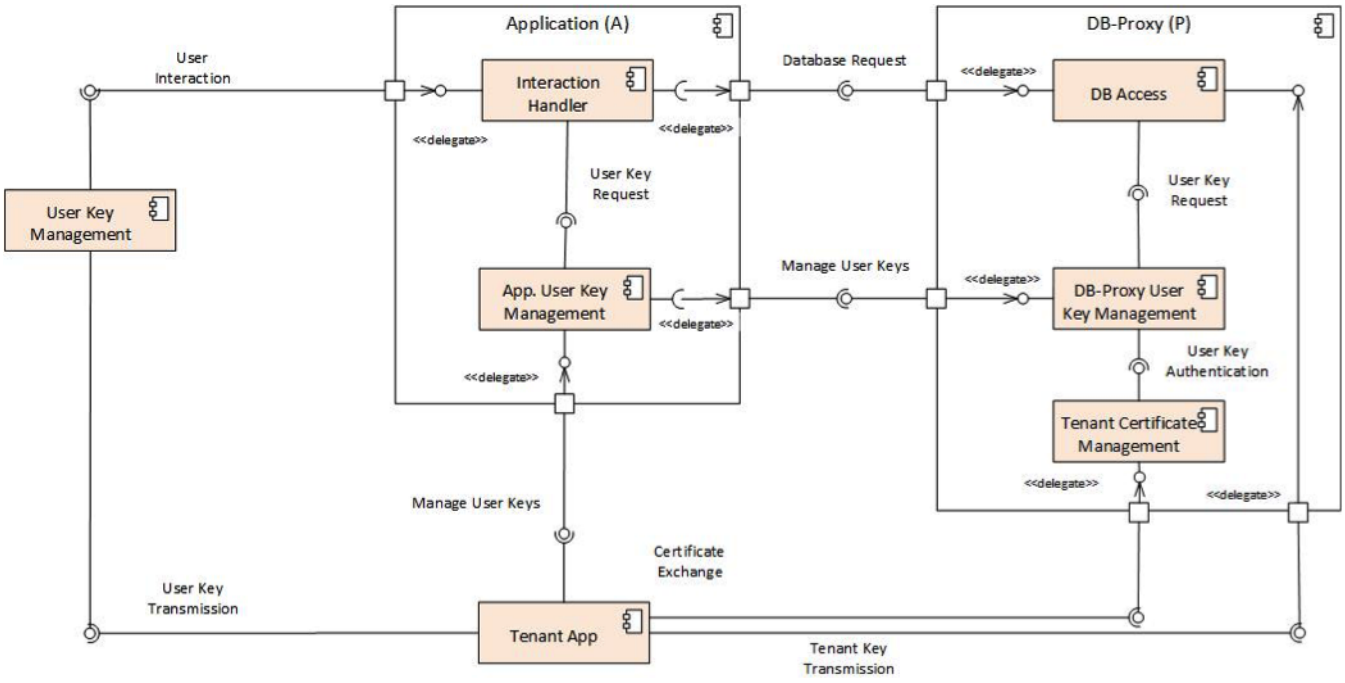

Fig. 4. Key Management Component Diagram.

attacks as long as the tenant admin is not affected.

The most likely case is that a user U_i loses his key part TK_{U_i}. Since the tenant admin is physically near the user, the recovery is very simple. After proper identification, the tenant admin just gives the user a copy of TK_{U_i} which he has stored on his machine. If a user key TK_{U_i} is compromised it is also very simple for the tenant admin to create new key parts for user U_i and distribute them to user U_i, application A and DB proxy P replacing the old ones.

In the unlikely case that the DB proxy loses the hash $H(\mathsf{TK})$ he can trigger a recovery where the tenant admin sends him $H(\mathsf{TK})$ again.

The only non-trivial case is when the application or the DB proxy gets compromised or loses user keys. The goal is to allow the users to keep their keys and only change the keys for the application and proxy while TK remains the same. The following pseudo code is executed for all users U_i:

- generate TK_{A_i} randomly
- set $\mathsf{TK}_{P_i} = \mathsf{TK}_{A_i} \oplus \mathsf{TK}_{U_i} \oplus \mathsf{TK}$ with a bit-wise XOR

After resetting all key shares for the users at the application and the proxy, TK can be reused because $\mathsf{TK} = \mathsf{TK}_{U_i} \oplus \mathsf{TK}_{A_i} \oplus \mathsf{TK}_{P_i}$ and the users can continue using TK_{U_i}.

IV. Discussion

The absence of need for secure hardware and key servers are important properties of the presented approach for distributed key management. These properties make it highly applicable in Software-as-a-Service (SaaS) scenarios, the common delivery model for business applications like CRM.

SaaS applications follow the client-server model. SaaS application servers are centrally hosted installations that are shared by all tenants. They are operated and maintained by the application provider and typically run in an environment provided as Platform-as-a-Service.

To implement sharing, these applications are built on top of platforms with a multi-tenant architecture. Multi-tenant architectures implement tenant separation at the data layer through either one individual database per tenant, one database schema per tenant or shared database tables. Clients are typically web clients for browsers, or mobile apps, that require only standard hardware and no or only minimal local installation by the user. Licenses are usually distributed in a self-service manner, i.e. new tenants and their database/data schema are automatically and dynamically created and provisioned upon user request. The requesting user is typically the administrator of the new tenant. After the tenant has been created, the typical workflow is that the tenant administrator logs in to configure his application instance, to create user accounts inside the tenant as required, to assign permissions to these accounts, and to make the credentials (UID_i, Login_i) available to people through a secure channel.

Securing SaaS applications through secure hardware or even key servers on the user's side would contradict their nature and greatly limit benefits of the SaaS model. The proposed approach for distributed key management allows providers to build secure SaaS applications while maintaining these benefits. This enables providers to create secure SaaS solutions, like the university CRM introduced in section I, that use encryption for tenant separation and to protect highly sensitive personal data.

More specifically, the proposed approach can be directly integrated into multi-tenant architectures that use either one

individual database per tenant or one database schema per tenant. The application platform's data layer must be modified to use the DB Proxy instead of the regular database adapter, while the logic layer must be extended to integrate the Interaction Handler and the Application User Key Management component (see Figure 4).

Existing platform mechanisms to create new tenants can easily adopt the creation of an encrypted database/database schema described in section III. Uploading the tenant certificate before the creation of the tenant's database/database schema and the secure transmission of TK to the tenant administrator are extensions to the tenant creation workflow that are easy to implement.

Clients must be extended to transmit $\mathsf{Enc}_P(\mathsf{TK}_{P_i} \| \mathsf{timestamp})$ with every request, and must integrate the User Key Management component. In the case of mobile apps, integration will usually be totally transparent to the user, as it can be implemented by modifying the mobile app. Web clients will use a portable and lightweight http-proxy.

The Tenant App is only required by the tenant administrator. Due to the simplicity of the key splitting operations, it can either be distributed as a lightweight portable tool or run inside a browser, using only client-side scripting for the key splitting and key encryption operations. User key distribution simply requires the tenant administrator to hand over (UID_i, Login_i, TK_{U_i}) instead of (UID_i, Login_i) over the secure channel of his choice.

Decisions regarding the secure storage of TK_{U_i} on the user side must generally be taken by the tenant administrator. In the most simple case, TK_{U_i} may be stored in an encrypted text file, protected by a key derived from a password only known to user U_i.

Due to the possibility to create and assign user individual keys, the tenant administrator himself can maintain the confidentiality of the encrypted data with the help of key renewal, even in case user keys are lost or corrupted, without intervention by the SaaS provider. Another feature of great importance for the practical applicability of our approach is the possibility of the tenant administrator to recover lost user keys.

V. Future Work

Future work will extend the usage of the distributed key management approach in two different directions. We consider NoSQL support as essential for upcoming areas of application like the Internet of Things, big data, and industry 4.0. It is therefore planned to integrate the distributed key management with encryption mechanisms for NoSQL databases.

We also consider developing a more fine-grained key splitting model that uses different keys for different database tables instead of whole schemas or complete databases. Considering the fact that there is usually a one-to-one mapping between the datatype of a business object and a database table, this level of granularity would enable the use of cryptographic keys to enforce an application's datatype permission system through key ownership.

Currently, datatype permissions are enforced only by application logic. In the example of the university CRM, all users are theoretically able to read and modify all encrypted tables of their tenant, because all tables are protected by the same key TK.

A strong guarantee that the exam result table is only accessible to members of the board of examiners is a desirable property of the university CRM. Today, only the datatype permission system of the CRM platform prevents, for example, clerks from accessing the exam result table. In case of bugs in the code that implements the datatype permission system, there is a possibility that exam result table could be accessed by clerks, even though it is encrypted. Using different keys for different tables, only the board of examiners would receive key parts required to access the exam result table. Clerks would still be able to use the university CRM, but are unable to access or even manipulate exam results, as they do not possess the required key.

Acknowledgements

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644814, the PaaSword project (www.paasword.eu) within the ICT Programme ICT-07-2014: Advanced Cloud Infrastructures and Services.

References

[1] Amazon Simple Storage Service Developer Guide (API Version 2006-03-01) Available online at: http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingEncryption.html. Last Accessed 23-08-2016.

[2] Amazon Web Services, Inc. Amazon EBS Encryption. Available online at: docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSEncryption.html. Last Accessed 23-08-2016.

[3] AWS Key Management Service (KMS) Available online at: https://aws.amazon.com/en/kms/. Last Accessed 23-08-2016.

[4] Ivan Damgård and Thomas P. Jakobsen and Jesper Buus Nielsen and Jakob I. Pagter. Secure Key Management in the Cloud. IMA Int. Conf. 2013: pp. 270–289, 2013.

[5] R. Dowsley, M. Gabel, K. Yurchenko, V. Zipf. A Database Adapter for Secure Outsourcing. Manuscript, 2016.

[6] European Parliament and the Council. General Data Protection Regulation. Regulation (EU) 2016/679 of the European Parliament and of the Council. Available online at: http://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679

[7] Fraunhofer Institute for Secure Information Technology. OmniCloud Available online at: http://www.omnicloud.sit.fraunhofer.de/index_en.php, 2016. Last Accessed 19-06-2016.

[8] H. A. Jäger, A. Monitzer, R. O. Rieken and E. Ernst. A Novel Set of Measures against Insider Attacks - Sealed Cloud. In Detlef Hühnlein (ed.), Heiko Roßnagel (ed.), *Lecture Notes in Informatics - Open Identity Summit 2013*, pp. 187–197, Gesellschaft für Informatik, Bonn, 2013.

[9] J. Lehner, A. Oberweis, G. Schiefer. Data protection in the Cloud – The MimoSecco Approach In Helmut Krcmar, Ralf Reussner, Bernhard Rumpe, *Trusted Cloud Computing*, pp. 177-186, Springer, Heidelberg, Januar 2015.

[10] NightLabs Consulting GmbH. Cumulus4j - Securing your data in the cloud - Deployment scenarios. Available at: www.cumulus4j.org/latest-stable/documentation/deployment-scenarios.html. Last Accessed 23-08-2016.