Versatile Key Management for Secure Cloud Storage

Sebastian Graf
Distributed Systems Group
University of Konstanz
Konstanz, Germany
sebastian.graf@uni.kn

Patrick Lang
Distributed Systems Group
University of Konstanz
Konstanz, Germany
patrick.3.lang@uni.kn

Stefan A. Hohenadel
University of Konstanz
Konstanz, Germany
stefan.hohenadel@uni.kn

Marcel Waldvogel
Distributed Systems Group
University of Konstanz
Konstanz, Germany
marcel.waldvogel@uni.kn

***Abstract*—Not only does storing data in the cloud utilize specialized infrastructures facilitating immense scalability and high availability, but it also offers a convenient way to share any information with user-defined third-parties. However, storing data on the infrastructure of commercial third party providers, demands trust and confidence. Simple approaches, like merely encrypting the data by providing encryption keys, which at most consist of a shared secret supporting rudimentary data sharing, do not support evolving sets of accessing clients to common data. Based on approaches from the area of stream-encryption, we propose an adaption for enabling scalable and flexible key management within heterogeneous environments like cloud scenarios. Representing access-rights as a graph, we distinguish between the keys used for encrypting hierarchical data and the encrypted updates on the keys enabling flexible join-/leave-operations of clients. This distinction allows us to utilize the high availability of the cloud as updating mechanism without harming confidentiality. Our graph-based key management results in an adaption of nodes related to the changed key. The updates on the keys again continuously create an overhead related to the number of these updated nodes. The proposed scalable approach utilizes cloud-based infrastructures for confidential data and key sharing in collaborative workflows supporting variable client-sets.**

I. Introduction

Storing data in the internet is more or less a synonym for *storing data in the cloud*. *Google*, *Amazon* or *Microsoft* as *Cloud-Service Providers*(CSPs) provide a specialized set of products satisfying any needs of customers and providers. *CSPs* thereby have full access to any information stored on their infrastructure even though some of them offer encryption performed directly on their infrastructure.

Simply encrypting the data before upload guards the information against unauthorized access. Satisfying common understandings of security (e.g. the *NIST*-definition [1]), encrypting works well for a limited amount of accessing users due to the necessary sharing of a common secret. The flexible utilization of cloud storages results in collaborative workflows where different users work on common data. Shared secrets neither offer secure ways to support such workflows nor utilize the availability and scalability of cloud-based services since changes within the set of authorized clients result in complex re-encryption operations and the distribution of new shared secrets to all authorized clients. Versioning of the data provided by multiple *CSPs* further complicates the key handling since the access to specific versions relies on corresponding specific shared secrets.

Our approach tackles the challenge of managing access rights upon shared versioned data on cloud infrastructures for a restricted, flexible group with the help of the following techniques:

- Disjunct clients share common data utilizing hierarchical organized access rights. The hierarchy related to these access rights relies on a *Directed Acyclic Graph*(DAG) where *Encryption Keys*(EKs) represent group-keys and summarize disjunct clients represented by *Client Keys*(CKs).
- Updates on the keys ongoing with changes on the set of authorized clients occur encrypted and scalable based upon well-established approaches from the area of stream-encryption.
- Access rights are applied to any stored element within past versions, the current version or future versions.

Our approach consists of three components, namely a global *Key Graph* stored on a trusted third party environment, encrypted updates on the *Key Graph* as well as versioned data, both stored encrypted in the cloud. The *Key Graph* relies on existing graph-based key management approaches namely *VersaKey* [2] extended as a *DAG* similar to [3]. This approach binds key material to nodes related to each other representing the *DAG* where the source nodes (represented by the "Client Keys"(*CKs*) with the *Key Graph*) constitute the client rights and the terminal nodes represent the most common access rights called "Encryption Keys"(*EKs*).

Another adaption on *VersaKey*, includes the persistence of the updates applicable on the *Key Graph*. These updates, denoted as *Key Trails*, are not only broadcasted to the clients once but stored in the cloud for on-demand updates of the client keys as well. Similar to *VersaKey*, the nodes are versioned whereas each version of each node contains unique key material to decrypt an element of the versioned data.

Any modification of the *Key Graph* results in an update of all reachable *EKs* originating from the adjusted access right represented by single nodes within the *Key Graph*. The update includes the generation of new key material for these nodes broadcasted over *Key Trails*. The resulting *Key Trails*, consisting of the fresh key material encrypted with the related

valid nodes, scale with the number of updated nodes since each *Key Trail* relies on an edge incident to the updated nodes. Instead of updating all shared keys, we therefore only update the summarizing groups. We furthermore extend the *VersaKey*-approach to utilize the version stored within the nodes to provide temporal-aware access rights represented by past versions of the data, the current version of the data and future versions of the data.

By applying stream-based key management to versioned data, we extend well-established graph-based key management schemas and utilize the generation of encrypted key updates by storing these *Key Trails* on high available and scalable but untrusted cloud-infrastructures parallel to the encrypted data.

II. Related Work

Related approaches in this area cover the scalable handling of keys with the help of *Key Graphs* combined with encrypted data on untrusted components represented by cloud-infrastructures.

Sato [4] proposes a trust model for secure cloud usage. The proposed model contains key management functionality although no concrete key management approach was described. Damiani and Pagano [5] propose an hierarchical organization of the keys used for encrypting and storing data on the cloud. The exclusion of existing users is performed by propagating new keys after re-encryption of the data. Xu [6] proposes the separation of content and format as a base for storing data secure on the cloud. The data is encrypted by public/private keys making collaborative key handling obsolete. Lou et al [7] proposes a schema similar to ours despite the fact, that their approach is in need of re-encrypting the data within changes of the authorized users whereby this task is delegated to the *CSP* as well. Capitani et al [8] offers a model for key management for untrusted storages similar to ours. Keys are re-encrypted to distribute the ability to access the data over the provider and the user whereas we re-encrypt the keys to offer a secure way to propagate updates over the cloud.

Storage upon untrusted components always needs sophisticated approaches to grant disjunct, fixed defined users access to common data without exposing any information about the underlaying group management. *Cryptree* [9] represents an approach to store data in an hierarchical manner with permission-rights on subtrees mapped on groups. The underlaying recursive data-structure scales with the numbers of keys since the keys are inherited top-down in the tree. The focus of *Cryptree* is similar to ours since we rely on hierarchical group permissions suitable for hierarchical data structures as well. Our approach extends *Cryptree* by versioning all keys and the data. The access of former versions utilizes therby the distributed environment while the distribution of the keys leverages from the availability of the *CSPs*.

Multiple approaches exist to map key management to graphs. Waldvogel [2] proposes the arrangement of client-bound keys to an overall encryption key within a tree-structure called *VersaKey*. We use the model of this approach for updating the nodes.

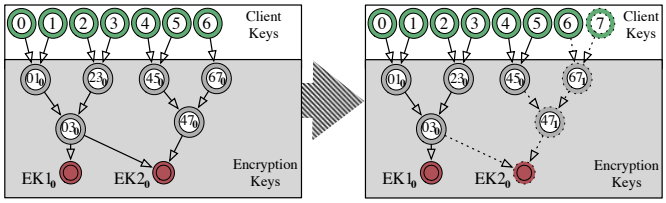


Fig. 1: Classic Key Graph

These approaches are extended to offer an even more efficient key handling by Wong [10] where the key graph propagates any changes via UDP/IP multicast. Forward error correction reduces the messages for efficient and reliable key updates.

Hassen [11] proposes another extension by introducing intra-level changes on the tree. This approach enables key graphs to change the affiliation of a node to a group.

The *EKs* in these approaches ensure scalability within join-/leave-operations of clients and utilize their keys for direct encryption within our approach. The resulting structure is not a tree anymore but a *DAG* as proposed in [3], [12].

In our approach, we rely on this architectural style of modeling hierarchical access rights into a *DAG*-structure. Even though the scaling of the *DAG* degenerates within consecutive changes on the keys, the combination of nodes to reduce the *DAG* to a more efficient *DAG*-representation as proposed in [13] stays out of focus since we use the hierarchy within the *Key Graph* as semantic representation for organizational issues.

III. Graph-based Key Management for Cloud Storage

Figure 1 shows a *DAG* constructed similar to classic stream-bound approaches [3], [12], [13].

Any data is encrypted with the help of *EKs*. To ensure scalability within updates, *CKs* are combined with the help of the *EKs*. Each client contains a subgraph consisting of its own *CK* and the descendants whereas one global *Key Graph* manages join-/leave-operations of nodes as well as insert-/remove-operations of edges. If a client, represented by a *CK*, joins or leaves the set of authorized clients, only parts of the keys stored within each client must be updated. These parts include the descendants of modified nodes.

Fig. 1 shows the insertion of the client 7 with the global *Key Graph*. As a consequence of the insertion, the nodes 67, 47 and $EK2$ have to update their key material to ensure that the new client has the ability to access the data encrypted by the descendants of its *CK*. Since each node contains a version counter, represented by the number in the subscript of the actual node identifier, the version of the updated nodes increases.

Based on this graph-representation, *VersaKey* encrypts the new key material of the updated descendants with the keys stored in the adjacent nodes staying valid after the modification. These encrypted updates, represented by the edges within the *Key Graph*, are called *Key Trails*. One *Key Trail* thereby

is represented by an edge connecting two nodes (representing single rights). The updates are propagated in a secure manner by encrypting the *Key Trails*. For each dotted line in Fig. 1, one *Key Trail* is computed as update e.g. $E_{67_1}(47_1)$ where the updated node 47 is encrypted with the updated node 67.

A. Key Graphs and Data Storage

VersaKey is originally applied to stream-based architectures whereas access to former encrypted data is not necessary. Regarding the usage of evolving *Key Graphs* for encrypting data within storage, three adaptions must be made to apply *VersaKey* on data storage:

1) The data to be encrypted is versioned. Stream-based encryption abdicates the availability of former keys and former data. The keys as well as the encrypted data are only valid within a given point of time. Regarding data storage, the sustainability of the data to be encrypted must respect the changes within the key management ongoing within single join-/leave-operations. The gained awareness is achieved by versioning the data to be encrypted.
2) The *Key Graph* is versioned equivalent to the versioning of the data. Since the key material changes regarding different versions of the same node, all former keys from the *Key Graph* must be available to ensure access to all versions of the data. After each update on the *Key Graph*, all modified nodes are therefore stored related to their version.
3) The updates on the *Key Graph* occur only over *Key Trails*. Since we rely on a versioning of the *Key Graph*, we use the *Key Trails* not only for on-the-fly adapting of the *Key Graph* but also as format for deltas between two versions on the *Key Graph*. The set of *Key Trails* must therefore respect the versions of the incident nodes. The encrypted nature of the *Key Trails* is utilized to store updated key material in the cloud as explained in Sec. III-B.

The defined adaptions result in a versioning of the *Key Graph* independent from the versioning of the data. The data is encrypted with the key material of the most recent version of a suitable node. The decrypting of the data is provided by the version of the corresponding node at the point of time of modification. Old versions of the data are thereby only able to be decrypted with fitting versions of *Key Graph*-nodes while the current version of the *Key Graph*-nodes encrypts ongoing modifications. Because of the binding of node-versions to data-versions, re-encrypting the data within key changes is not necessary. Figure 2 shows an evolving *Key Graph*, the related *Key Trails* and an hierarchical data-structure.

CK $u2$ leaves the group $g1$ updating the same nodes like the insertion of the new *CK* $u0$. The version of the nodes $g1$ and p is thereby increased twice. The join of the existing *CK* $u1$ to the group $g2$ however results in the update of the nodes $g2$ and p whereas $g1$ stays unmodified. The updated nodes receive new key material and increase their version number. The old versions of the nodes stay accessible containing their original key material and the corresponding pointers to their environment.

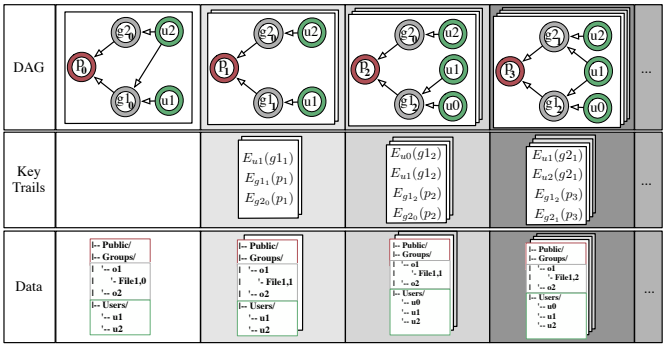


Fig. 2: Evolving Data, *Key Trails* and *DAG*

Independent from the updates of the *Key Graph*, the data undergoes modifications. The keys on higher-levels in the *Key Graph* encrypt higher-level elements within the hierarchical data to benefit from inherited access rights similar to *Cryptree* [9]. The related data encrypted with these nodes must be aware of the current version of the *Key Graph* within each modification. Fig. 2 shows the change on the *Key Graph* where the *CK* $u2$ is excluded from the group $g1$ occuring before the modification of the file "File1". The *EK* used for encrypting the new version of "File1" is based upon the new version of the node $g1$. If the modification on the data occurs before the update of the *Key Graph*, $u2$ has still access to the related version of $g1$ and therefore "File1".

All modifications on the *Key Graph* result in the generation of the *Key Trails* depending on the set of modified nodes. The *Key Trails* represent retrievable deltas replaying any changes upon the *Key Graph* whereas the *Key Trails* are versioned as well. Related to the example represented by Fig.2, *CK* $u0$ has access to the most recent *EK* p by decrypting the *Key Trail* $E_{u0}(g1_2)$ and, with the resulting access on the most recent node version of the node $g1$, the *Key Trails* $E_{g1_2}(p_2)$ and $E_{g1_2}(p_3)$. With the help of the *Key Trails*, any version of the *Key Graph* can be reconstructed out of a former version of the *Key Graph* depending on the initial set of nodes.

B. Synergies between the Cloud and the Key Management

The aim of the appliance of our approach to cloud-based infrastructures is to utilize the high available but untrusted [14], [15] components within the cloud.

We therefore distribute the *Key Graph*, the data and the *Key Trails* over different components as shown in Fig. 3:

- The key management is provided by two types of *Key Graphs*:
  - A centralized instance upon a trusted component represents the overall *Key Graph* called *Key Manager*. The *Key Manager* contains all nodes within all versions and triggers all changes upon the authorized client-set.
  - Besides the centralized instance containing all valid keys, each client holds its specific *CK* representing its specific access rights in the defined versions.
- The *Key Trails*, fully encrypted by default, are stored in the cloud. Due to the high availability and the scalability

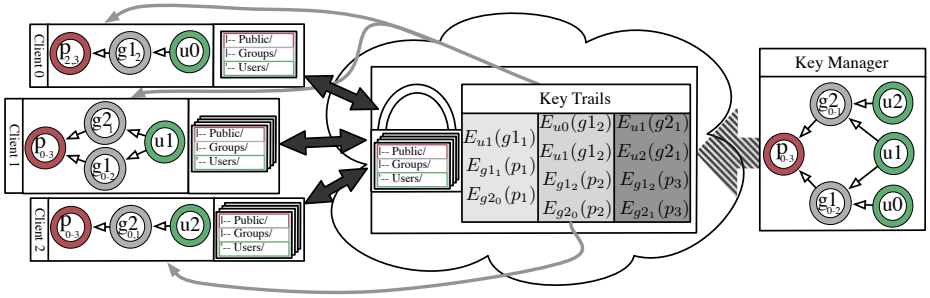


Fig. 3: Overview of proposed architecture

of the cloud-based services, the *Key Trails* stay accessible for any accessing client even if the centralized *Key Manager* is not available. Since the key material in the *Key Trails* is encrypted, the *CSP* has no ability to access any encrypted data in the cloud. The cloud is only utilized for storing the updates and offer easy propagation of the *Key Trails* to the clients.

- The data itself is stored encrypted in the cloud as well.

In Fig. 3, the *Key Manager* handles changes within the set of authorized clients and computes the *Key Trails* pushed in the cloud afterwards. The clients update their keys (if necessary and possible) while the black arrows denote the transfer of any data in/from the cloud. The access on the data is thereby bound to the keys stored within the clients. *Client 0* as one example has only access to one version while the other clients cache multiple versions since the related *CK* 0 was introduced later in the global *Key Graph*.

IV. Versioned-based Access

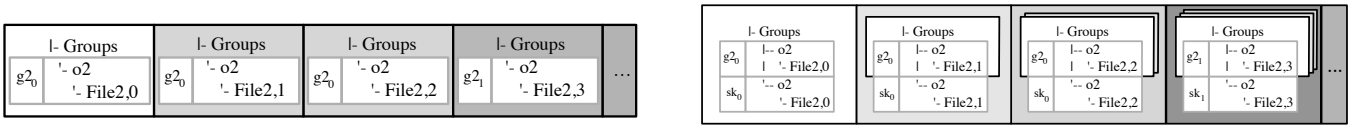
Even though the modifications on the *Key Graph* are independent from the modifications on the data, one node of the *Key Graph* in one version must map to one data element in one version. Depending on the granted access rights, a client might have the non-exclusive access on former versions of the data, on the current version of the data or on future versions of the data. To offer version-granular access on the data, an extension to our approach must be made, since one version of an *EK* has the ability to cover multiple versions of the data. If a client should gain access only to the current status without the ability to read former versions, simple sharing the related node violates this restriction.

Figure 4a describes this problem: Consecutive modifications on the "File2" are encrypted with one version of $g2$. Based upon the usage of one *EK* to protect multiple versions, the sharing of this key results in the access of all guarded versions. A *CK* joining $g2$ within its first version automatically has the ability to access all versions of "File2". An access to only version 2 of the data is not supported by the classic *VersaKey*-mapping. Sections IV-A and IV-B describe two possible extensions to our approach offering version-granular access even on former versions of the data.

A. Shadow Structure

The first proposed extension to our approach restricting access to only the recent version is represented by the insertion of additional structures related to the *Key Graph* and the data called shadow structures within the rest of the paper. While the shadow structure of the *Key Graph* needs to be versioned similar to the *Key Graph* itself containing own keys, the shadow structure of the data is a clone of the data within its most actual version. All modifying requests on the data are thereby encrypted by a node from a shadow-*Key Graph* called *Shadow Key* and applied to an unversioned shadow structure of the data called *Shadow Data*. This operation takes place simultaneously to the described encryption of the data performed by the *Key Graph* for the versioned data storage. The *Shadow Key* is implemented as extension to the original *Key Graph* containing additional key material stored within each node of the *DAG* to access only the *Shadow Data* consisting of only the most recent version of the data. Since the *Shadow Data* consists of only the current version of the data, no access to former data versions is provided. As a result, the *Shadow Key* is distributed and updated the same way as the normal *Key Graph* including the storage of corresponding *Key Trails* related to the *Shadow Key* in the cloud.

Figure 4b represents an example of the shadow structures. While the data is encrypted and versioned with the help of the *EKs*, the most recent modification is also applied to a copy encrypted by the *Shadow Key*. Therefore, the access to "File2" is provided not only via $g2$ but via the related *Shadow Key*. Even if $g2$ within its version 0 guards several versions of the data, the related *Shadow Key* guards only the most recent version stored in the *Shadow Data*. If a client should only access the latest version, the *Shadow Key* is published in version 0 and suitable *Key Trails* are provided to reconstruct its version 1. Besides, $g2$ is only provided within its most recent version. The client, accessing the *Shadow Key* within all of its versions, has the ability to access the *Shadow Data*. The provided $g2$ with its latest version offers no access to former versions of the normal data. Therefore, the client has only access to the most recent version of "File2": The access to the versioned storage is only provided by non-accessible versions of $g2$ while the *Shadow Data* is encrypted with the accessible versions of the related *Shadow Key*.



(a) Encrypting different versions of the same data

(b) Shadow structure to provide access to current version

Fig. 4: Handling of past versions

B. Token-based Extension

Another mechanism for restricting the access to defined former versions on the client is the deployment of an authorization layer within the distributed environment. The *Key Manager* contains a list of resources applied to the nodes. This mapping between the nodes and the data is enriched by the versions being valid for the different clients. For each client, the resources in the data encrypted with a node are bound to versions. The additional authorization structure is deployed additionally to the global *Key Graph* within the *Key Manager*. The binding between the versioning of the *Key Graph* and the data is represented by this structure.

Figure 5 shows an example: "Client 0" has access to version 3 of $g1$ whereas "Client 1" has the ability to access all versions stored under the same *EK*. Since the mapping between *DAG* and data takes place by a dedicated structure, each client contains all descendants of the own *CK* in all versions. Related to the example of Fig. 5, "Client 0" contains all versions of $g0$ since the authorization for different versions is not in need of the *Key Graph*.

Each descendant of each *CK* represents at most one rule mapping the versions of the data to the versions of the *Key Graph* for the accessing clients. The rules are thereby not bound to the different versions of one node but to the node itself. Regarding the example of Fig. 5, "Client 0" contains three descendants of the own *CK* resulting in at most three rules represent the version-mapping between "Client 0" and the *Key Graph*.

The proposed access-structure acts as a base for a token-based approach including the cloud, the disjunct clients and the centralized *Key Manager*. The workflow is denoted by Fig. 5. The client requests a version. The requested access is verified against the proposed authorization structure within the *Key Manager*. Based upon the versions allowed for this client and the requested resource, a token is negotiated between the *Key Manager* and the cloud-instance. The token is encrypted and only readable for the cloud and the *Key Manager* representing a single rule for a dedicated client. After negotiation, the resulting token is sent to the client. The client is not able to decrypt the content neither to modify the content without violating it. With every request, bound to the fixed resource within one of the desired versions, the token must be delivered to the cloud-instance from the client. The cloud has the ability to decrypt the token and to verify the access on the requested version of the data.

Due to the encryption of the modifications with the help

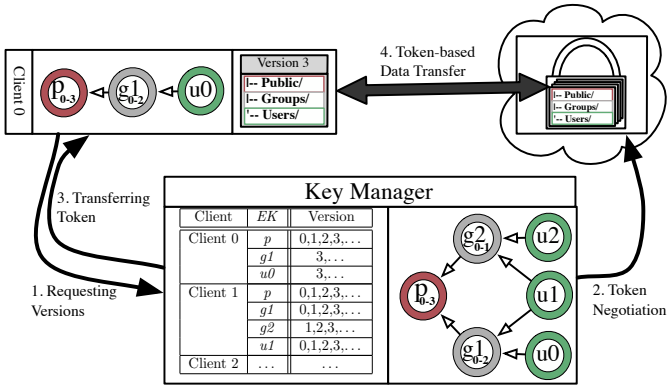


Fig. 5: Token-based extension

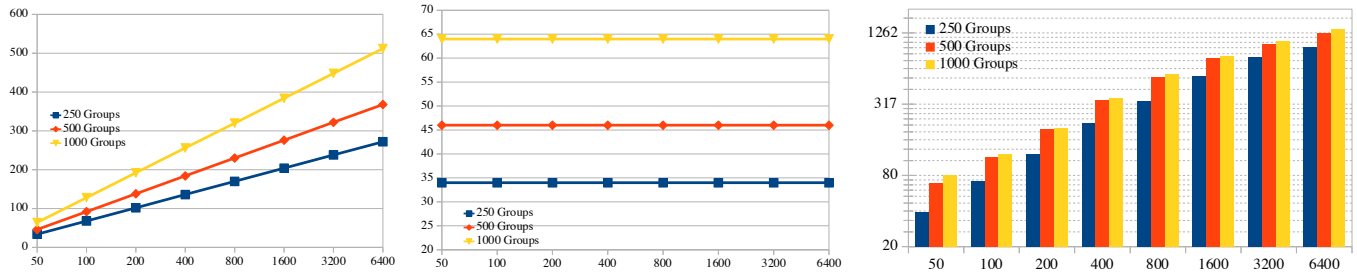
of the different *EKs*, the awareness of the data by the *CSP* is constricted only to the versioning and not to the data itself. The client has the ability to encrypt all data stored in the cloud with the help of the decrypted *Key Trails* resulting in a subgraph of the global *Key Graph*. Based upon the authorization structure mapping the versions of the *Key Graph* to the versions of the data, the access on the encrypted data in the cloud is guarded additionally to the encryption by the *Key Graph*.

V. Implementation, Evaluation and Scaling

The proposed approach was implemented within the secure storage system *Treetank* [16], [17] as extension containing a random generated *DAG*. The *DAG*, representing our *Key Graph*, is generated randomly with 250, 500 and 1000 nodes and consists of 10 levels whereas these parameters are chosen arbitrarily to benchmark the scaling of our approach. While the *EKs* are distributed equally on maximal 10 levels, 8 terminal nodes are included in this set of *EKs*. The outdegree of each node, except the terminal nodes, is at most 3, meaning that each node has at most 3 children. The indegree of the *EKs* in opposite is not restricted.

Incrementally, 6400 different *CKs* are deployed to the resulting *DAG*. Each *CK* is linked to between 1 and 3 random selected *EKs*. After each *CK*-insertion, a new version for all descendants of the inserted *CK* is created. After a fixed number of insertions (50, 100, 200, 400, 800, 1600, 3200 and 6400), the generated *Key Trails* and the updated nodes are traced.

Within the insertion of single *CKs*, only a constant number of related *EKs* is updated as shown in Fig. 6b since the edges between the *EKs* are already existing before the insertion of the *CKs*.



(a) Nodes updated in all versions (b) Nodes updated in one version (c) Number of *Key Trails* per user

Fig. 6: Nodes updated within modifications

Figure 6a shows the scaling of updated nodes cumulated within all versions whereas the y-axis again denotes the number of *EKs* updated and the x-axis represents the *CKs* joining the *DAG*. The number of nodes scales linear with the number of *CKs* inserted as expected. This scaling substantiates our assumption that only a constant number of nodes is updated within each *CK*-insertion.

Figure 6c shows the number of *Key Trails* cumulated over all versions whereas the y-axis represents the *Key Trails* generated (scaled logarithmically) and the x-axis denotes the *CKs* joining the *DAG*. Logically, the more *CKs* are introduced in the *DAG*, the more *Key Trails* are generated. Since the *Key Trails* are computed based upon the incident edges on the modified nodes, the scaling is linear. Any modification on the *DAG* results in only a constant number of updated nodes, namely the descendants of the modified node.

VI. Conclusion and Future Work

Within our approach, we successfully bring stream-based *Key Graph*-approaches to the area of cloud storage. Our proposed distributed architecture versions not only the data but also the *Key Graph* enabling changes within accessing clients without the need of re-encrypting any data. Modifications on the *Key Graph* update the descendants of the modified node. The updates themselves are introduced as *Key Trails* representing the edges within the *Key Graph*. Since the *Key Trails* are encrypted and stored, we use the high availability of untrusted cloud-based services propagating any changes within the clients. The access to former versions is provided either by a separate shadow-structure of the data and the *Key Graph* or by utilizing the distributed architecture of our approach. Even though this enables access to former versions within new clients, we believe that in this area more sophisticated ideas can be developed by utilizing the distributed architecture. Further improvements of our approach include the distribution of the key management to make the centralized *Key Manager* obsolete similar to the original *VersaKey*-approach. Since we update the *Key Graph* manually, we further evaluate function-based adaptions of updated nodes making the manual key generation within each node obsolete and utilizing the difference between join- and leave-operations of *CKs* similar to *VersaKey*. Utilizing the keys within a versioned storage offers us furthermore an inclusion of higher level security goals like *non-repudiation* [1] when equipping the *Key Graph* with a node-unique signature signing all version on the data.

VII. Acknowledgments

We would like to thank Anna Dowden-Williams for her more than valuable input.

References

[1] G. Stoneburner, "Underlaying technical models for information technology security," *National Institute of Standards and Technology*, 2001.

[2] M. Waldvogel, G. Caronni, D. Sun, N. Weiler, and B. Plattner, "The VersaKey framework: Versatile group key management," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 9, pp. 1614–1631, Sep. 1999.

[3] C. K. Wong, M. Gouda, and S. S. Lam, "Secure group communication using key graphs," *IEEE/ACM Transaction on Networking*, vol. 8, no. 1, 2000.

[4] H. Sato, A. Kanai, and S. Tanimoto, "A cloud trust model in a security aware cloud," in *Applications and the Internet '10*, 2010.

[5] E. Damiani and F. Pagano, "Handling confidential data on the untrusted cloud: An agent-based approach," in *Cloud Computing '10*, 2010.

[6] J.-S. Xu, R.-C. Huang, W.-M. Huang, and G. Yang, "Secure document service for cloud computing," in *ClouCom '09*, 2009.

[7] W. Lou, K. Ren, C. Wang, and S. Yu, "Achieving secure, scalable, and fine-grained data access control in cloud computing," in *Proceedings of the 2010 Infocom Conference*, ser. Infocom '10.

[8] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Over-encryption: Management of access control evolution on outsourced data," in *Proceedings of the 2007 VLDB Conference*, ser. VLDB '07.

[9] D. Grolimund, L. Meisser, S. Schmid, and R. Wattenhofer, "Cryptree: A Folder Tree Structure for Cryptographic File Systems," in *25th IEEE Symposium on Reliable Distributed Systems (SRDS), Leeds, United Kingdom*.

[10] C. K. Wong and S. S. Lam, "Keystone, a group key management service," in *International Conference on Telecommunications*, 2000.

[11] H. R. Hassen, A. Bouabdallah, and H. Bettahar, "A new and efficient key management scheme for content access control within tree hierarchies," in *Advanced Information Networking and Applications Workshops*, 2007.

[12] Y. Sun and K. R. Liu, "Scalable hierarchical access control in secure group communications," in *Proceedings of the 2004 IEEE Infocom*, 2004.

[13] Q. Zhang, Y. Wang, and J. P. Jue, "A key management scheme for hierarchical access control in group communication," *International Journal of Network Security*, vol. 7, no. 3, pp. 323–334, 2008.

[14] A. Lenk, M. Klems, J. Nimis, S. Tai, and T. Sandholm, "What's inside the cloud? an architectural map of the cloud landscape," in *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, ser. CLOUD '09.

[15] P. Mell and T. Grance, "The nist definition of cloud computing," *National Institute of Standards and Technology*, vol. 53, no. 6, 2009.

[16] S. Graf, M. Kramis, and M. Waldvogel, "Treetank: Designing a versioned XML storage," in *XMLPrague'11*, 2011.

[17] S. Graf, "A secure cloud gateway based upon xml and web services," in *ECOWS'11, Phd Symposium*, 2011.