Versatile Key Management for Secure Cloud Storage

Sebastian Graf Patrick Lang Stefan A. Hohenadel Marcel Waldvogel

Distributed Systems Group
University of Konstanz
(sebastian.graf|patrick.3.lang|stefan.hohenadel|marcel.waldvogel)@uni-konstanz.de

Abstract

Storing data on cloud-based infrastructures facilitates infinite scalability and all-time availability. Putting data in the cloud additionally 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. Often 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 well-established approaches regarding 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 any confidentiality. Our graph-based key management results in a constant adaption of nodes related to the changed key. The updates on the keys generate a constant overhead related to the number of those updated nodes. The proposed scalable approach utilizes cloud-based infrastructures for confidential data and key sharing in collaborative workflows supporting variable client-sets.

Categories and Subject Descriptors K.6.5 [*Security and Protection*]: Unauthorized access; C.2.4 [*Distributed Systems*]: Cloud Computing

General Terms VersaKey, Key Handling, Cloud, Directed Acyclic Graph

Keywords Encryption, Confidentiality, Cloud Service

1. Introduction

Storing data in the internet is more or less a synonym for *storing data in the cloud*. Regardless the service itself being a cloud-service or not, the probability that the underlaying application runs on dedicated hardware decreased within the last years. Cloud-based services provide customers as well as providers with scalable ways to utilize any kind of service without the need to care about the underlaying concrete infrastructure. *Google*, *Amazon* or *Microsoft* as *Cloud-Service Providers*(CSPs) provide a specialized set of products satisfying any needs of customers and providers. The *CSPs* have as a result full access to any information stored on their infrastructure demanding direct storage of any confidential data.

Most applications simply solve this issue by encrypting the data. This straight-forward approach satisfies common understandings of security (e.g. the *NIST*-definition [8]) and works well for a limited amount of accessing users. The sharing of easy accessible data for disjunct users is intensified due to the availability of modern mobile devices enabling users to access their cloud-based data from anywhere. The flexibility ongoing with the mobility 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 like 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 based upon 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.

These three components, namely a global *Key Graph* stored upon a trusted third party environment, encrypted updates on the *Key Graph* stored on the infrastructure of any *CSP* and versioned data, stored encrypted in the cloud, rely mainly on existing graph-based key management approaches namely *VersaKey* [10] extended as a *DAG* similar to [12]. These approaches bind key material to nodes related to each other representing the *DAG*. While the source nodes (represented by the *CKs* with the *Key Graph*) constitute the client rights, the terminal vertices represent the most common access rights. Despite current approaches, the *EKs* are not only used to offer scalability but also to provide group-based access on the data.

A second adaption on *VersaKey*, besides the usage of a *DAG* with semantic *EKs* instead of a tree, includes the persistence of the updates applicable on the *DAG*. These updates, denoted as *Key Trails*, are not only broadcasted to the clients once but persisted on the cloud for on-demand updates of the keys. 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 *DAG* results in an update of all reachable group keys originating from the adjusted access right namely one node within the *DAG*. This adaption includes the generation of new key material for all those nodes. The resulting *Key Trails*, consisting out of the fresh key material encrypted with the related valid nodes, scale with this number of updated nodes since each *Key Trail* relies on an edge incident to the updated nodes. Instead of updating all shared keys, we only adjust the summarizing groups. By introducing *Virtual Nodes* for functional combination of groups, the scaling regarding the evolving key management is constant to the classic *VersaKey*-approach while the *Key Trails* scale with the number of updated nodes instead of the number of incident edges.

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.

2. Related Work

Related approaches in this area cover the storage of encrypted data on untrusted components like cloud-infrastructures as well as the scalable handling of keys with the help of *Key Graphs*.

Cloud security became a major issue within the last years.

Sato [7] 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 [1] 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 [13] 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.

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* [3] 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 ongoing with an hierarchical data structure as well. Our approach furthermore focuses on the versioning of the keys and the data and utilizes the distributed environment.

Multiple approaches exist to map key management to graphs. Waldvogel [10] proposes the arrangement of client-bound keys to an overall encryption key within a tree-structure called *VersaKey*. We base on this approach including a model for updating the nodes within our approach.

These approaches are extended to offer an even more efficient key handling by Wong [11] where the key graph propagates any changes via UDP/IP multicast. Forward error correction reduces the messages for efficient and reliable key updates.

Hassen [4] 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 [9, 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 [14] stays out of focus since we use the hierarchy within the *DAG* as semantic representation for organizational issues.

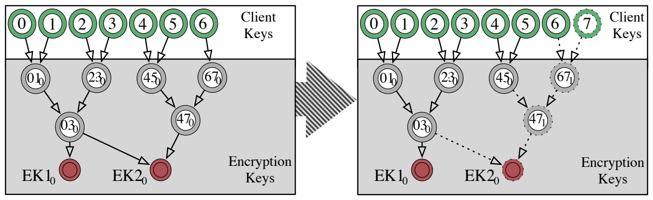


Figure 1: Classic Key Graph

3. Graph-based Key Management for Cloud Storage

Figure 1 shows a *DAG* constructed similar to classic stream-bound approaches [9, 12, 14].

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 out of its own *CK* and the descendants whereas one global *DAG* manages join-/leave-operations of nodes as well as insert/remove-operations related to edges in a centralized manner. If a client, represented by a node, joins or leaves the set of authorized clients, only parts of the keys stored within each client must be adapted. These parts include the descendants of the nodes where the modification on the *DAG* occurred.

Consider the graph in Fig. 1 that shows the insertion of the client 7. As a consequence of the insertion, the nodes 67, 47, $EK1$ and $EK2$ have to generate new 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 name, the version of the updated nodes increases.

Based upon 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 *DAG*, are called *Key Trails*. The updates are propagated in a secure manner based upon the encryption of 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 new node 47 is encrypted with the new material of node 67.

3.1 Theoretical Foundations of the *Key Graph*

We define the underlying *DAG* as an ordered pair $G = \langle V, E\rangle$ where V denotes the nodes in the graph while $E \subseteq V \times V$ represents the set of edges. Additionally, we assign a direction to each edge, i.e. each edge has a source node and a target node. A directed edge leading from source v to target w is therefore denoted as an ordered 2-sequence of a source node and a target node as in $\langle v, w\rangle$. For a node v, we define the indegree $i(v) := \left|\left\{\langle u, v\rangle \in E \mid u \in V\right\}\right|$ as the number of edges "incoming" at v and the outdegree $o(v) := \left|\left\{\langle v, u\rangle \in E \mid u \in V\right\}\right|$ as the number of edges "outgoing" from v. We denote by $C := \left\{v \in V \mid i(v) = 0\right\}$ the set of *CKs*, which are the root nodes. The *EKs* are the set $N := \left\{v \in V \mid i(v) > 0 \ \wedge \ o(v) \geq 0\right\}$ containing all nodes that are not *CKs*. The "terminal vertices" are just the set $K := \left\{v \in N \mid o(v) = 0\right\}$. A (directed) path $P \subset G$ is a non-empty subgraph $\langle V', E'\rangle$ of the graph G such that $V' \subseteq V$, $E' \subseteq E$ that has the form $V' = \left\{v_0, v_1, \ldots, v_k\right\}$ and $E' = \left\{\langle v_0, v_1\rangle, \langle v_1, v_2\rangle, \ldots, \langle v_{k-1}, v_k\rangle\right\}$ where all v_i with $0 \leq i \leq k$ are distinct. We will say that $P \subset G$ to denote that P is actually a subgraph of G. For the sake of simplicity, we denote paths in a short form as an indexed sequence of nodes like $\langle v_0, v_1, \ldots, v_k\rangle$ and say that P "leads from v_0 to v_k" (for any $k > 1$). In this case, we call v_k a *descendant* of v_0. We furthermore assert the following:

1. For each node $v \in N$ there must exist at least one *CK* $w \in C$ such that a path leads from w to v:
   $\forall v \in N : \exists w \in C \ \exists P = \langle w, \ldots, v\rangle$.

Informally, the assertion means that each "terminal vertex" is the end point of at least one path starting at a *CK* and there are *EKs* which could be left out without breaking such a connection. Note that nonetheless, *CKs* are allowed to be isolated and thus the resulting *DAG* is not guaranteed to be connected. It is also possible that the *DAG* does actually neither contain any *EKs* nor "terminal vertices".

Two types of modification operations are supported by our infrastructure:

1. Modifications of the node set to derive the updated set V_{i+1} from the prior set V_i:
   (a) A new node $x_{new} \notin V_i$ joins the *DAG*: $V_{i+1} := V_i \cup \left\{x_{new}\right\}$
   (b) An existing node $x_{old} \in V_i$ leaves the *DAG*: $V_{i+1} := V_i \backslash \left\{x_{old}\right\}$
2. Modifications of the set of edges to derive the updated set E_{i+1} from the prior set E_i:
   (a) A new edge $\langle u, v\rangle \notin E_i$ between two existing nodes $\left\{u, v\right\} \subseteq V_i$ is inserted: $E_{i+1} := E_i \cup \left\{\langle u, v\rangle\right\}$
   (b) An existing edge $\langle u, v\rangle \in E_i$ between two existing nodes $\left\{u, v\right\} \subseteq V_i$ is removed: $E_{i+1} := E_i \backslash \left\{\langle u, v\rangle\right\}$

Corresponding to the indices used with V and E, we will denote by G_i the prior *DAG* $\langle V_i, E_i\rangle$ and by G_{i+1} the posterior *DAG* $\langle V_{i+1}, E_{i+1}\rangle$ which is obtained as a result of one or more modification operations.

Consider the typical case when a node $x_{new} \notin V_i$ joins the graph G_i. If x_{new} is supposed to get any access rights, new edges have to be inserted into G_i. Be edge $e_{new} = \langle x_{new}, y\rangle \notin V_i$ with $y \in V_i$ the edge whose insertion is triggered by the join of x_{new}. This means, e represents the initial access rights of x_{new} in G_{i+1}. After the node has joined the graph and edge e was inserted, all descendants $desc(x_{new})$ of x_{new} in G_{i+1} must increase their version index and update their key material. We call these two consecutive operations an *update*. The set of nodes which are required to be updated after a modification has been performed is denoted by V^*_{i+1}. Actually, V^*_{i+1} is the set $desc(x_{new})$ in G_{i+1}. (Since we inserted only edge e, this set is in fact narrowed down to the descendants of y in G_{i+1}.)

Now consider the case of a leave: when a node x_{old} leaves the *DAG*, the update has performed on set of those nodes in G_{i+1} which are descendants of x_{old} in G_i, i.e. $V^*_{i+1} = desc_i(x_{old}) \cap V_{i+1}$.

Note that regardless if a node x joins or leaves, the set $V^*_{i+1} \backslash \{x\}$ of updated nodes in the result graph G_{i+1} is identical for both operation types.

If an edge $\langle v, w \rangle$ is inserted connecting two nodes already present in G_i, all descendants $desc(w)$ of the target node w in G_{i+1} have to be updated and exactly the same nodes have also to be updated after the removal of an edge. If an edge $\langle v, w \rangle$ is removed from G_i, all nodes in G_{i+1} which are descendants of w in G_i have to be updated.

Each type of modification causes the generation of *Key Trails* for all edges in $E^*_{i+1} = (V_{i+1} \times V^*_{i+1}) \cap E_{i+1}$ whereas one *Key Trail* represents one edge. Note that it is sufficient to consider only the edges for which the target is in V^*_{i+1} since there is by definition no edge that leads from a source in V^*_{i+1} to a target in $V_{i+1} \backslash V^*_{i+1}$.

It is important to note that the number of *Key Trails* is identical to the number of not-updated edges in G_{i+1} as well.

Therefore, the computation and storage of the *Key Trails* generates an overhead in the size of the number of edges incident to the nodes which are updated in G_{i+1} in the course of any modification.

To scale the number of *Key Trails*, we furthermore introduce *Virtual Nodes* within our approach described in Sec. 3.2.1.

3.2 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, four adaptions must be made to apply *VersaKey* on data storage:

- The stored data to be encrypted must be hierarchical organized. Due to the inheritance of access rights within the *DAG*, any data-structure adoring this inheritance (e.g. a file system or XML) provides the ability to encrypt different levels within the data with related access rights derived from the *DAG*. The *EKs* in the *DAG* offer not only scalable updates but also group-based access rights.
- 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 adore the changes within the key management ongoing within single join/leave operations. This awareness is achieved by versioning the data to be encrypted.

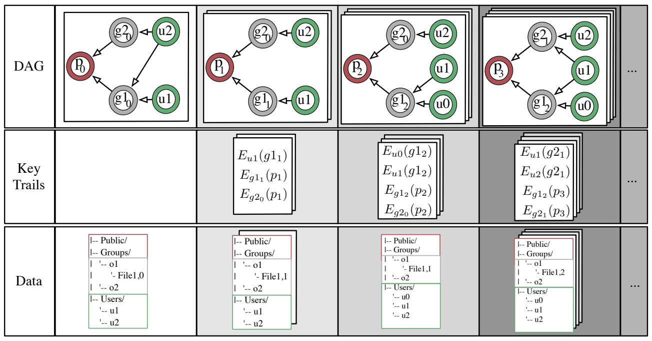


Figure 2: Evolving Data, *Key Trails* and *DAG*

- The *DAG* is versioned equivalent to the versioning of the data. Within the hierarchical structure of the stored data, each modification of an element is encrypted within a given node. Since the key material changes regarding different versions of the same node, all former keys from the *DAG* must be available to ensure access to all versions of the data.
- The updates on the *DAG* occur over persisted *Key Trails*. Since we rely on a versioning of the *DAG*, we use the *Key Trails* not only for on-the-fly adapting of the *DAG* but also as format for deltas between two versions on the *DAG*. Furthermore, the encrypted nature of the *Key Trails* is utilized to store updated key material in the cloud as explained in Sec. 3.3.

The four points defined above result in a versioning of the *DAG* 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 decrypt-able with fitting versions of *DAG*-nodes while the current version of the *DAG*-nodes encrypts ongoing modifications. The binding of node-versions to data-versions obviates re-encrypting the data within key changes. Figure 2 shows an evolving *DAG*, the related *Key Trails* and an hierarchical data-structure.

The versions of the *DAG* relate to the *Key Trails* computed on updated nodes. The updated nodes base upon the points of insertion / removal of edges / nodes within the *DAG*. Regarding the example of Fig. 2, the leave of the *CK* $u2$ results in the adaption of the same nodes than the insertion of the new *CK* $u0$ increasing the version of the nodes $g1$ and p twice. The join of the existing *CK* $u1$ to the group $g2$ however results only in the adaption of the nodes $g2$ and p whereas $g1$ stays unmodified. Each node updated by the modification gets new key material and an increased version even though the old versions of the nodes stay accessible containing their original key material and the corresponding pointers to their environment. Fig. 3 shows the layout of a node within the *DAG* persisted for the entire lifetime of the client. The unique identification of a node is handled over the field *Fixed ID*. The *Version*-field is incremented each time a node updates its key material and possibly the pointer to its parents and children if the adjustment of the access-right influences this node directly namely within a removal or insertion of an edge. All pointers to the environment, namely the edges to all incident nodes, are represented by two lists (*Parent IDs* and *Children IDs*). All of this data represents the information necessary to identify the position and the relation to other nodes and are denoted as *Key Selector* while the *Key Data* contains the key material used for encryption / decryption operations on the data. The *Secret Material* is changed within every modification of the node ongoing with an increment on the *Version*-field.

All modifications on the *DAG* result in the generation of the *Key Trails* based upon the edges incident to the updated nodes as shown in Fig. 2. Based upon their purpose as retrievable deltas to replay any changes upon the *DAG*, the *Key Trails* are versioned as well. Independent from the modifications within the *DAG*, the data undergoes modifications as well resulting on own versions. The related data encrypted with those nodes must be aware of the current version of the *DAG* within each modification. For example, if the change on *DAG* where the *CK* $u2$ is excluded from the group $g1$ occurs before the modification of the file "File1", the encryption key used for encrypting the new version of "File1" is based upon the new version of the node $g1$. *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 Trail* $E_{g1_2}(p_3)$. Fig. 4 represents a *Key Trail*. Each *Key Trail* consists out of a plain readable part and an encrypted part. The plain readable part hosts the relevant ids for the nodes: The *BaseID* and the *Base Version* identify the node and its version with which the *Key Trail* was encrypted. The *KeyID* and the *Key Version* represent the updated node. In other words, the *BaseID* represents the source node of the directed edge while the *KeyID* represents the sink node of the same edge. Regarding our example of Fig. 4, within $E_{g1_2}(p_3)$, $g1$ is the *BaseID* in the version 2 while p represents the *KeyID* in the version 3. Besides the plain part needed for the identification of the *Key Trail*, the second part is encrypted and stores the *secret material* for the updated node.

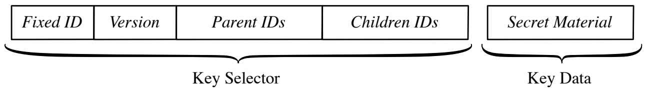


Figure 3: Layout of a node within the *Key Graph*

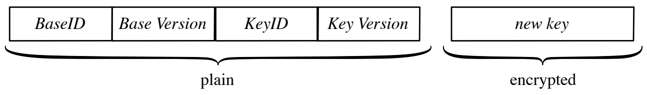


Figure 4: Layout of *Key Trail*

All nodes except the *Virtual Nodes* can therefore host encryption keys for the data. The keys on higher-levels in the *DAG* preferable encrypt higher-level elements within the hierarchical data to gain benefits from inherited access rights similar to *Cryptree* [3].

In our example shown in Fig. 2, the data "File1" is modified within several versions. For encryption, the current version of the associated key $g1$ is used. As a consequence, the new *CK* $u0$ has no access on former versions of "File1" than the one encrypted with version 2 of the node $g1$. The challenge to access former versions within new access rights is described in detail within Sec. 4.

3.2.1 *Virtual Nodes*

Since the *Key Trails* rely on the number of all incident nodes, nodes with a multiple parents produce multiple *Key Trails*. Since the number of associated modifications scale with the size of the group, the number of necessary *Key Trails* scales only with the number of incident nodes. To reduce the number of necessary *Key Trails*, we introduce *Virtual Nodes*. Wong [12] already showed how the outdegree of nodes within a *Key Graph* influences the performance of rekeying operations.

Based upon the usage of *EKs* to summarize a set of *CKs*, the scaling related to updates bases upon the number of parents of single *EKs*. Therefore, we introduce a threshold $t \geq i(v) := \left|\left\{\langle u,v\rangle \in E \mid u \in N\right\}\right|$ If $i(v) > t$, *Virtual Nodes* are inserted. *Virtual Nodes* are defined as follows: $T := \left\{s \in N\right\}$. Even though similar to *EKs*, the purpose of *Virtual Nodes* is not to encrypt any data. Instead, *Virtual Nodes* are utilized to act as proxy between the related nodes $u \in N \cap E, v \in N$ if $i(v) > t$: $E_{i+1} := E_i \backslash \left\{\langle u,v\rangle\right\}, E_{i+2} := E_{i+1} \cup \left\{\langle u,s\rangle, \langle s,v\rangle\right\}$

Consider the situation when a node u joins the graph and would now become a parent of node v but this would make $i(v)$ to exceed the threshold. A *Virtual Node* s is inserted as a parent of v and makes u and s adjacent to each other. Now, any path leading from u over v to some *EK* starts with the sequence u, s, v. Since the *Key Trails* rely on all adjacent edges to all updated nodes and the capping of the indegree of all nodes, the number of *Key Trails* scales by the introduction of the *Virtual Nodes* with the number of updated nodes and not with the number of adjacent edges to those nodes.

Fig. 5 represents the usage of *Virtual Nodes* where a new node 7 is introduced in the *DAG* with $t = 4$. The current nodes associated to the group key $g2$ are after the modification referring to the *Virtual Node* $v2$. Regarding our example shown in Fig. 5, the generated number of *Key Trails* ongoing within the insertion of node 8 decreases from 8 (without *Virtual Nodes*) to 6. With the help of the *Virtual Nodes*, the number of generated *Key Trails* scales with the number of the updated nodes and not with the number of incident edges. Even though *Virtual Nodes* have the same layout than other nodes within the *DAG*, their key material is due to their fanout-reducing purpose only used for encrypting related *Key Trails*.

3.3 Synergies between the Cloud and the Key Management

The aim of the appliance of our approach to cloud-based collaborative workflows is to utilize the high available but untrusted [5, 6] components within the cloud.

We therefore discuss the *DAG*, the data and the *Key Trails* over different components:

- The key management is divided into two disjunct types of *DAGs*:
  - A centralized instance upon a trusted component represents the overall *Key Graph*. This *DAG* consists out of all keys in all versions and triggers all changes upon the authorized client-set.
  - Besides the centralized instance containing all valid keys, each client holds its *CK* including all descendants in all accessible versions.
- The *Key Trails* are stored on the cloud. Due to the high availability and the scalability 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 within 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 clients are able to decrypt the *Key Trails* if still authorized to access the related node.

Figure 6 shows the appliance of our approach within a cloud-based infrastructure. A centralized *Key Manager* maintains the complete *DAG* representing all access rights for all data stored in the cloud. Any changes regarding the client set such as joins, leaves of clients and group adaptions occur within the *Key Manager*. Since the *Key Manager* adapts the changes within the client-set on-demand, the availability of the *Key Manager* can be adapted. After each update of the containing *DAG*, the *Key Manager* pushes the resulting *Key Trails* in the cloud.

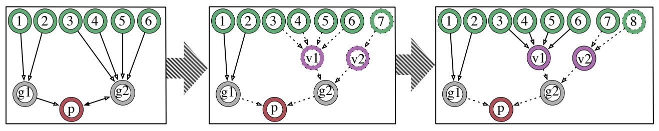


Figure 5: Inserting *Virtual Nodes*

Due to the availability and the scalability, the cloud infrastructure has the ability to store all *Key Trails* including all versions. Since the *Key Trails* are encrypted, the cloud service provider has no ability to decrypt them neither to access any data with the help of the *Key Trails*. Each client accessing a fixed version of the data retrieves the suitable version number and identifier for the corresponding node. If the client has access to this node but not the fitting version, the related *Key Trail* is transferred to the requesting client. The *Key Trails* are not versioned on the client since they only represent an encrypted, high available delta on the *DAG* persisted in the cloud for replaying any changes on the subgraphs within the clients. For checking the currentness of the keys, the last *Key Trails* to each node are persisted. The concrete workflow between server and client, for checking and updating the keys and transferring data, is described in Sec. 3.4.

The clients have the ability to encrypt any data with any node within their own subgraph. The data itself is stored on the cloud encrypted denoted as the locker in Fig. 6. Since the encryption takes place before transfer, the confidentiality of the data is provided while the data is in transfer or in storage. If a modification occurs on the data, the version number of the corresponding node is checked to be up-to-date. If the node is outdated, the related *Key Trails* are transferred to the client and decrypted if the client was not excluded within the overall *Key Graph*. Since each modification on the data results in one version, the version is mappable to exact one version of one node within the *DAG*. The versioning of the data furthermore results in an append-only storage on the cloud easily manageable within its scalability.

Even though the server deployed in the cloud contains no knowledge about the inlying data, it must be aware about the version stored or retrieved as pointed out in Sec. 4. Nevertheless the versioning itself is performed by the clients containing all permitted versions of the permitted data. Any data retrieved from the cloud is cached in the client to reduce network overhead. Within Fig. 6, the black arrows denote the transfer of any data in the cloud. As clearly visible, *Client 0* 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*.

Besides the different versions of the data, each client stores all related, accessible keys as well. A corresponding subgraph of the overall *Key Graph* stored on the *Key Manager* constitutes out of these specific keys. Updates on this set occur over the *Key Trails* transferred from the cloud if a client tries to access a node with an unknown key. In this case, the related *Key Trails* are transferred and, if the client has the corresponding current keys, encrypted as represented by the gray arrows in Fig. 6. Within this mechanism, new group keys denoted by *EKs* are transferred to the clients as well.

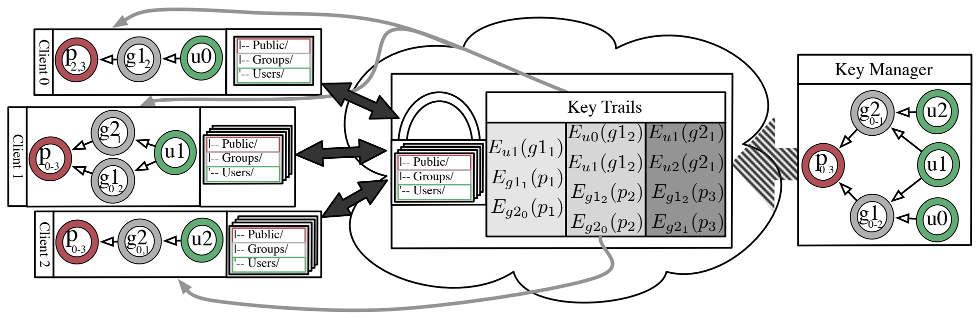


Figure 6: Overview of proposed architecture

3.4 Updating the Keys

The communication between clients and the server deployed in the cloud is represented by Fig. 7. Preceding any data access, the currentness of the corresponding node is checked related to the requested data. The consecutive requests of synchronizing the key material ongoing within read- / write-requests enforce a session-based protocol whereas we choose *SOAP* within our approach. The client starts a session with a server instance deployed on a *PaaS*-infrastructure. A challenge is provided to the requesting client by the server. The challenge is concatenated on the client with the hash of the last *Key Trail* for the node to be checked and returned to the server. The server performs the same operation to check if the last *Key Trail* of the client is the most recent one.

If the check fails, the server pushes the most actual *Key Trails* to the client. The client decrypts the *Key Trails* with the former keys and updates its local *Key Graph*. Afterwards, the request on the data is performed with the valid keys including encrypting the data, sending the data to the *PaaS*-instance and checking the integrity of the access with the help of a hash computed on the data transferred. If the request fails, this request is repeated. After a successful request, the session for this client is closed.

The usage of a challenge provides secure transmission despite additional security layers like e.g. *SSL*. Regarding new groups including new clients, the persistence and the versioning of the data mapped to the *DAG* and the *Key Trails* enables the insertion of new access rules based upon existing nodes. Only an initial graph consisting out of a single *CK* must be provided to the client. Further adaptions take place in an encrypted manner over the *Key Trails* provided by the cloud service.

The availability and scalability of the cloud is utilized without harming the confidentiality of the keys and, as a consequence, the data. The versioning of the data and the *DAG* makes re-encryption operations obsolete since each version of the data points to exact one version of a node. Even though existing *CKs* can be disabled within the *DAG*, new inserted *CKs* must gain access to data encrypted with former versions of reachable group-keys. This access, including the restriction to the current status of the data only, is described in Sec. 4.

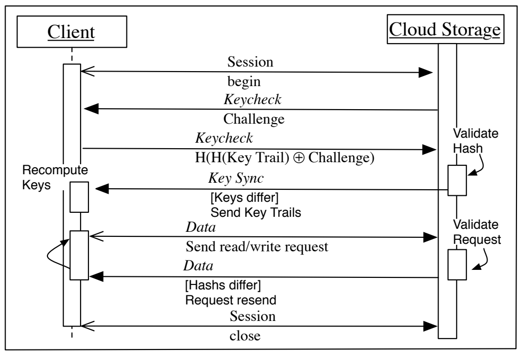


Figure 7: Sequence of write including checking of *Key Trails*

4. Versioned-based Access

Even though the modifications on the *DAG* 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. Access to former versions of the data are provided by sharing the related node including the relevant *Key Trails*. Since one encryption key within one version has the ability to map on multiple modifications on the data, consecutive modifications are guarded by only one key. The granularity for version-based access is thereby defined by the versioning of the related *EKs* within the *Key Graph*. Access to upcoming modifications are equivalent to *VersaKey*: A suitable *CK* is inserted and linked to the *EKs* representing the related rights. All modifications encrypted with any node reachable from the *CK* are afterwards accessible for this client. Although the access to preceding and succeeding versions are given, an adaption related to our approach must be made to provide access only to the current version of

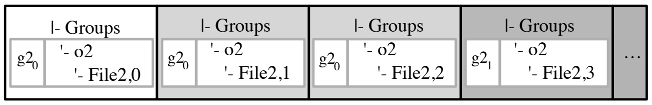


Figure 8: Encrypting different versions of the same data

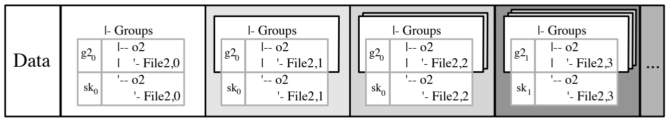


Figure 9: Shadow structure to provide access to current version

the data. Since one version of an encryption key has the ability to secure multiple versions of the data, an additional restriction of the encryption keys must be provided. If a client should gain access only to the current status without the ability to read former versions, simple sharing the related node violated this restriction.

Figure 8 describes this problem: Consecutive modifications on the “File2” are encrypted with one version of group key “g2”. Based upon the usage of one encryption key to protect multiple versions, the sharing of this key results in the access of all guarded versions. Giving access to a client joining group “o2” therefore automatically provides the ability to access former versions of the data as well. An access to only version 2 of the data is not supported by the classic *VersaKey* mapping. The access to current versions without exposing past versions encrypted with the same node therefore represents a challenge.

Due to the independence of the *Key Graph*-versioning and the versioning of the data, simple sharing an older version of the fitting encryption key without additional restriction not only eventually violates the access. Furthermore, it is not guaranteed to access the data in such way since the last modification on the requested data maybe encrypted by any former version of the *Key Graph*. We therefore propose two solutions to solve this problem: The first solution consists of redundant data and additional keys resulting of a shadow structure of the data and the keys as described in Sec. 4.1. The second approach utilizes the distributed architecture within our approach as described in Sec. 4.2.

4.1 Shadow Structure

The first proposed extension to our approach restricting access to only the recent version represents the insertion of shadow structures related to the *Key Graph* and the data. While the shadow structure of the *Key Graph* needs to be versioned similar to the *Key Graph* itself, the shadow structure of the data is not versioned. All modifying requests on the data thereby are first encrypted by the suitable node in the most recent version of the *Key Graph* and applied to the versioned data storage. Additionally, the modification is also encrypted by the same node of a shadow-*Key Graph* called *Shadow Key* and applied to an unversioned shadow structure called *Shadow Data*. The *Shadow Key* is provided as additional key material stored within each node of the *DAG* to access only the *Shadow Data*. The *Shadow Data* consists only out of the most recent version of the data. As a consequence, each modification results in a new version on the data encrypted with the normal key material of the linked node and the adaption of the *Shadow Data* encrypted with the key material of the *Shadow Key*. Since the *Shadow Data* contains no inlying versioning, each new client joining the *DAG* access the newest version of the *Key Graph* and the first version of the related *Shadow Key*. With the help of the applicable *Key Trails* on the *Shadow Key*, the new client has the ability to decrypt all data within the *Shadow Data*. Since the *Shadow Data* consists only out of the current version of the data, no access to former data versions is granted. If a client is excluded, the *Shadow Keys* are adapted similar to the normal *Key Graph* prohibiting any access to newly modified content within the *Shadow Data*.

Figure 9 represents an example of the shadow structures. While the data is encrypted and versioned with the help of the nodes within the *Key Graph*, the most recents modifications are also applied to a copy encrypted by the *Shadow Key*. Therefore, the access to “File2” is provided not only via the group key “g2” but also over the *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 the 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 group key “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*.

The *Shadow Data* consumes a constant factor of additional resources since the latest version of the data must be mirrored while the *Shadow Key* results in an additional field stored related to each node. By sharing the *Shadow Key* to accessing clients in a secure way (e.g. by encrypting the *Shadow Key* with the related *CK* as *Key Trail*), the client gains access to the most recent version without any possibility to read preceding versions of the data.

4.2 Token-based Extension

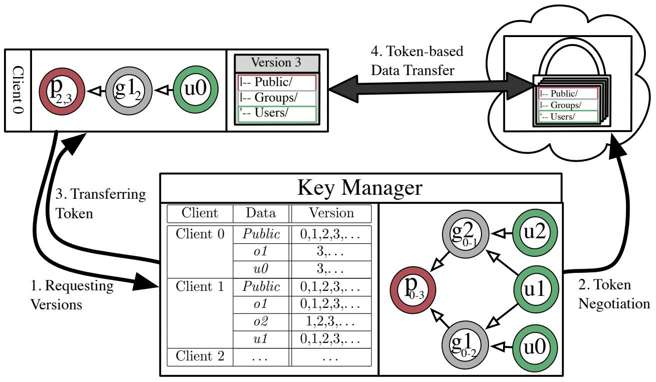
Another mechanism for restricting the access to defined former versions on the client is the deployment of an authorization layer within the distributed environment. The *Key Manager* contains a list of resources applied to the nodes. This mapping between the nodes and the data is enriched by the versions which are 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 besides the global *Key Graph* within the *Key Manager*. The binding between the versioning of the *DAG* and the data is represented over this structure.

Figure 10 shows an example: "Client 0" has access to version 3 of the data stored under "o1" whereas "Client 1" has the ability to access all versions stored under the same resource. Since the mapping between *DAG* and data takes place over a dedicated structure, each client contains all nodes within all versions accessible within the own *CK*. Related to the example of Fig. 10, "Client 0" contains all version of the accessible group key "g0" since the authorization for different versions takes not place over the *DAG*.

This approach scales with the number of rules bound to each version. For one c =*CK*, $\forall \{u, x\} \in V, p = (c, u_0)(u_0, x_0)(u_{m-1}, x_{m-1})(x_{m-1}, k)$ whereas $m \in \mathbb{N}_0 \wedge x_{i-1} = u_i, \forall i \in \{1, \ldots, m-1\}$, there can be a rule for all u, v mapped to one client. More concise, for each node key reachable within the path from the related *CK* within a client, there exists one rule mapping the versions of the data to the versions of the *DAG*. The rules are thereby not bound to the different versions of one node but to the node itself. Regarding the example of Fig. 10, "Client 0" contains three nodes accessible from the own *CK* based upon the path-definition from above. Therefore, at most three rules represent the version-mapping between "Client 0" and the *DAG*.

The proposed access-structure acts as a base for a token-based approach between the three disjunct components of our architecture namely the cloud, the disjunct clients and the centralized *Key Manager*. The workflow is denoted by Fig. 10 as well:

1. The client requests a version. The requested access is verified against the proposed authorization structure within the *Key Manager*.
2. 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*. Each token represents a single rule for a dedicated client.
3. After negotiation, the resulting token is sent to the client. The client has no ability to decrypt the content neither to modify the content without violating it.
4. 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.



Client	Data	Version
Client 0	*Public*	0,1,2,3,...
	o1	3,...
	u0	3,...
Client 1	*Public*	0,1,2,3,...
	o1	0,1,2,3,...
	o2	1,2,3,...
	u1	0,1,2,3,...
Client 2	...	...

Figure 10: Token-based extension

As already pointed out in Sec. 3.3, the storage must be aware of the versioning of the data although the different versions are encrypted. Due to the encryption of the modifications with the help of different keys, the awareness of the data by the *CSP* is constricted only to the versioning and not to the data itself. As a result, the *CSP* can identify different versions within the keys and the data but has neither the ability to gain access to the inlying data nor to get access to any keys.

The client has the ability to encrypt all data stored on the cloud with the help of the decrypted *Key Trails* resulting in a subgraph of the global *DAG*. Based upon the authorization structure mapping the versions of the *DAG* to the versions of the data, the access on the encrypted data in the cloud is guarded additionally to the encryption by the *DAG*. Since each request from the client must contain a valid token, negotiated between *Key Manager* and cloud-instance, the client has only limited access to the hierarchical data although it would be possible to utilize all nodes accessible within the *CK* within all versions if the client would have unauthorized access to all versions.

5. Implementation, Evaluation and Scaling

The proposed approach was implemented within the native XML database called *Treetank* [2] as data backend and a random generated *DAG* stored similar to the proposed architecture denoted in Fig. 3 and Fig. 4. The *DAG*, representing our *Key Graph*, is generated randomly with 250, 500 and 1000 nodes and consists of 10 levels. While the nodes are distributed equally on maximal 10 levels, 8 terminal vertices are included in this set of *EKs*. The outdegree of each node, except the terminal vertices, is at most 3, meaning that each node has at most 3 children. This restriction regarding the number of children does not restrict the indegree of a node. While one *DAG* is created with the usage of *Virtual Nodes*, the other *DAG* abdicates the insertion of *Virtual Nodes*. If an edge between two *Virtual Nodes* is inserted, the *Virtual Node* representing the sink is replaced by another node.

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 of 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.

The number of edges is based upon the number of nodes inserted. As a consequence the number of nodes affected within single update operations increases with the number of nodes inserted in the *DAG*.

Figure 11 represents the results of this benchmark whereas the y-axis denotes the number of *EKs* updated and the x-axis represents the *CKs* joining to the *DAG* mapped on single versions. Within the insertion of single *CKs*, only a constant number of related *EKs* is updated if no *Virtual Nodes* are inserted as shown in Fig. 11a. Since the edges between the *EKs* are already existing before the insertion of the *CKs*, the number of the nodes related to the insertion is independent from the number of previous inserted *CKs*. Nevertheless, depending on the numbers *EKs* in the *Key Graph*, different *EKs* are updated.

The same benchmark is performed including the generation of *Virtual Nodes* represented by Fig. 11b. Since the *Virtual Nodes* are inserted in the *DAG* while creation of the *Key Graph*, the *DAG* with the *Virtual Nodes* has a different layout than the *DAG* created without *Virtual Nodes*. This different architecture explains the lower number of nodes updated especially at the beginning. Even though the number of updated nodes is lower at the beginning, the scaling shows the price to be paid for the usage of the *Virtual Nodes*: Independent from the size of the *Key Graph*, the scaling is not linear any more due to the continuos insertion of *Virtual Nodes*. As a result, regarding huger graphs, the scaling results in lesser nodes updated since the usage of *Virtual Nodes* loosens the *DAG* whereas each *CK* has less descendants. The number of updated nodes within the usage of *Virtual Nodes* is therefore decreasing with the size of the *Key Graph*.

Figure 12 shows the scaling of updated nodes cumulated within all versions. The number of nodes without the usage of *Virtual Nodes* as shown in Fig. 12a 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.

By inserting *Virtual Nodes*, the scaling of updated nodes represented by Fig. 12b is not linear any more. Due to the increasing number of *Virtual Nodes* within each update, the number of nodes increases even by insertion of *CKs* only. The scaling improves, the more nodes the *DAG* contains since a loosely coupled structure enforces less updates ongoing within less insertions of *Virtual Nodes*.

The main argument for introducing *Virtual Nodes* is the generation of *Key Trails*. Figure 13 shows the number of *Key Trails* cumulated over all versions. 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. Based upon the fixed architecture of the generated *DAG* without *Virtual Nodes* as shown in Fig. 13a, the more nodes are present in the *DAG*, the more *Key Trails* are generated.

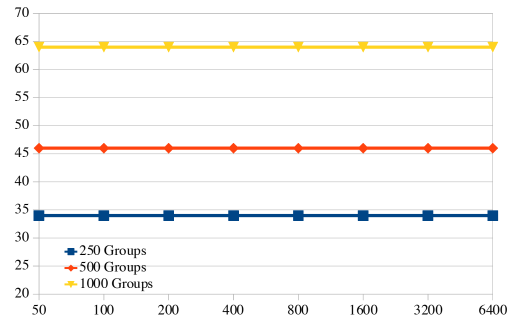
Since less nodes are updated within the introduction of *Virtual Nodes*, the number of *Key Trails* generated within each joining-operation of a *CK* is smaller represented by Fig. 13b. This sustains our results regarding the number of updated descendants of the *CKs* within Fig. 11 and Fig. 12.

The number of *Key Trails* is smaller by the usage of *Virtual Nodes*. Less nodes are affected if the *Key Graph* is already existing within insertions of single *CKs* and *Virtual Nodes*. Both aspects motivate the usage of *Virtual Nodes* based upon suitable selected thresholds even though the number of nodes in the *DAG* increases.

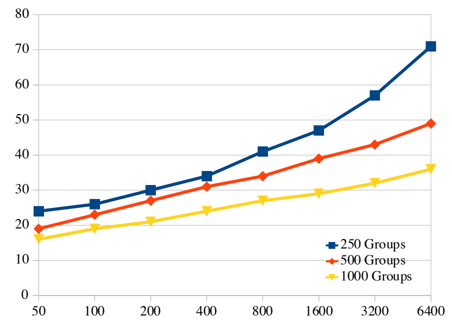
The evaluation of the *Virtual Nodes*-usage, our approach behaves like expected: Within any modification on the *DAG*, only a constant number of nodes are updated, namely the descendants of the modified node. Furthermore, the number of *Key Trails* playing a mandatory role within our approach, scales with the number of nodes and can be further improved within the usage of *Virtual Nodes*.

6. Conclusion and Future Work

Within our approach, we successfully bring stream-based *Key Graph*-approaches to the area of data storage. Our proposed distributed architecture utilizing versioning not only related to data storage but also related to the *Key Graph* enables changes within accessing clients without the need of re-encrypting any data. Modifications on the *Key Graph* update the descendants of the modified node. *Virtual Nodes* furthermore enable us to ensure scalability regarding the nodes adjacent to the updated descendants. The updates themselves are introduced as *Key Trails* representing the edges within the *Key Graph*. Since the *Key Trails* are encrypted and persisted, 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 *DAG* obsolete similar to the original *VersaKey*-approach. Since we update the *Key Graph* manually, we further evaluate function-based adaptions of updated notes making the manual key generation within each node obsolete and utilizing the difference

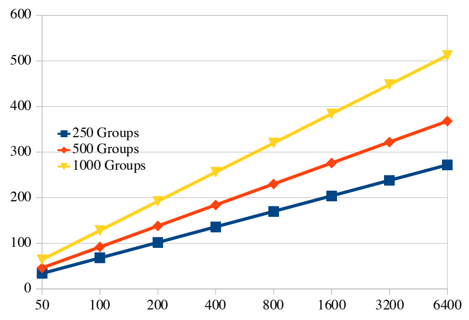


(a) Nodes updated in one version without *Virtual Nodes*

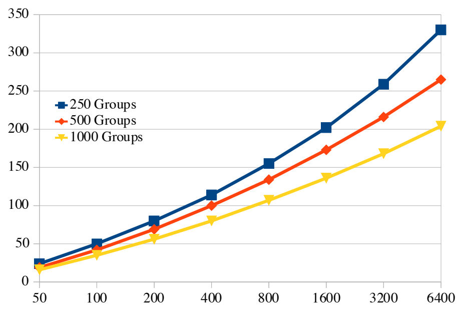


(b) Nodes updated in one version with *Virtual Nodes*

Figure 11: Nodes updated within modifications

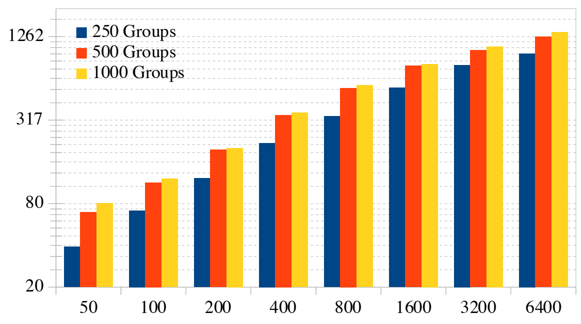


(a) Nodes updated in all versions without *Virtual Nodes*

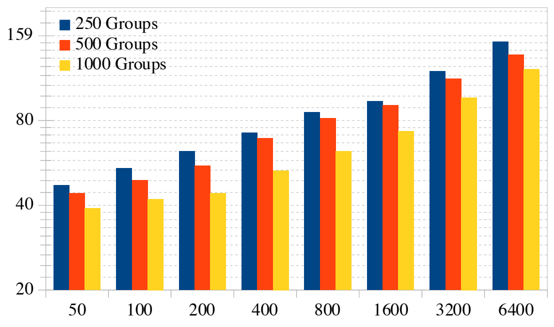


(b) Nodes updated in all versions with *Virtual Nodes*

Figure 12: Nodes updated within modifications



(a) Number of *Key Trails* per user without *Virtual Nodes*



(b) Number of *Key Trails* per user with *Virtual Nodes*

Figure 13: Number of *Key Trails* per user

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* [8] when equipping the key-*DAG* with a node-unique signature signing all version on the data. Based upon our focus for a secure cloud storage, such functionality benefits from the presented approach.

Acknowledgments

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

References

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

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

[3] 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*.

[4] 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.

[5] 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*, CLOUD '09.

[6] P. Mell and T. Grance. The nist definition of cloud computing. *National Institute of Standards and Technology*, 53(6), 2009.

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

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

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

[10] 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*, 17(9): 1614–1631, Sept. 1999.

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

[12] C. K. Wong, M. Gouda, and S. S. Lam. Secure group communication using key graphs. *IEEE/ACM Transaction on Networking*, 8(1), 2000.

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

[14] Q. Zhang, Y. Wang, and J. P. Jue. A key management scheme for hierarchical access control in group communication. *International Journal of Network Security*, 7(3):323–334, 2008.