University of Konstanz
Department of Computer and Information Science
Distributed Systems Group

Scientific Thesis
in Fulfillment of the Requirements for the Degree of
Masters of Science (M.Sc.)
at the Department of Computer and Information Science, University of Konstanz

Versatile Group Security For Tree-Based Storage

Author:
Patrick Lang

January 3, 2012

Referees:
Prof. Marcel Waldvogel
Prof. Marc H. Scholl
University of Konstanz
Department of Computer and Information Science
Distributed Systems Group
D–78457 Konstanz
Germany

Patrick Lang
Von-Raumer-Str. 17d
91550 Dinkelsbühl
Mat.-Nr. 01/735074
E-Mail: patrick.3.lang@uni-konstanz.de
Versatile Group Security For Tree-Based Storage
Master Thesis, University of Konstanz, 2012.

Abstract

The need to encrypt data in infrastructures containing thousands and thousands of entities having different access rights increases with the rapid growth of electronically stored sensitive data in all areas of business and life. A group-oriented communication system referred to as hierarchical access control is a main part of such an infrastructure. It forms a graph hierarchy in order to bind access rights to group nodes on different levels. Clients are able to join group nodes for getting permission in order to decrypt data which is encrypted with the corresponding group key. Each client maintains a key set contaning an own client key and all keys binded to groups the client is assigned to. Keys need to be renewed when a group affiliation changes, so that leaving clients are not able to decrypt group data anymore. In order not to re-encrypt data by using a renewed key, all keys are versioned and stored, so that the complete key graphs can be restored again. To sent renewed keys over an untrusted component (i.e. a server maintaining the key graph including the corresponding keys and a client holding a sub-key set), the new keys are encrypted with a key that is already known by the client.

This thesis describes an architecture for a revision-aware encryption framework on versioned data using a directed-acyclic key graph for hierarchical group management based on the *VersaKey*-approach. Previous approaches base on tree key graphs and provide only one top access right. Our approach enables multiple root instances connecting several key graph hierarchies. The key graph operates join and leave updates without re-encrypting storage data. The usage of a directed-acyclic key graph brings along more flexibility in key graph structure and group affiliations compared to binary tree hierarchies. On the other hand, this flexibility leads to an uncontrolled key graph expansion, key graph unbalance and non-linear scaling in key encryption. Proxy nodes are inserted which act as auxiliary nodes to avoid those drawbacks. Inserted proxy nodes are only part of the key management hierarchy and do not encrypt storage data.

The major aspect of this work is to make the versioned key management scalable even on updates affecting many node entities. Therefore, we examine and describe the directed-acyclic key graph management in detail and introduce proxy nodes ensuring scalability regarding the nodes adjacent to the updated ancestor nodes. We show that the directed-acyclic graph-based key management that implements proxy nodes, leads to a constant number of encryption steps for newly created key materials outgoing from the updated node entity. This proceeding results in a constant overhead related to the number of affected and updated node entities.

Acknowledgements

First of all, I would like to thank Prof. Dr. Marcel Waldvogel for not only being my supervisor but also giving me the opportunity to work on this issue. I also would like to thank Prof. Dr. Marc H. Scholl for being my second referee for this work.

I am truly grateful to Sebastian Graf for providing me with many friendly and valuable advices and ideas on many occasions and for the patient guidance.

A huge thanks goes to Lukas Lewandowski who was more than a good fellow on and off campus over the past six years. I would not want to miss the cooperation in many seminaries and study groups, as well as the countless good conversations, advices and coffee breaks.

Many thanks goes to the DiSy research group for the experienced time while I was a part of it.

My deepest gratitude goes to my mother. My studies would not have been possible without her powerful support under adverse circumstances.

Contents

List of Figures vi

List of Tables vii

1 Introduction 1

1.1 Motivation 1

1.2 Contribution 3

1.3 Outline 3

2 Preliminaries 7

2.1 VersaKey Framework 7

2.2 TreeTank 7

2.3 Related Work 8

3 Revision-Aware Encryption of Versioned Data 13

3.1 Architecture 13

3.2 Rekeying 16

3.3 Key Trails 17

3.4 Revisioning and Storage 19

4 Theoretical Foundations 23

4.1 Key Graph Theory 23

4.2 DAG Management 26

4.2.1 Insertion 27

4.2.2 Deletion 30
4.2.3 Summary 33

5 Proxy Nodes **35**
5.1 Motivation 35
5.2 Description 35
5.3 Key Trails on Proxy Nodes 37

6 Performance Evaluation **41**
6.1 Testing Preliminaries 41
6.2 Results and Discussion 44

7 Conclusion **51**
7.1 Final Summary 51
7.2 Future Outline 52

References **55**

List of Figures

3.1 Interaction of right management and storage component. 14
3.2 (a) linear graph (b) binary tree graph (c) directed-acyclic graph. 15
3.3 A timeline showing group affiliation actions of a client. 16
3.4 Affected graph nodes on a joining update. 18
3.5 *Key Trail* layout [8]. 19
3.6 Database excerpt showing distinctions between two revisions. 20

4.1 Hypergraph with four vertices and two hyperedges. 26
4.2 Leaf insertion example. (a) Original, (b) single node and (c) multiple nodes. 27
4.3 Intermediate insertion of (b) single and (c) multiple nodes. 29
4.4 Leaf deletion examples of (b) single and (c) multiple nodes. 31
4.5 Intermediate deletion examples of (b) single and (c) multiple nodes. . . . 32

5.1 Proxy nodes creation and key graph reorganization. 36
5.2 Encryption steps on proxy nodes. 38

6.1 A directed-acyclic key graph showing the different node types. 43
6.2 Number of *Key Trails* (a) without proxy nodes (b) with proxy nodes . . 45
6.3 Generation time of *Key Trails* (a) without proxy nodes (b) with proxy nodes. 46
6.4 Generation time of a single *Key Trail* (a) without proxy nodes (b) with proxy nodes. 47
6.5 Number of nodes affected (a) without proxy nodes (b) with proxy nodes. 48
6.6 Accumulated nodes affected (a) without proxy nodes (b) with proxy nodes. 49

List of Tables

2.1 Summary of Papers 10

3.1 Distinctions between *VersaKey*-approach and our approach. 15

„Computer security is nowadays no longer a question of intelligence but the time and discipline.“

Marc Ruef, Swiss computer security expert and author

Chapter 1

Introduction

This chapter gives insight into the motivation to work on this issue, the contribution to new scientific knowledge, a brief overview of the structure and the further progress of this thesis.

1.1 Motivation

Today we live in an electronic data controlled world. The amount and type of data that is gathered, stored and distributed is increasing daily. Whether private companies, non-governmental organizations (NGOs), government agencies or private individuals, all are storing and sharing sensitive and less sensitive data. People and organizations reveal and share their privacy data with others more than ever before. Almost daily, we are confronted with breaking news of spectacular electronic data theft. For private and research-intensive industries, data protection against industrial espionage and unauthorized access has become a fateful question into global competition. Often, data and information must be shared among different types of individuals having different access permissions organized in access groups, especially in systems like multimedia applications, video- or tele-conferencing, pay-per-view broadcasting of events, network gaming, collaborative work (e.g. Google-Docs), military or e-newspaper subscription systems,

Therefore, group-oriented communication systems were developed referring to as hierarchical access control (HAC). Such security systems require an application infrastructure that ensures multiple levels of access rights for system members. Members subscribe to get access to a particular resource. Only members with the corresponding access privileges are able to decrypt encrypted information and data streams they receive. The support of backward and forward secrecy[1] is an essential requirement for an hierarchical access control system. In backward secrecy a member is not able to access and decrypt data exchanged before it joined a group. In forward secrecy a member is not able to continuing follow and decrypt group communication after it left a group.

Key management schemes (KMS) ensure backward and forward secrecy for updating and multicasting encryption keys dynamically whenever a join or leave group update happens. Key management schemes are mainly distinguished into two categories: Centralized and Distributed schemes. A centralized scheme includes a key controller. A key controller generates, updates and manages the keys centrally. That makes the implementation simple but opens a single point of failure. A distributed scheme (sometimes also called contributory scheme) is more complex, but offers more data and key integrity and availability by spreading the key data over multiple servers, which in turn leads to an additional key data encryption, what makes it more sophisticated.

Usually, a tree-based structure is used to maintain keys into a centralized key management scheme in order to hold the computation and storage overhead low. Hence, a tree graph is mostly a broad and sometimes depth limited structure like a binary tree. Such a tree type includes only one root node and a lots of intermediate nodes having one direct accessor group and two direct successor groups. Every leaf node has exactly one accessor group and no successor group. An alternative is a less strictly limited key graph in the form of a directed-acyclic graph (DAG). A DAG includes multiple root nodes, and intermediate nodes having multiple accessor and successor nodes. A leaf node has multiple accessor nodes but no successor nodes, what means that a client is allowed to be a member of several groups. As previous research show, low limitation of a implemented key graph has negative impact on computation and storage overhead. The number of necessary key updates increases logarithmically with the size of groups [2] [3] [4].
To store and manage the bulk of data efficiently and protect it against unauthorized access is one of today's and future's biggest challenge. Data security is no one-way

street. Often enough, security mechanisms no longer meet today's quickly changing requirements. Thus, computer security systems constantly need to rethink and to call into question the state-of-the-art security techniques. These reasons have inspired us to contribute a part by doing an extension and progression of an encryption-aware hierarchical group management approach called *VersaKey*[5] and transform it into a revision-aware encryption approach on versioned structured data.

In this work, we present an hierarchical revision-aware group management framework using a *DAG*-structure. The intention is to show the architecture of this new approach and to examine the performance by using our implemented prototype. The prototype implementation provides a revision-aware directed-acyclic key graph framework to encrypt and decrypt structured versioned data. Since our approach bases on a *DAG*, we insert proxy nodes to hold the encryption keys generation overhead low.

1.2 Contribution

Much research work has been done in the area of stream-based encryption in hierarchical group management frameworks. This work extends some of those already researched patterns by applying revision-aware encryption on versioned structured data and by modeling group permissions in a directed-acyclic key graph. In a nutshell, this thesis contributes the following new ideas and research work:

- Join and leave operations without rekeying of the complete directed-acyclic key graph and without re-encryption of data.
- Revision-aware encryption of versioned data.
- Introduction of proxy nodes to decrease the overhead of rekeying.

1.3 Outline

The thesis outline is structured as follows:

Chapter 2 starts off with a general description of the tools and frameworks we have used in creating this work. The understanding helps to follow the upcoming discussions.

Finally, the chapter describes related research work and emphasizes the difference of our approach to already proposed approaches.

Chapter 3 gives a detailed insight into our approach. It presents the entire architecture, the rekeying and encryption procedure, revisioning and storage. The introduction is the basis of the upcoming chapters which examine some of these components in more detail.

Chapter 4 discusses the management of a directed-acyclic graph on *VersaKey*. It provides a general introduction into *DAG* theory. We outline eight cases of *DAG* management in detail comprising several insertion and deletion scenarios.

Chapter 5 enhances the *DAG* management by introducing proxy nodes aiming to improve the overall key graph performance regarding the generated key encryption steps on updates.

Chapter 6 evaluates the scalability between a *DAG* that implements proxy nodes as covered in chapter 4, and a DAG that does not implement proxy nodes as stated in chapter 5. Afterwards, we present and discuss the received results.

Chapter 7 finally, concludes this thesis by giving a final summary of the covered issue and the received evaluation results. Last but not least, we show ways how research can be pursued in future.

Chapter 2

Preliminaries

This chapter shortly introduces and describes some elementary tools and frameworks which constitute an important part of this thesis. The last subchapter discusses research work related to the thesis' issue.

2.1 VersaKey Framework

VersaKey is a versatile group key management framework providing perfect forward secrecy[6]. It allows to add and remove group members without re-encrypting the complete key tree hierarchy and provides an efficient key material distribution among change affected participants. Those join and leave operations have low complexity of *O(log N)*. This guarantees scalability even for a large number of groups. *VersaKey* supports some closely related key management schemes including centralized tree-based, centralized flat and distributed flat.

2.2 TreeTank

TreeTank [7] is an open source native embedded XML database. It is completely implemented in Java and in active development by the Distributed Systems Group at the University of Konstanz. Compared to other existing XML databases, *TreeTank* provides the feature of tree-structured data revisioning. It handles huge amounts of structured data and enables to switch between different data revisions. Our implemented prototype

uses *TreeTank* as the storage component whose versioned stored data is encrypted via the *VersaKey*-approach.

2.3 Related Work

There has been made an extensive number of publications in the fields of key management schemes and the hierarchy of keys. To cover and present all these publications and their sometimes slightly different approaches is far beyond the frame. Due to this and the fact that our approach addresses new ideas in several areas, we will consider and examine only proposals relating to centralized key management approaches treating scalable handling of keys in hierarchical key graphs, version-based key management, a *DAG* applied as key graph, and approaches proposing proxy nodes for better scalability. Furthermore, we will consider only *independent key* approaches[29] which usually use key graph structures. In an *independent key* approach, a client knows the key a data stream was encrypted with to decrypt it, whereas in *dependent key* approaches a client forms a de-encryption key of its own key and a public key to decrypt a data stream.

The original idea of a hierarchy of keys was proposed independently at same time by Walter et al.[4] and in a technical report[3]. The topic of structuring and updating group key management schemes was addressed by [2], [3] and [4]. A range of centralized key management schemes [3], [9], [10], [11], [12], [13], [14] and distributed key management schemes [15], [16], [17], [18], [19], [20], [21], [22] has been published to form a key graph and to update keys efficiently. A centralized key management scheme maintains key graphs on a central key server, distributes corresponding keys to clients on updates and maps the key management to graphs. A distributed key management scheme stores the keys in a distributed manner. All these schemes address the issue of structuring and dynamically updating changes. They treat the access control in a single multicast session.

Some of those approaches lack of flexibility in data access right management. They only provide the same access right for the same data streams for clients sharing the same group affiliation. In [23], [24] and [25], early hierarchical key management schemes have been proposed where clients are able to have access rights for different data streams. A key server manages the sub-key graphs containing the clients having access to a corresponding data stream.

Wang et al.[24] proposed an approach in where clients belonging to a higher-level group get more access rights. The higher-level group clients have access to all data streams sent to lower-level groups in their corresponding sub-key graph.

These approaches, however, may lead to key management inefficiency because of the maintenance of all sub-key graphs for clients having several access rights. The idea to use DAGs arose from this inefficiency.

Zhang et al.[26] proposed a hierarchical access control (HAC) scheme based on a unified DAG to overcome those drawbacks. The scheme considers the relations of clients and data streams, whereat equivalent data streams are encrypted by the same encryption key based on the data stream relation. It also enables to apply the periodic batch rekeying technique proposed in [14] in order to further improve key management efficiency.

Wong et al.[3] present an approach by using three rekeying strategies for joining and leaving a key graph. Each of these strategies is applied to a special class of key graph. They distinguish a star key graph, a tree key graph and a complete key graph. Each strategy is scalable to large groups with frequent join and leave operations. Similar to our approach, they apply a DAG as key graph and established that the optimal outdegree is four. But unlike ours, they do not use proxy nodes but simply limit the number of maximal outgoing edges and disallow to add further edges to a node once the maximum has been reached.

Sun et al.[23] proposed a multi-group management scheme with different access privileges by integrating a key graph maintaining the key materials for all key graph clients. Similar to our approach, the new generated key materials are versioned. The approach supports centralized and distributed environments and reduces the usage of communication, computation and storage resources, and scales with the number of data streams.

Waldvogel et al.[5] propose a tree-based multicast key management framework called *VersaKey*. *VersaKey* arranges client-bound keys to an overall encryption key. It grands scalability even for a large number of groups. Similar to our approach, *VersaKey* addresses the versioning of keys.

Our approach extends the previous presented key management approaches by mapping access rights of storage to a key graph. Those level-based access rights grant access to encrypted data.

Grolimund et al.[27] presented a folder tree structure called *Cryptree*. *Cryptree* stores

data hierarchical with access rights on subtrees mapped to groups and grants access to data without exposing the data of accessors. The approach includes a similar component to our approach since they focus on hierarchical group permissions ongoing with a hierarchical data structure. We extend this by implementing the versioning of keys.

Wong et al.[28] propose a tree-based approach called *KeyStone*. *KeyStone* propagates changes via UDP/IP multicast. It reduces the actual message loss for efficient and reliable updates of keys by using forward error correction.

Finally, Hassan et al.[29] propose a content access control approach by extending the usual group affiliation change and level change by intra-level change. The approach applies the changes on the rekeying process to the key-tree graph. The extension enables to change the node affiliation to a group.

We shortly summarize and compare the discussed papers in table 2.1.

Paper	Graph	Version-based	Proxy Nodes	Layout*	Scheme**
Wong[3]	Tree, *DAG*	No	No (but limits outgoing edges)	H	C
Waldvogel[5]	Tree	Yes	No	H, F	C, D
Sun[23]	Tree	Yes	No	H	C, D
Zhang[26]	*DAG*	No	No	H	C
Grolimund[27]	Tree	No	No	H	C, D
Wong[28]	Tree	No	No	H	C
Hassan[29]	Tree	No	No	H	C

Table 2.1: Summary of Papers

* H = Hierarchical, F = Flat

** C = Centralized, D = Distributed

We include the idea from [3] to limit the outgoing edges for better key graph performance. This approach does not allow to add further outgoing edges to a node once a maximum threshold is reached. We extend this idea by introducing proxy nodes to add an unlimited number of outgoing edges without losing performance regarding key graph scalability. We further extend the idea of key versioning from [5] and [23] by storing update affected keys within a key graph, and make it possible to restore each key graph version including its old keys. We also refine and alienate the idea of access right mapping to folders as proposed in [27], to map access rights to versioned storage.

Chapter 3

Revision-Aware Encryption of Versioned Data

In the following, we present an overview of our revision-aware encryption key graph on versioned data based on a directed-acyclic graph structure. The chapter describes the architecture and the functional principle related to the *VersaKey*-approach. This chapter constitutes an overview, whereas the upcoming chapters will get deeper into the details.

3.1 Architecture

This thesis pursues an approach consisting out of two main components. The first component represents the centralized hierarchical group right management in terms of a key graph. The hierarchical access control mechanism ensures the forward and backward secrecy and is inspired by the *VersaKey*-approach where disjunct clients share data based upon hierarchical organized access rights. The second component is any kind of storage whose data packages are encrypted with encryption keys of corresponding right groups of the first component. The encryption function uses a symmetric-key algorithm. This kind of approach is called the group model communication architecture aiming to provide confidentiality, authenticity and integrity of messages between multiple group members. This thesis bases on the author's master project prototype implementation where *TreeTank* was used as the storage component. *TreeTank* enables the revision-aware encryption of versioned data by providing the feature of storing revisioned data.

In this work, we focus on the first component only. The second component was shortly introduced to complement the architecture and the functional principle. Technically, any other storage system is able to being applied as well. Figure 3.1 depicts the interaction of the group right management component and the versioned storage component.

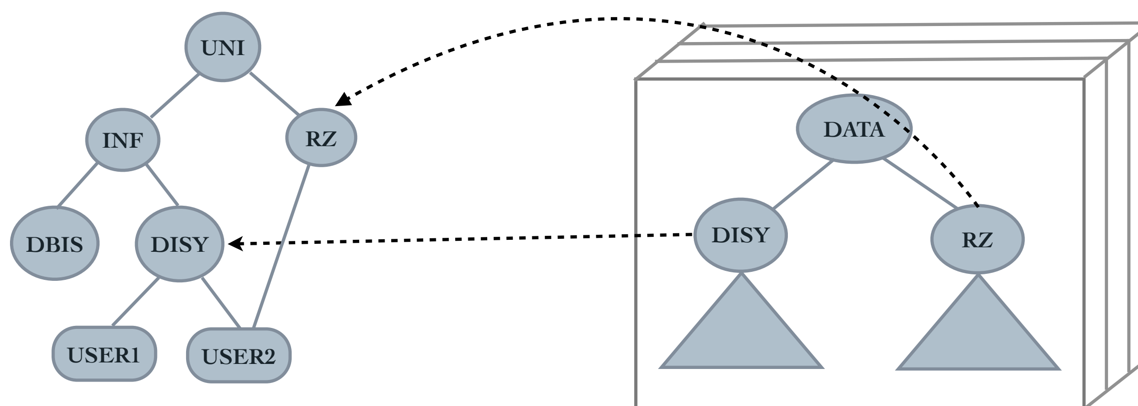

Figure 3.1: Interaction of right management and storage component.

The key graph represents the hierarchy related to the access rights. It bases on a directed-acyclic graph consisting out of a set of three different kinds of nodes: Root nodes, group nodes and client nodes. A directed-acyclic key graph has multiple root nodes, meaning the overall key graph enables to connect multiple subsets of hierarchy structures. A root node has ingoing nodes but no outgoing nodes. Each group node has one or more accessor nodes and zero to multiple successor nodes. The hierarchy has two types of keys: *Encryption Keys* (EKs) representing the group keys and *Client Keys* (CKs) representing the keys of the disjunct clients. Both kinds of keys are responsible to encrypt data. EKs are further responsible to offer balanced scalability. CKs however, are additionally used as starting point for the encryption of new key materials on updates. The following subsections explain the schematic in more detail.

A *DAG* does not contain a cycle. That means that there must not be a direct connection starting from a node over one to multiple other nodes back to the output node. In chapter 4, we go deeper into the characteristics of a *DAG*. A client node has one to multiple ingoing nodes but no outgoing nodes. Hence, a client is able to be a member of several groups, and thus it is able to own multiple different access rights.

Each group node, as well as a each client node owns a unique encryption key material. The shorter the path of a group node to one of the root nodes is, the higher is the access

privilege being a member of this group. Each client owns a key set that includes all keys of group nodes being on the path to the group's root node(s). A client knows its own sub-key graph including all access rights it is authorized for. A root node is known by all clients into the sub-graph. The key set of a client is formed as *keyset(client)* = $\{key_0, \ldots, key_n\}$. Each group node owns a client set containing all clients which know the key bind to the group node. The client set of a group is formed as *clientset(groupkey)* = $\{client_0, \ldots, client_n\}$.

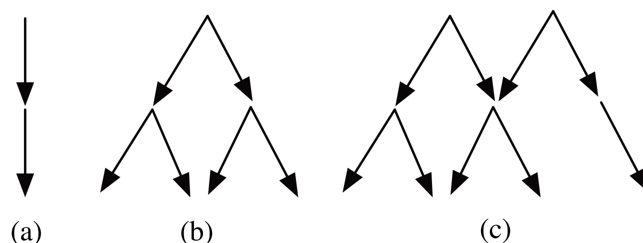

Figure 3.2: (a) linear graph (b) binary tree graph (c) directed-acyclic graph.

The introduced architecture bases on the *VersaKey*-approach but differs in some components. *VersaKey*'s key management bases on a binary tree, and thus it does not enable clients to have multiple group affiliations. As a result of this, proxy nodes do not play a role, since a binary tree naturally balances a key graph. Proxy nodes equalize the drawback coming along with the higher flexibility of a *DAG* regarding multiple group affiliations. *VersaKey* versions key materials within the key graph, but does not store revisions to restore previous key graphs. Table 3.1 summarizes the distinctions of both approaches.

Approach	Graph	Version-based	Proxy Nodes	Revisioning
VersaKey	Binary-Tree	Yes	No (but naturally balanced)	No
Thesis' approach	*DAG*	Yes	Yes	Yes

Table 3.1: Distinctions between *VersaKey*-approach and our approach.

3.2 Rekeying

Rekeying ensures the forward and backward secrecy. Both mechanisms guarantee that a client is not able to access and decrypt data which were exchanged before the client joined a group, as well as a client is not able to continuing follow and decrypt a group communication after it left a group. A central key controller component manages the keys and conducts the rekeying process on updates. The management of keys comprises the keys of the latest DAG revision as well as the keys of past revisions.

A key controller is responsible for all actions around keys comprising storage, retrieval, updating, creation and the distribution of keys. It shares an individual key material with each client node and each group node. Each client knows its own client key and all group keys of its known sub-key graph. Any modification of the key graph leads to an adaption of all group keys within a sub-key graph.

A new client subscribes to a group for getting access permission to a particular data resource, or a client leaves a group voluntarily or forcefully. Key data is renewed on a group change, no matter whether a joining or a leaving change happens. The key controller distributes a new client key in a unicast fashion. This happens only on joins and concerns only the joining client. Group key distribution however, is multicasted to all group members.

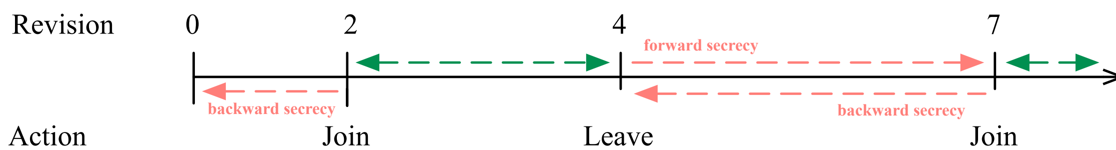

Figure 3.3: A timeline showing group affiliation actions of a client.

Join

The group key and all other keys of groups lying on the direct path to the root node(s) change on a client join. This is necessary to fulfill backward secrecy. A joining client has no more access to previous encrypted communication. Figure 3.3 shows a timeline of group affiliation actions of a client. A client joins a group at revision two. Backward secrecy ensures that the client has no more access to group communication between

revision zero and revision two. A client join creates a new version for each affected group node within the sub-key graph. Figure 3.4 depicts such a sub-key graph a client knows. Client *5* joins group *G*. All group nodes lying on the paths to the both root nodes, including *A*, *E*, *F* and *G*, belong to the client's known sub-key graph and change. For keeping the feature of restoring all previous key graph revisions, a copy of the changed nodes is set up and adapted in the database before the key controller creates a new key material for each of the affected nodes. After all affected nodes were set up and adapted in the database, the key controller encrypts and transmits the new key materials to the corresponding clients within the entire key graph including the new client. The encryption of key materials is described in chapter 3.3.

Leave

A group node key changes when a client leaves a group. The same holds to all keys of the group nodes lying on the direct path to the root node(s), originated at the group node of the former client affiliation to meat the requirement of forward secrecy. Considering figure 3.3, a leaving client has no more access to upcoming communication after its leaving in revision four. All the keys a client received during the time of its group affiliation expire. It is not enough just to withdraw the corresponding keys a client owns. It may be that a client has extracted and stored the keys in some way before its leaving. For that reason, the key controller creates a new version and a new key material for each affected node. The new version creation and the new key material transmission work the same way as previously described for a join.

3.3 Key Trails

By implementing the approach into a distributed architecture comprising highly available but untrusted components (i.e. a cloud infrastructure), there is a need to transmit the new created secret materials triggered by an update, securely to the corresponding clients. Hence, we introduce the term *Key Trail*. The actual idea of *Key Trails* was proposed by [5].

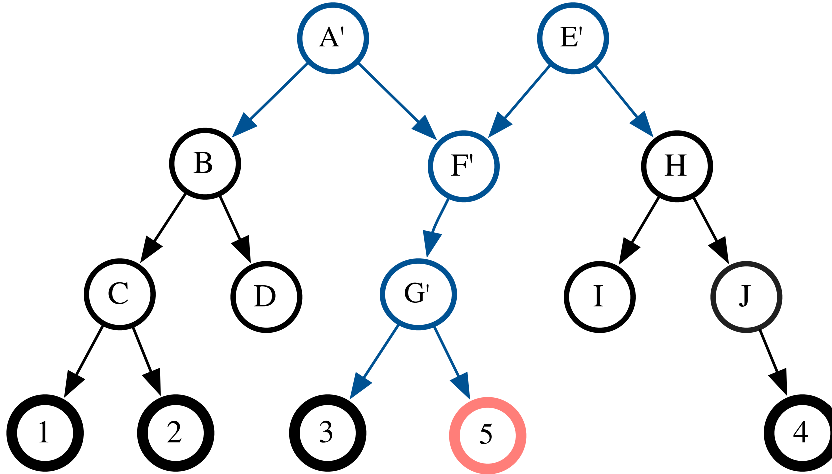

Figure 3.4: Affected graph nodes on a joining update.

Key Trails are encrypted key materials. More precisely, a single *Key Trail* is an encrypted key material of an affected node. The key controller creates new key materials for all affected nodes on a key graph update. Each child node of an affected group node encrypts the group's new assigned key material. Each new group key encryption by a child node generates a single *Key Trail*. *Key Trails* generation is an iterative bottom-up process originated at the updated group node and goes until all root nodes have been reached. The total number of generated *Key Trails* is the sum of all child node edges of all affected nodes. Referring to figure 3.4, the join of client *5* to group G affects four nodes (blue-colored nodes) having seven child edges altogether, resulting in seven *Key Trail* generations. A *Key Trail* layout contains out of a unique base identifier, a key identifier, a version and an encrypted new key material. The base identifier makes the *Key Trail* unique and identifiable. The key identifier and the version provide information about the key material a new node key material was encrypted with. For the sake of presentability and simplicity, we denote a *Key Trail* as $E_x(y')$, where y' is a new key material of node y which is encrypted with the key material of x. x is a child node of y', but not necessarily of y.

Figure 3.4 visualizes the *Key Trail* procedure. It depicts a *DAG* where a new client *5* joins group G. The update creates a new node version including new key materials for nodes A, E, F and G (an inverted comma shows a new node version). This example generates the following $Key\ Trails$:

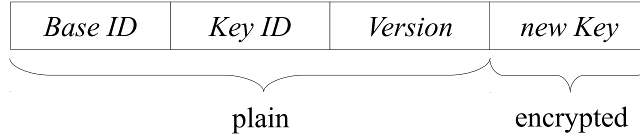

Figure 3.5: *Key Trail* layout [8].

$E_5(G')$, $E_3(G')$, $E_{G'}(F')$, $E_{F'}(A')$, $E_{F'}(E')$, $E_B(A')$, $E_H(E')$

The clients 3 and 5 encrypt the new key material of G' with their own client key material. A client key material never changes until a client leaves a key graph. G' encrypts the new key material of F' by using its new key material. F' encrypts the new key material of A' and E' each. B encrypts A' and H encrypts E'.

The clients are able to receive the new key materials over an untrustworthy connection after *Key Trails* generation. If someone intercepts the connection, it is impossible to decrypt the key material without knowing the original encryption key that is knowing only by the respective recipient client. Each client is able to encrypt the *Key Trails* of the corresponding sub-key graph it is part of. For instance, client 3 knows the key materials of node A, E, F and G, and its own key material, summarized as $keyset(3) = \{A, E, F, G, 3\}$. Hence, the client has the authorization to receive the *Key Trails* $E_3(G')$, $E_{G'}(F')$, $E_{F'}(A')$ and $E_{F'}(E')$. Client 3 decryps the new key material of G' by using its own key material and puts it into its key set. Then subsequently, by using the decrypted new key material of G', client 3 is able to decrypt the new key material of F'. Eventually, the encrypted key material of F' decrypts the new key materials of A' and E'. Client 3's key set contains $\{A, E, F, G, 3, A', E', F', G'\}$ after all decryptions.

3.4 Revisioning and Storage

Each join or leave update creates a new revision of a directed-acyclic key graph. Since a *DAG* usually comprises several thousands of group nodes and client nodes, a complete storing of each *DAG* revision is unreasonable and leads to a fast and nonlinear growing

of the database. Hence, the key controller only stores the nodes affected by an update and their corresponding child nodes for a new revision. However, the key controller only creates new key material for directly affected nodes, not for their corresponding child nodes. Since an update does not affect the child nodes directly and since the child nodes are not part of the changing sub-key graph, the child nodes keep their old key materials. Those child nodes are nevertheless created since their parents change, and thus all child nodes need a new version referencing to the new parent node. This proceeding guarantees a proper and complete restoring of past, current or future revisions.

ID	Name	Version	Parents	Children	Secret Key
1	A	0	-	2;6	…
2	B	0	1	3;4	…
3	C	0	2	11;12	…
4	D	0	2	-	…
5	E	0	-	6;8	…
6	F	0	1;5	7	…
7	G	0	6	13	…
8	H	0	5	9;10	…
9	I	0	8	-	…
10	J	0	8	14	…
11	1	0	3	-	…
12	2	0	3	-	…
13	3	0	7	-	…
14	4	0	10	-	…
15	A	1	-	16;18	… (new)
16	B	1	15	**3;4**	…
17	E	1	-	18;20	… (new)
18	F	1	15;17	19	… (new)
19	G	1	18	21;22	… (new)
20	H	1	17	**9;10**	…
21	3	1	19	-	…
22	5	1	19	-	… (new)

Figure 3.6: Database excerpt showing distinctions between two revisions.

Figure 3.6 shows a database design containing the data of the key graph depicted in figure 3.4. We assume that the join of client 5 is the first update in *DAG* history. Revision zero forms the original key graph. Each database entry represents a single node and its data. An entry consists out of a unique identifier, a node name, a version, a parent list, a child list, and a secret key material. The parent and child lists hold the identifiers linking to the entries of their parents and children in each revision. Entries that have no parent node links are root nodes and entries that have no child node links

are client nodes.

While still placing the focus on figure 3.6, all entries from 15 to 22 change due to the join update as depicted in figure 3.4. These entries form the new and latest revision one. The nodes B and H are not directly affected by the update and they are not known by the joining client, but their parents change. Therefore, the key controller creates a new version of both nodes for the new revision. The links pointing to the child nodes of B and H do not change and keep the same. These both child nodes act as a bridge and they connect the current revision with not changing parts of a former revision. Through that the key controller only stores a subset of the new key graph revision instead of the entire key graph. As a consequence, a revision key graph consists out of distinct nodes from different revisions. This approach leads to a linear growth of the database relative to the update operations.

Chapter 4

Theoretical Foundations

While most previous approaches for group management hierarchy frameworks (including *VersaKey*) are based on tree structures like binary trees, this chapter covers the theoretical foundations of implementing a directed-acyclic key graph and eight scenarios of DAG update operations.

4.1 Key Graph Theory

General

A directed-acyclic graph (DAG) is an abstract representation of a set of nodes and arrows. The nodes are also called vertices and arrows are commonly named directed edges or arcs. A DAG is an ordered pair defined as $DAG = \langle V, E \rangle$ where V denotes the nodes in the key graph and E the directed edges having a source and target node each. A directed edge between a source node and a target node is denoted as $\langle v, w \rangle$, where v is a source node and w a target node. The arrowhead points to the target node. Such a two nodes interconnection is called ordered 2-sequence. Each node v has several incoming and outgoing edges. A node is denoted as target node on an incoming edge and a node is denoted as source node on an outgoing edge. For a node v, we denote an incoming edge by $incoming(v) := |\{\langle u, v \rangle \in E \mid u \in V\}|$ and an outgoing edge by $outgoing(v) := |\{\langle v, u \rangle \in E \mid u \in V\}|$. A key graph node set is composed of three subsets, namely a root node set R, an intermediate node set I and a leaf node set L. A root node $r \in R$ has no incoming edge and zero-to-many outgoing edges. We denote a

root node by $r := \{v \in V \mid incoming(v) = 0 \ \wedge \ outgoing(v) \geq 0\}$. An intermediate node $i \in I$ has one-to-many incoming and one-to-many outgoing edges. We denote an intermediate node by $i := \{v \in V \mid incoming(v) > 0 \ \wedge \ outgoing(v) > 0\}$. A leaf node $l \in L$ has one-to-many incoming edges and no outgoing edge, and constitutes an endpoint of a key graph. All leaf nodes within a key graph form a set of client keys. We denote a leaf node by $l := \{v \in V \mid incoming(v) > 0 \ \wedge \ outgoing(v) = 0\}$.
A DAG has at least one node that represents a root node. A DAG consists out of several hierarchies including their own roots. Those hierarchies are not necessarily interconnected. All connections among nodes within a key graph must be in an acyclic order without directed cycles. Meaning, no direct connection starting from node n over several other nodes back to node n exists.

The graph theory distinguishes three kinds of edges:

- $Forward - edge$: leads from a node v to an adjacent successor node w.
- $Backward - edge$: leads from node w to an adjacent ancestor node v.
- $Crossing - edge$: connects two independent nodes i.e., two nodes, none of which is an adjacent successor node of the other.

These three edges determine the validity of a DAG. A valid DAG must not have backward-edges. The constant DAG validation is an important task to avoid errors in the encryption process in an application that uses a DAG as key graph.

A non-empty sub-key graph SG with $\langle V', E' \rangle$ contains distinct nodes of a key graph G such that $SG \subset G$. We denote $V' = \{v_0, \ldots, v_n\}$ and $E' = \{(v_0, v_1), \ldots, (v_{n-1}, v_n)\}$, where $V' \subset V$ and $E' \subset E$.

Updates

Since our key graph enables insertions and deletions resulting in different key graph versions, we denote DAG_i as the preceding graph and DAG_{i+1} as the updated graph. An update change comprises the insertion or deletion of nodes and their corresponding edges. We denote a preceding node set by V_i and the updated node set by V_{i+1}. The

same holds to updates on edges. E_i is a preceding edge set and E_{i+1} an edge set after an update. An insertion of a new distinct node n that is not into node set V_i yet, is formally denoted by $V_{i+1} := V_i \cup \{n\}$. We denote a removal of a node n from a DAG_i by $V_{i+1} := V_i \setminus \{n\}$. Concomitant with the insertion or deletion, an update inserts or deletes the edges of an updated node, too. We denote a not yet existing connection $\langle v, w \rangle$ between two existing nodes in V_i by $E_{i+1} := E_i \cup \{\langle v, w \rangle\}$. We furthermore denote the removal of an existing connection by $E_{i+1} := E_i \setminus \{\langle v, w \rangle\}$.

The key controller inserts a new edge between the joining group and n_{new} when a new node n_{new} joins a key graph where $n_{new} \notin V_i$. All ancestor nodes of inserted node n_{new} are updated, where $n_{new} \in ancestors\{n_{new}\} \subset V_{i+1}$ is part of subgraph $V' = \{v_0, \ldots, v_n\}$ of DAG_{i+1}. We denote the node n_{new} by v_0 and the subgraph's root node by v_n . Hence, v_1 to v_{n-1} form the intermediate node set, if n is larger than zero and v_n is not the parent node of v_0.

When a node n_{old} leaves a key graph and creates DAG_{i+1}, the key controller updates all its former ancestor nodes $n_{new} \notin ancestors\{n_{old}\} \subset V_{i+1}$ within DAG_i.

By inserting a new edge $\langle v, w \rangle$ between two nodes which are already part of DAG_i, the controller updates all ancestor nodes of the target node $ancestor(w)$ within DAG_{i+1}. Exactly the same node path within DAG_{i+1} is updated when the controller removes an edge $\langle v, w \rangle$ between two existing nodes within DAG_i.

All update procedures trigger the generation of *Key Trails*. The key controller generates *Key Trails* for all edges of all affected nodes of the updated sub-key graph. The number of generated *Key Trails* equals the number of all target nodes of $V' = \{v_0, \ldots, v_n\}$.

Hypergraphs

A hypergraph connects an edge with any number of vertices. We formally describe a hypergraph by $H = \langle V, E \rangle$, where V is a set of vertices and E is a set of hyperedges. A hyperedge is a non-empty subset of X, whereas X is a non-empty subset of V. Figure 4.1 depicts an example hypergraph with $X = \{v_1, v_2, v_3\}$ and

$E = \{e_1, e_2\} = \{v_1, v_2\}, \{v_2, v_3, v_4\}$. The example shows a hyperedge e_2 connecting nodes v_1 and v_2 with each other, and a hyperedge e_1 connecting node v_2 with v_3 and v_4.

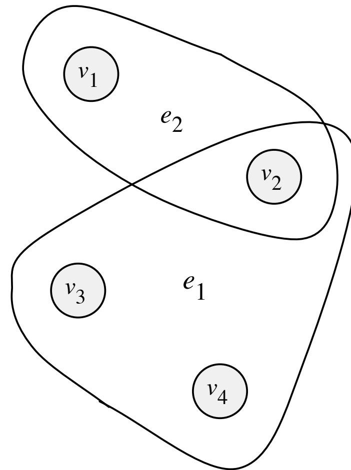

Figure 4.1: Hypergraph with four vertices and two hyperedges.

A hypergraph is an alternative for a *DAG* but it brings along much more complexity for tasks like hierarchical group management. In a conjunction with several versions on an edge, a hypergraph is an ideal tool to meet the complex requirements for such an implementation and eases the connection of multiple node versions. For instance, a client belonging to a key graph that has been updated several times, has a connection over one edge to all updated group versions including the corresponding versions of key materials instead of many edges. We have not pursued this approach in this work since we have focused on *DAG*s. We just mention the topic of hypergraphs for the sake of completeness.

4.2 DAG Management

This subchapter describes the management of a directed-acyclic key graph within the *VersaKey*-approach. We illustrate eight cases of inserting and deleting nodes and their impacts on the *DAG* and on its encryption hierarchy. Hereby, we distinguish between two operations on two key graph levels: Insertion and deletion operations on leaf level and intermediate level.

4.2.1 Insertion

Single Node (Leaf-Level)

An insertion of a single node at the lowest level, the so-called leaf level, constitutes the simplest case of a node insertion. The key controller extends the new sets of V' and E' by the new node and by its new edge to its new parent node. The number of new edges added to E' equals the number of a new node's new parent nodes if it has more than one parent node.

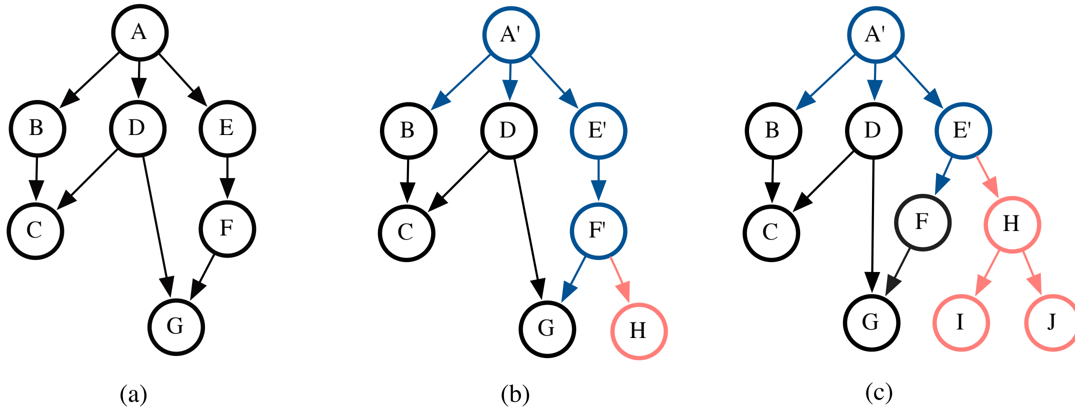

Figure 4.2: Leaf insertion example. (a) Original, (b) single node and (c) multiple nodes.

We formally denote a single leaf node insertion by $V_{i+1} := V_i \cup \{n\}$ and $E_{i+1} := E_i \cup \{(\langle v_0, w_0\rangle, \ldots, \langle v_k, w_k\rangle)\}$, where k is the number of new edges or parent nodes. The sub-key graph SG_{i+1}, the new added node is part of, includes all vertices and edges which are affected by the update. The subgraph contains all key graph parts needed for the *Key Trails* generation.

Figure 4.2 shows the original *DAG* (a) and the updated *DAG* (b) after inserting a single node H with an edge to F. The blue-colored elements (vertices and edges) show the parts affected by the insertion update. Those parts also form the sub-key graph a new node knows. The update affects all intermediate vertices lying on the paths to the root node(s). Moreover, all blue-colored edges are simultaneously encryption steps, where an updated node is encrypted by all its child nodes.

The updates illustrated in figures 4.2 (b) and (c) generate the following *Key Trails*:

(b): $E_H(F')$, $E_G(F')$, $E_{F'}(E')$, $E_{E'}(A')$, $E_D(A')$, $E_B(A')$

(c): $E_H(E')$, $E_F(E')$, $E_{E'}(A')$, $E_D(A')$, $E_B(A')$

Note that only update affected edges of existing nodes trigger the generation of *Key Trails*. The key controller does not generate *Key Trails* for outgoing edges of new nodes (i.e. edge H → I and H → J in 4.2 (c)).

Multiple Nodes (Leaf-Level)

The insertion of multiple nodes can be a sequence in a row or an entire sub-key graph. By inserting multiple nodes, there is technically no difference to a single node insertion. This is because the encryption process only considers the root node of the new sub-key graph, even if the sub-key graph contains multiple root nodes. Therefore, the node encryption behaves like on a single node insertion.

The number of new edges equals the number of parents to which the root node(s) of the new sub-key graph is/are added to, plus the edges connecting the nodes within the sub-key graph themselves. The total number of new edges is determined by $\mid \; new \; parent \; nodes \mid + \mid sub - key \; graph \; nodes \mid - 1$, if the new sub-key graph contains exactly one root node. If a new sub-key graph contains more than one root node, the number depends on the interconnections within the sub-key graph. Figure 4.2 (c) illustrates an example for such a case.

Formally, we denote multiple nodes insertion on leaf level by $V_{i+1} := V_i \cup \{(n_0, \ldots, n_{j-1})\}$ and $E_{i+1} := E_i \cup \{(\langle v_0, w_0 \rangle, \ldots, \langle v_{k-1}, q_{k-1} \rangle)\}$, where j is the number of new nodes and k the number of new edges.

Intermediate Node(s)

The insertion of a so-called intermediate node is slightly different of inserting a simple leaf node. A new node is defined as intermediate node n_{new} when the node includes at least one incoming and one outgoing edge after insertion. On the contrary of inserting

a leaf node, an intermediate node triggers a reorganization of a part of the key graph. Meaning, the key controller dissolves at least one existing edge. More precisely, the controller dissolves the edge leading from node a to node b where a new intermediate n_{new} is inserted to. So that, a former edge $\langle v_{old}, w_{old} \rangle$ dissolves and node v becomes the new source node of n_{new} and w becomes the new target node of n_{new}. Besides single intermediate node insertion, multiple intermediate nodes insertion is possible as well. Multiple intermediate nodes form a sequence in a row or an entire sub-key graph, so that $level_0 < level_i < level_{n-1}$, whereas $level_i$ includes the root node of a new nodes sequence or a sub-key graph that is inserted.

Formally, we denote intermediate node insertion by $V_{i+1} := V_i \cup \ \{(n_0, \ldots, n_k)\}$ and $E_{i+1} := E_i \cup \ \{(\langle v_0, w_0 \rangle, \ldots, \langle v_k, w_k \rangle) \ \setminus \{\langle v_{old}, w_{old} \rangle\}\}$.

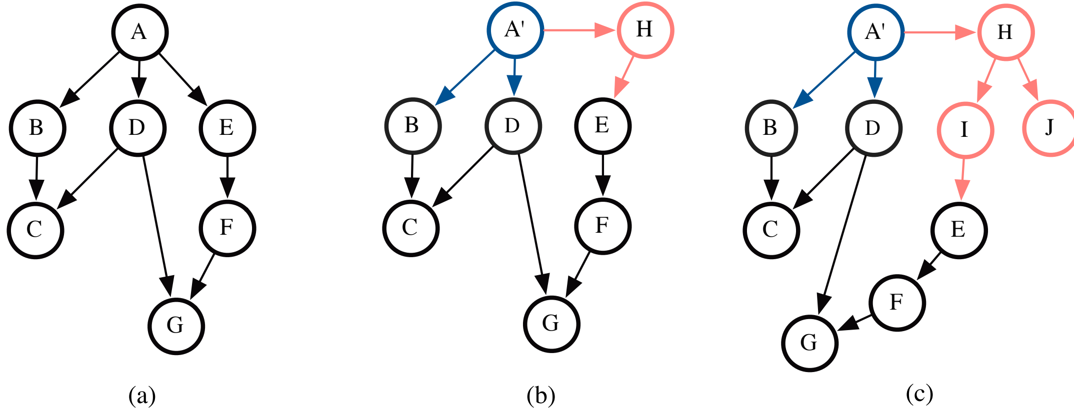

Figure 4.3: Intermediate insertion of (b) single and (c) multiple nodes.

Figure 4.3 (b) illustrates how the key controller inserts a new single intermediate node H by removing the former edge between A and E, and by inserting the new node H with an incoming edge from A and an outgoing edge to E. The key controller directly uses the new child node E of H in order to encrypt its new parent node for generating *Key Trails*. This is necessary to create a new key graph revision and to forward the updated key materials of the affected nodes (in case of 4.3 (b) nodes A' and H) to the corresponding client nodes. The figure colors new or reorganized elements in red. The discussed example in 4.3 (b) generates the following *Key Trails*:

$E_E(H)$, $E_H(A')$, $E_D(A')$, $E_B(A')$

Figure 4.3 (c) shows a multiple intermediate nodes insertion in the form of a sub-key graph. Technically, there is no difference to a single intermediate node insertion. The former child node E constitutes the initial point for *Key Trails* generation and concerns all new inserted nodes in bottom-up manner.

On the contrary to a leaf node insertion, the controller not only encrypts edges of affected existing nodes but also encrypts all incoming and outgoing edges of the new inserted nodes. This is because the new inserted nodes have descendant nodes, and thus the descendant nodes being client nodes need to know the new key materials of the inserted intermediate nodes. To put it simply, a client node needs to know which new inserted intermediate nodes belonging to its sub-key graph after an insertion.

The created *Key Trails* of the example in figure 4.3 (c) are:

$E_E(I)$, $E_I(H)$, $E_J(H)$, $E_H(A')$, $E_D(A')$, $E_B(A')$

4.2.2 Deletion

Single Node (Leaf-Level)

The deletion of a single leaf node is the simplest deletion case. The key controller removes the node and all its incoming edges on a single leaf node deletion. Afterwards, the controller updates the former ancestor node set of the deleted node.

Formally, we denote a single node deletion by $V_{i+1} := V_i \setminus \; n_{old}$ and $E_{i+1} := E_i \setminus \{(\langle v_0, w_0\rangle, \ldots, \langle v_k, w_k\rangle)\}$, where k is the number of n_{old}'s former parent nodes.

Figure 4.4 depicts the original graph (a) and the updated graph (b), where the controller removes the node G and its incoming edges from D and F. Since node G has two different paths to the sole root node A ($A \rightarrow D \rightarrow G$ and $A \rightarrow E \rightarrow F \rightarrow G$), the update affects the nodes A, D, E and F. The key controller creates a new node version and new key material for each of them. The update affects six edges what comes to six encryption steps altogether.

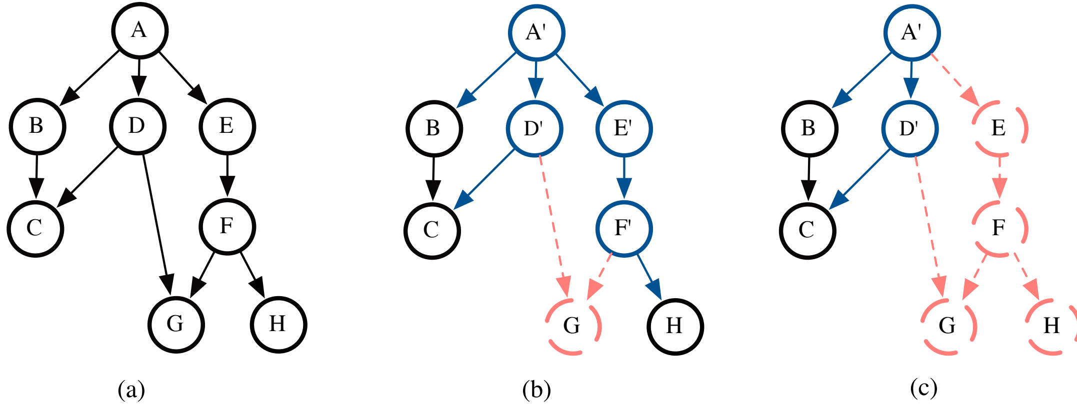

Figure 4.4: Leaf deletion examples of (b) single and (c) multiple nodes.

$E_H(F')$, $E_{F'}(E')$, $E_{E'}(A')$, $E_C(D')$, $E_{D'}(A')$, $E_B(A')$

Multiple Nodes (Leaf-Level)

The deletion of an entire sub-key graph that reaches the key graph's leaf level, is slightly similar as deleting a single node. Note that all descendant nodes going out from the deleting sub-key graph's root node are deleted, too. The controller updates all nodes having an outgoing edge to one of the nodes within the deleting sub-key graph.

We denote multiple nodes deletion on leaf level by $V_{i+1} := V_i \setminus \{(n_0, \ldots, n_k)\}$, where n_0 is the sub-key graph's root node followed by its descendants. $E_{i+1} := A_i \setminus \{(\langle v_0, w_0\rangle, \ldots, \langle v_{k-1}, q_{k-1}\rangle)$, where k is the total number of edges within the deleting sub-key graph, plus the edges leading to nodes in node set $V := V_i \setminus V_{i+1}$.

Figure 4.5 (c) shows how the key controller removes node E and all its descendant nodes from the DAG. The key controller updates all remaining key graph nodes which shared an edge with a deleted node. The update leads to three encryption steps in total.

$E_C(D')$, $E_{D'}(A')$, $E_B(A')$

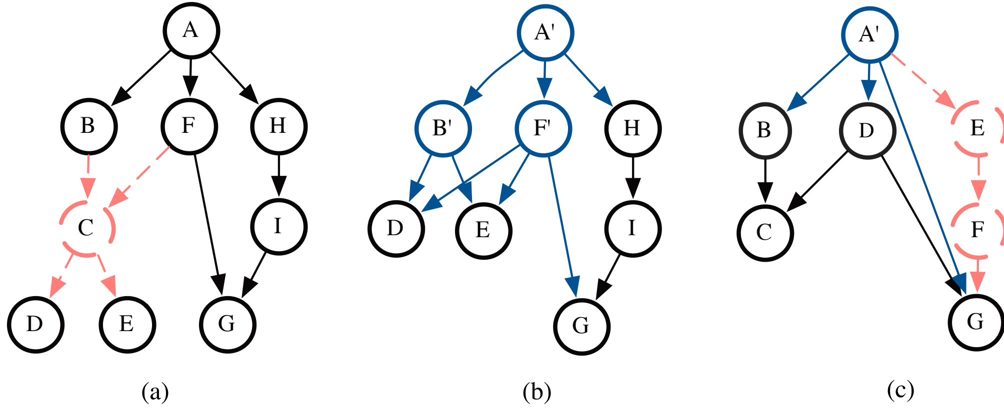

Figure 4.5: Intermediate deletion examples of (b) single and (c) multiple nodes.

Intermediate Node(s)

When the key controller removes an intermediate node within a key graph, unlike in multiple intermediate node deletion the key controller removes only the node itself, and the deleted node's descendant nodes remain into the key graph. As a result of that, the key controller conducts a reorganization of the remaining descendants after deletion. All remaining child nodes get a direct edge to all of the deleted node's parent nodes. The same holds when the controller removes multiple intermediate nodes which do not reach the leaf level as depicted in figure 4.5 (c).

We formally denote intermediate node deletion by $V_{i+1} := V_i \setminus \{(n_0, \ldots, n_k)\}$ and $E_{i+1} := E_i \setminus \{(\langle v_0, w_0\rangle, \ldots, \langle v_k, w_k\rangle)\} \cup \{(\langle v_0, w_0\rangle, \ldots, \langle v_j, w_j\rangle\}$, where k is the number of removed edges and j the number of new assigned edges caused by the reorganization.

Depiction 4.5(a) visualizes the deletion of intermediate node C, in the following denoted as n_{old}. After the reorganization in (b), n_{old}'s child nodes D and E receive a direct edge to all of n_{old}'s former parent nodes. The key controller encrypts all those former parent nodes by their new assigned child nodes. This proceeding and the following bottom-up encryption process result in eight encryption steps as depicted in figure 4.5 (b).

$E_D(B')$, $E_D(F')$, $E_E(B')$, $E_E(F')$, $E_G(F')$, $E_{B'}(A')$, $E_{F'}(A')$, $E_{H'}(A')$

The key graph in figure 4.5 (c) depicts the deletion of multiple intermediate nodes. It shows the deletion of the nodes sequence E to F. The remaining child node of F gets a direct edge with node E's former parent node A. All three child nodes of A (B, D and G) encrypt A' each, resulting in three $Key\ Trails$.

$E_B(A')$, $E_F(A')$, $E_G(A')$

4.2.3 Summary

In the previous subsections, we described eight cases of DAG updates. Some of those update scenarios are simple and others complex. An insertion or deletion of a single node at the leaf level, or of a sub-key graph holding multiple nodes reaching the leaf level, is simple and requires no complex reorganization of parts of a DAG. An insertion or deletion of one or multiple intermediate nodes however, brings along complexity in key graph reorganization as well as in *Key Trails* generation. Especially, many insertions of leaf or intermediate nodes lead to an uncontrolled expansion of the key graph. Which in turn brings out the drawback that a non-restricted key graph leads to an unbalanced node distribution and to a non-linear growing of encryption steps when the number of node insertions increases. In the following chapter, we present the topic of proxy nodes to stem the complexity and the non-linear increasing of encryption steps.

Chapter 5

Proxy Nodes

This chapter covers the topic of proxy nodes on directed-acyclic graphs. By extension, it is an enhancement of the DAG management covered in the preceding chapter and gives a detailed description about proxy nodes, their traits and applications. The goal is to improve the performance of key graph updates and the resulting rekeying process.

5.1 Motivation

As we already described in the preceding chapter, the number of *Key Trails* depends on the number of parent nodes of the update affected nodes. An update affected node with x parent nodes creates x *Key Trails*. We introduce proxy nodes to overcome this obvious drawback within a fine-meshed DAG containing a large number of nodes with continual updates. Proxy nodes head to a proper scaling since the number of encryption steps does not base on the adjacent edges of affected nodes, but on the number of affected nodes itself. The implementation of proxy nodes improves the overall scalability of a DAG as previous research work[3] show. Additionally, we show that proxy nodes reduce the number of created *Key Trails* on updates in a directed-acyclic key graph.

5.2 Description

A proxy node is a special node that is used to constrain a directed key graph and control its expansion. We define a threshold f that is called fanout in order to balance

and trim a key graph structure and to limit its outgrowth. A fanout represents the maximal number of outgoing edges of a root or intermediate node. If a node $node_{par}$ gets an additional outgoing edge to a new node $node_{new}$ resulting in the exceedance of $node_{par}$'s fanout, a proxy node $node_{proxy}$ is created and inserted between the new node and its actual parent node, so that levels are $node_{par} < node_{proxy} < node_{new}$. Instead of $node_{par}$, $node_{proxy}$ gets a new outgoing edge directing to $node_{new}$. Formally, we denote this procedure by $f \geq outgoing(v) :=| \{\langle v, w\rangle \in E \mid w \in N\} |$. If $outgoing(v) > f$, a new proxy node is inserted and the new sub-key graph path sequence turns from $\langle v, w\rangle$ into $\langle v, p, w\rangle$. p acts as a proxy between v and w. We denote this as follows: if $outgoing(v) > f : E_{i+1} := E_{i+1} \setminus \{\langle v, w\rangle\},\ E_{i+2} := E : i+1 \cup \{\langle v, p\rangle, \langle p, w\rangle\}$. The more new outgoing edges are added to node v, the more often the fanout thresholds of already inserted proxy nodes exceeds, and the more new proxy nodes are inserted, resulting in $\langle v, p_0, \ldots, p_n, w\rangle$. On each new proxy node insertion, p_n is split into p_{n+1} and p_{n+2}, whereas p_{n+1} lies on the direct path only and p_{n+2} forms a new sub path. Each path that is going from v to w leads over p_{n+1}.

The layout of a proxy node is exactly the layout of any other kind of node within a DAG as shown in figure 3.5. Also, their secret key materials are used to encrypt adjacent incoming nodes the same way. In summary, a proxy node does the same things as any other node in a key graph does - except encrypting storage data.

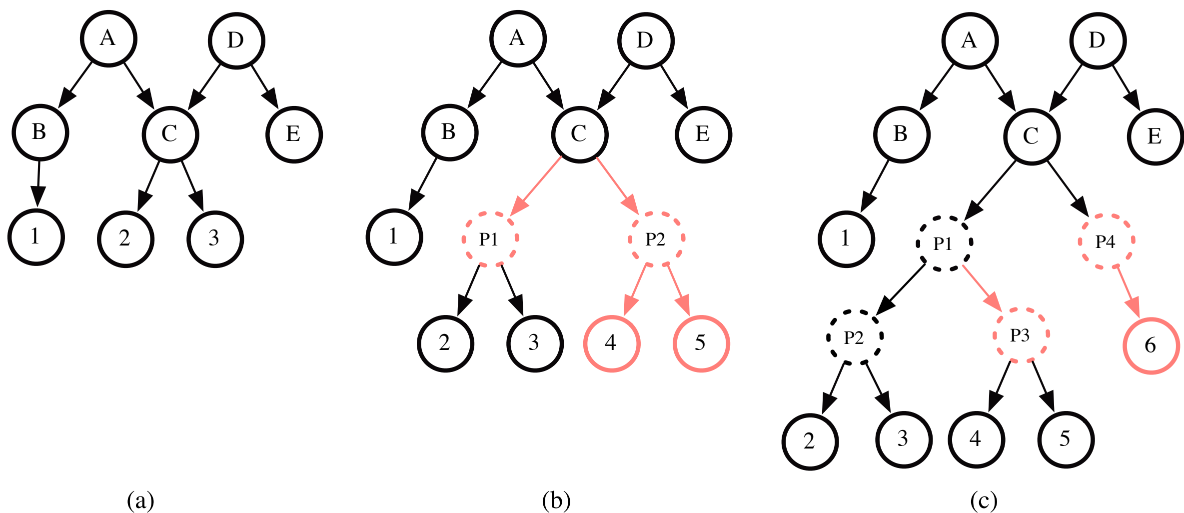

Figure 5.1: Proxy nodes creation and key graph reorganization.

We depict a case where proxy nodes are inserted when the fanout exceeds two in

figure 5.1. We add two additional nodes 4 and 5 to the original key graph in figure (a). Since four outgoing edges exceed the defined threshold, the proxy nodes p_1 and p_2 are inserted as depicted in figure (b). Both proxy nodes connect the four outgoing edges with node C as intermediator nodes and comply with the threshold limit. Figure 5.1 (c) shows a further edge addition to node C. Since both inserted proxy nodes in (b) reach the fanout, a further split is conducted, resulting in two new proxy nodes p_3 and p_4. In the process, p_2 is reorganized and attached to p_1, proxy node p_3 becomes the right counterpart of p_2 and p_4 the right counterpart of p_1. The new node 6 gets an new incoming edge from p_4. Further node attachments to node C would result in stocking up the left branch path under p_4 until the right sub-key graph is balanced like its left equivalent, before a new split of all proxy nodes is conducted again. In this way, proxy nodes keep the graph well-balanced.

5.3 Key Trails on Proxy Nodes

After we introduced the characteristics of proxy nodes on a *DAG*, we show how the number of generated *Key Trails* changes by using proxy nodes. Due to the limitation by a maximal number of outgoing edges for each root and intermediate node, the number of generated *Key Trails* on updates is limited as well. Each update affected node generates f *Key Trails* at maximum. This results in a proper key graph scaling, since the number of *Key Trails* does not base on the adjacent edges of affected nodes. That decreases the total number of generated *Key Trails* compared to a directed-acyclic key graph that does not implement proxy nodes.

We define $\sum_{i=0}^{n} outgoing(v_i)$ to calculate the set of created *Key Trails* on updates, where n is the number of affected nodes and v_0 the node a new outgoing edge is attached to. The formula applies to restricted and non-restricted key graphs similarly.

Considering the examples in figure 5.2 (a). The key graph does not implement proxy nodes. Assume, no key graph restriction exists and a key graph allows to add as many as possible outgoing edges to a node. An insertion of a new node 6 with an incoming edge

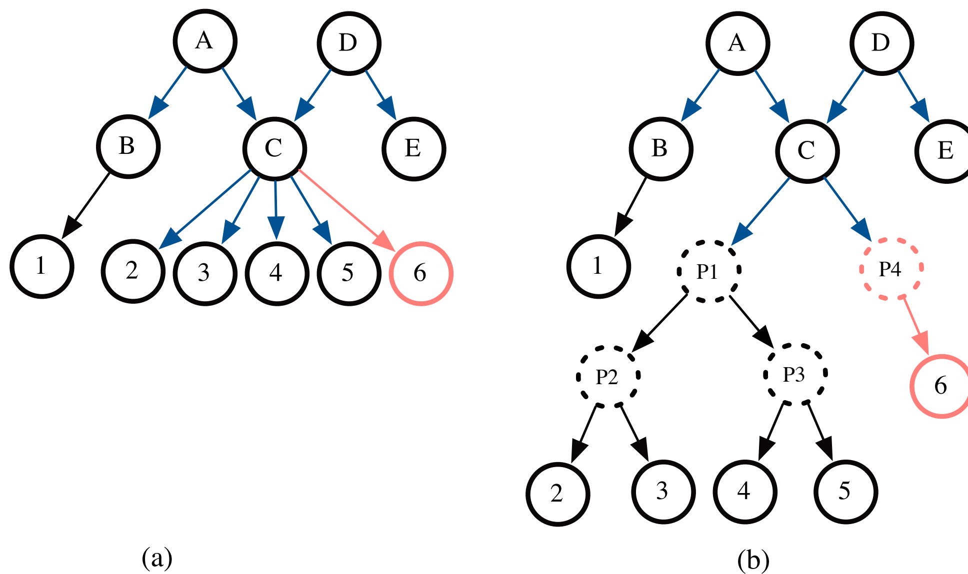

Figure 5.2: Encryption steps on proxy nodes.

from node C leads to seven *Key Trails* generations. All blue-colored edges including the new red inserted edge express the single encryption steps. The depicted key graph in 5.2 (b) contains the same set of V as (a), but a controller limits the outgoing edges per node to two. The key graph implements proxy nodes and becomes obviously more well-balanced than a non-restricted key graph. An insertion of node 6 leads to a new proxy node insertion resulting in a key graph reorganization, so that node 6 gets an incoming edge from proxy node p_4. The overall number of generated *Key Trails* is seven.

This example shows that even in a tiny key graph the number of *Key Trails*, and thus the number of encryption steps shrinks significantly. The disparity of encryption steps between the approach using proxy nodes and the one that forgoes proxy nodes, increases with the growth of nodes in a key graph.

[3] shows that the best performing fanout is four. Due to this fact, we are going to use a fanout of four as standard set up for all our following practical examinations.

Chapter 6

Performance Evaluation

After we presented the architecture and the implementation of our *VersaKey*-based prototype, the *DAG* management and the proxy nodes topic, this chapter presents, examines and discusses the received results by using our implemented prototype. We performed several tests using proxy nodes and using no proxy nodes and compared the received results to each other. The performance evaluations base on a *DAG* constructed by using a randomly created test data set as described in 6.1.

6.1 Testing Preliminaries

Environment

We used a testing machine with the following technical attributes for all our tests and examinations:

Hardware

- iMac Intel Core 2 Duo 3,06 GHz
- 4 GB 1067 MHz DDR-RAM

Software

- Mac OS X 10.7
- Eclipse 3.6 / Netbeans 7.0.1 Profiler (Java 1.6)

Data Set

We developed an own test data set generator since there exists neither a standardized benchmarking data set or generator nor a suitable authorized and applicable real word data set for performing benchmarks on hierarchical group management frameworks. The generator is able to create data for a certain amount of group nodes and client nodes which are added to a directed-acyclic key graph. It generates three kinds of nodes: Root nodes, group nodes and client nodes. The generator equally distributes the generated nodes among a defined maximal graph level and checks the inserted node connections for cycles. Further, it limits the maximal number of outgoing nodes by creating proxy nodes when the outgoing edges of a group node exceeds outdegree.

The generator works in the following manner: At first, it creates the defined number of root nodes. A root node has no incoming edges. Then, the generator randomly creates group nodes. Each root and group node name is unique and consists out of arbitrary ten characters. The generator assigns randomly one, two or three parent nodes or incoming edges to a group node. An incoming edge defines an edge from a node that has already been generated and is randomly chosen out of the already generated root and group node set. At last, the generator creates the user nodes. Each user node consists out of a name in the form of „$User_X$", where X is an unique number from zero to $(n-1)$. Hereby, n is the defined number of maximal user nodes. Each user node has one to three incoming group edges which are randomly chosen from the already generated root and group node set.

We generated and applied a data set with the following attributes for performing the tests:

- 8 root nodes
- 1000 group nodes

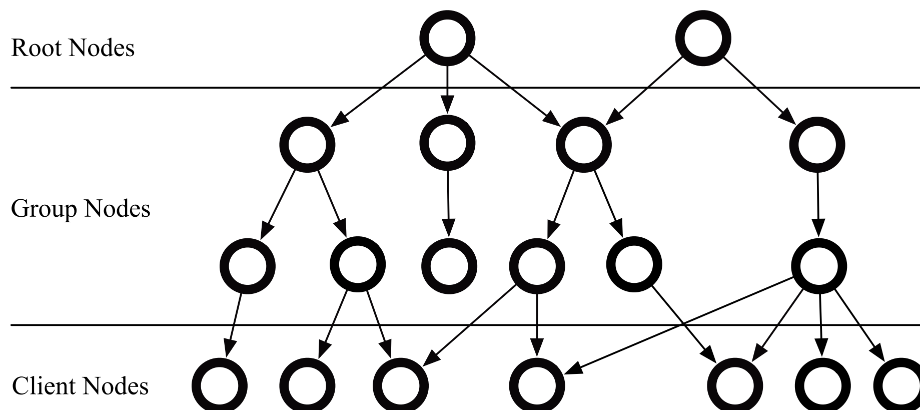

Figure 6.1: A directed-acyclic key graph showing the different node types.

- 6400 user nodes
- maximal node indegree of three
- maximal node outdegree of four (fanout)
- maximal key graph depth of ten

Application

We examined the scalability and the performance by performing three iterations employing 250, 500 and 1000 group nodes each. This was applied two times. Once implementing proxy nodes and once without implementing proxy nodes to a *DAG*. The number of employed client nodes is 6400 on each iteration. The procedure works as follow: Before we applied the measurements, a key graph consisting out of root and group nodes has been created. The insertion of root or group nodes does not trigger the encryption process, and thus it does not create *Key Trails*. The *Key Trails* generation starts with the insertion of client nodes. The generation of *Key Trails* does not make sense without having client nodes employed to the key graph. The generator inserts each of the 6400 client nodes one by one. The algorithm of the generator uniformly distributes the client nodes over the existing root and group node set. Each client node insertion triggers the encryption process and creates *Key Trails*. The *Key Trails* are then distributed to the incident clients according the procedure as described in subchapter 3.3. A measurement

is performed after a predefined number of inserted client nodes. At each measured point, we measured:

- the total number of generated *Key Trails*
- the total generation time of the generated *Key Trails*
- the average generation time for a single *Key Trail*
- the number of affected nodes
- the number of affected nodes accumulated

We defined eight measured points (50, 100, 200, 400, 800, 1600, 3200, 6400) for our tests.

6.2 Results and Discussion

Number of Generated Key Trails

We performed the first evaluation by measuring the total number of *Key Trails* created by a client insertion update within a key graph containing a certain number of clients. Meaning, we counted all created *Key Trails* which were created by an insertion of a single client node at one of the predefined measured points. Hence, the earlier the measurement is performed or the less client nodes have employed to the *DAG* yet, the less *Key Trails* are created. Figure 6.2 visualizes the results of the first evaluation. The logarithmic scale on the y-axis depicts the number of created *Key Trails*. The x-axis depicts the number of client nodes contained in the key graph when an insertion update is applied. Figure 6.2 (a) shows that the usage of no proxy nodes leads to a linear *Key Trails* creation growth relative to the employed client nodes. The more group nodes a key graph provides to join, the more encryption steps are conducted.

An obvious question arises by observing this progress. Might there not be less created *Key Trails* if the key graph contains more group nodes, and thus the key graph is lesser spread out and linked? This is a correct assumption when each group or client node would have exactly one incoming edge. But, since a group node has multiple

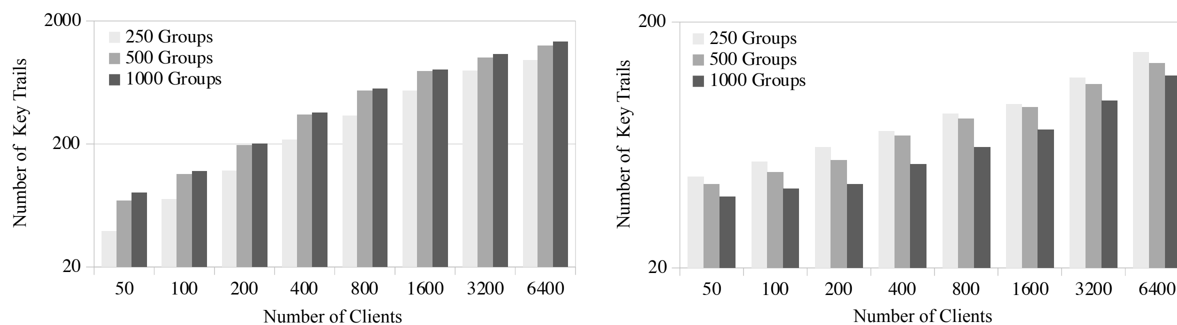

Figure 6.2: Number of *Key Trails* (a) without proxy nodes (b) with proxy nodes

incoming edges, the connections between group nodes increases. Therefore, it is an excepted behaviour caused by the additional edges a high number of group nodes bring along. We conclude, the more group and client nodes are employed to the *DAG*, the more *Key Trails* are generated.

The opposite is partly true if proxy nodes are implemented as depicted in figure 6.2 (b). A *DAG* gets a different layout by implementing proxy nodes. The total amount of created proxy nodes decreases. The number of *Key Trails* still increases with the rise of client nodes, but the more groups are applied, the less *Key Trails* are generated. The more group and client nodes are added, the more proxy nodes are inserted. On the contrary to a „normal“ group node, a proxy node consists out of only one incoming edge, and thus causes fewer *Key Trails*.

We conclude that the approach using proxy nodes scales properly and creates fewer *Key Trails*. The use of proxy nodes reduces the generated *Key Trails* with the increasing number of inserted group and client nodes. The average factor of generated *Key Trails* is about 10 smaller by using proxy nodes. Also, the more group and clients nodes are inserted to the key graph, the bigger is the disparity of generated *Key Trails* between the approach using proxy nodes and the one using no proxy nodes. In real world applications with thousands and thousands of clients having different access rights, this is a desired and a satisfactory effect.

Generation Time for Key Trails

The second evaluation aims to measure the total generation time of all *Key Trails* generated on an update at one of the defined measured points. In other words, the evaluation measures the time required for all *Key Trails* generated in the first evaluation test. Figure 6.3 (a) depicts the time needed for *Key Trails* generation without inserting proxy nodes. The generation time increases with the growth of client nodes. This observation results from the fact that the generation of a single *Key Trail* does not only comprise the encryption of new key material, but it also includes the database search for the old key that encrypts the new key material. The database search times increase with the continuous inserting of client nodes to the *DAG*. Each client insertion results in a new key graph revision. Hence, the generation time for *Key Trails* increases with the number of revision nodes stored into the database. The time for the encryption process itself remains constant.

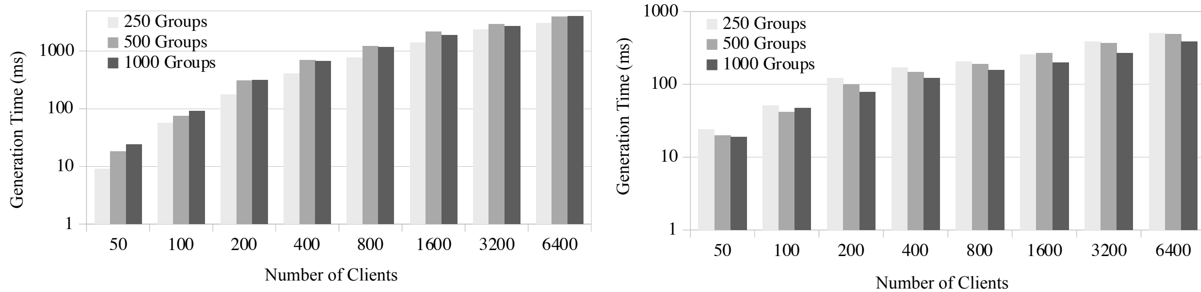

Figure 6.3: Generation time of *Key Trails* (a) without proxy nodes (b) with proxy nodes.

The database contains root, group and client nodes, and it also stores proxy nodes. For that reason, the approach using proxy nodes needs longer for *Key Trails* generation due to the bigger database size. The searching time for a key material to encrypt a new key material depends on the database size. By looking at the progress in figure 6.3 (b), we observe that the first measured point performed after 50 inserted client nodes, indeed takes longer than the equivalent measurement in figure 6.3 (a). However, this observation only applies to the first measured point. All following measured points take lesser time. The smaller number of generated *Key Trails* has more influence on generation time than the database search overhead caused by the stored proxy nodes. We also observe that the time scales with the number of generated *Key Trails* as depicted in figure 6.2. In figure 6.3 (a), we observe a discrepancy for 800, 1600 and 3200 inserted client nodes

applied to 500 group nodes compared to 250 and 1000 group nodes. In fact, there are only minimal time differences caused by concurrent processes on the testing machine. A control experiment show that these discrepancies dissolve by having no major concurrent processes on the testing machine. At the control experiment, the time for 500 groups is smaller than for 1000 groups and higher than for 250 groups.

Generation Time for a Single Key Trail

The third test results directly from the second evaluation. It examines the average generation time for a single *Key Trail*. The evaluation makes the trend clearer that we already observed in figure 6.3. The more nodes are added to a *DAG*, and thus stored into the database, the more time it needs for looking up the encryption key to encrypt the new key material. Figure 6.4 (a) visualizes this trend. The distinctions between the three group node amounts are noticeable at the beginning. Along with the increasing of client nodes, the generation time for a single *Key Trail* approximates similar to the total generation time observed in figure 6.3. Furthermore, the statistical inaccuracies decline and the measured values become more stable. It also shows our previous assumption that key material encryption using proxy nodes needs about 15-20% longer than using no proxy nodes. The slight variations in the curves are explainable by the low measurement range of milliseconds.

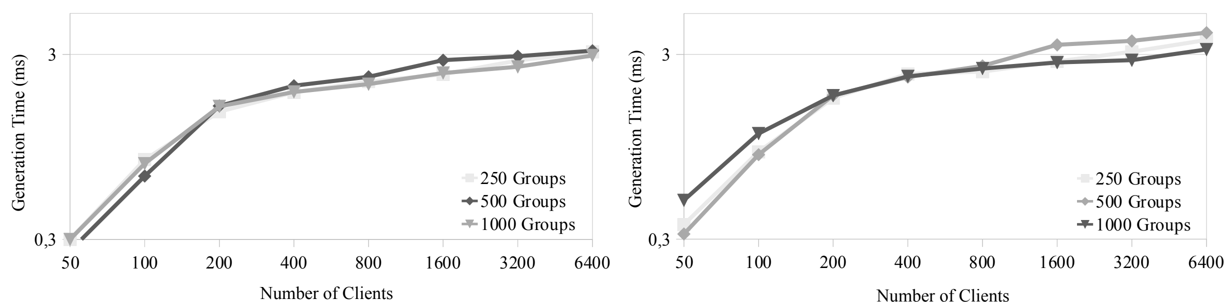

Figure 6.4: Generation time of a single *Key Trail* (a) without proxy nodes (b) with proxy nodes.

Affected Nodes

Our last test evaluates the affected nodes resulting by an update. The y-axis depicts the number of affected nodes and the x-axis depicts the number of client nodes contained in the key graph when an insertion update is applied. The number of affected nodes remains constant when no proxy nodes are inserted. Only intermediate nodes and root nodes are affected by an update. By inserting no proxy nodes, the number of root and groups nodes does not change, no matter how many client nodes are inserted to the key graph. Therefore, the number of affected nodes does not change either. The benchmark chart in figure 6.5 (a) emphasizes this effect. A *DAG* containing 250 group nodes affects 34 nodes constantly, a *DAG* containing 500 group nodes affects 46 nodes and a *DAG* containing 1000 group nodes affects 64 nodes. The doubling of the number of group nodes costs about 35-45% more affected nodes.
An obvious question arises similar to the question arose while the total number of *Key Trails* generation discussion in chapter 6.2 by observing this progress. The number of affected nodes must decrease with the increasing of group nodes within a *DAG*, resulting in a lesser spread out and linked key graph? This assumption is right if each group or client node would have exactly one incoming edge. Since a group or client node has up to three incoming edges, the key graph is more linked. Which in turn leads to a tightly knit key graph network causing in many possible leaf to root paths including many distinct nodes.

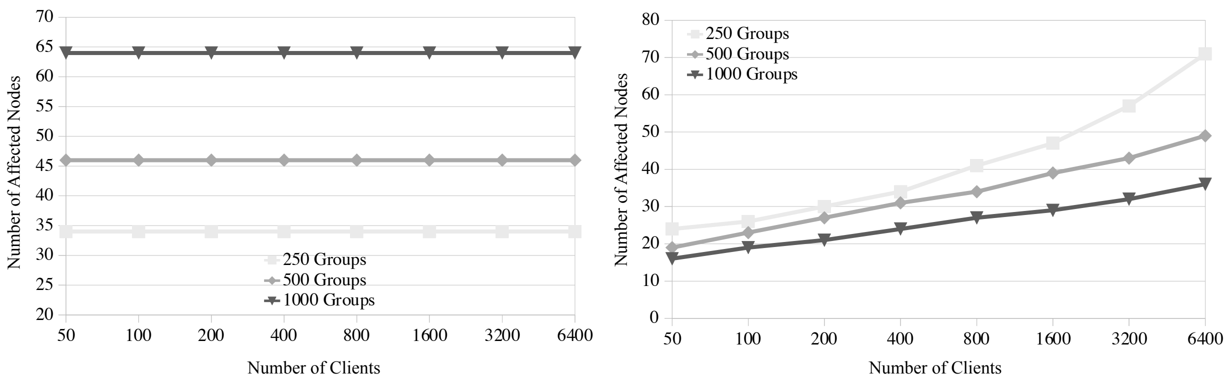

Figure 6.5: Number of nodes affected (a) without proxy nodes (b) with proxy nodes.

A *DAG* using the proxy nodes approach forms a different key graph layout. The key graph layout changes by inserting new client nodes. Meaning, the continuous insertion

of new client nodes triggers the creation of proxy nodes as soon as the number of outgoing node edges reaches the outdegree threshold. Therefore, the key graph grows not only by inserting new client nodes, but also by the automatic insertion of proxy nodes. The insertion of proxy nodes leads to an increasing number of affected nodes with the increasing number of inserted client nodes. Benchmark 6.5 (b) uses proxy nodes and visualizes the described effect. At the first measured points, the scaling results in a lesser number of affected nodes on an update, since the usage of proxy nodes loosens the key graph. The scaling does not behave linear anymore with the increasing number of clients node insertions. Figure 6.5 (b) also shows an expected characteristics. The more client nodes with up to three incoming edges are inserted to the *DAG*, the faster the maximum outdegree of group nodes is reached, resulting in many proxy nodes insertions. This affects more nodes (including proxy nodes) on updates.
Furthermore, we observed that the number of affected nodes decreases by decreasing the maximal number of incoming edges of intermediate and client nodes. This reduction results in less leaf to root paths.

Figure 6.6 represents the affected nodes accumulated within all versions. It shows that the affected nodes without the use of proxy nodes scales linear and it emphasizes the assumption that only a constant number of nodes is updated caused by a client node insertion.

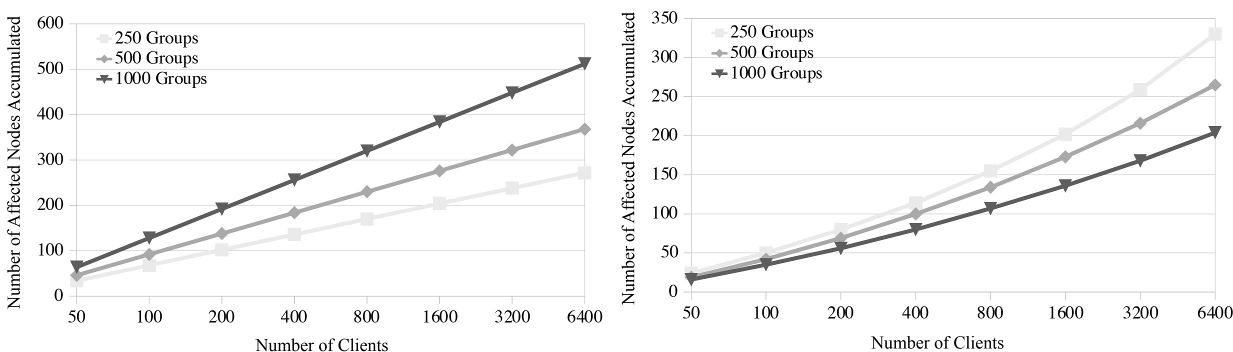

Figure 6.6: Accumulated nodes affected (a) without proxy nodes (b) with proxy nodes.

Note that a different *DAG* layout with a different setup causes different results. This applies to all examinations and tests we made.

Chapter 7

Conclusion

7.1 Final Summary

In this thesis, we presented the idea and a prototype implementation of a version-aware encryption framework mapped on revisioned data by using a directed-acyclic key graph as hierarchical group management structure. The key graph enables join and leave changes of node entities without re-encrypting the storage data. We introduced proxy nodes to decrease the encryption steps on updates and make the key graph scalable. We introduced the idea of *Key Trails* as ananother term for encryption steps representing the encrypted edges adjacent to an updated node. *Key Trails* enable to send newly encrypted key materials over an untrusted component without having the fear of interception and unauthorized de-encryption. We focused less on the framework itself, rather on the efficient *DAG* management and the examination of proxy nodes. Proxy nodes counteract the not always linear growing of encryption steps in a closely linked key graph with the increasing number of client nodes. We show that updates on a key graph scale acceptable relative to the ancestor nodes of the closest update affected node. Even better, we show up that the limitation of a key graph using proxy nodes improves the results and scalability regarding the number of encryption steps of the nodes adjacent to the updated ancestor nodes. Furthermore, our evaluations revealed a non-excepted behaviour in which the number of generated *Key Trails* decreases by increasing the number of root and intermediate nodes, and by inserting proxy nodes. Whereas in a key graph without inserting proxy nodes, the number of *Key Trails* scales with the number of nodes.

7.2 Future Outline

While writing and researching, some ideas for improvements and further development come up which could not be considered in this work. We present these ideas in order to give suggestions for further research.

The current implementation enables to get access to former key graph versions when a client is removed from the *DAG* and is inserted again later. This procedure violates the guarantee for only getting access to the data shared during group affiliation. A possible idea to overcome this lack of feature has already been proposed in [8]. Graf et al. proposes the deployment of an authorization layer. The approach grants access to a former version by negotiating a token between the *Key Manager* and a distributed instance if the client is authorized to access a requesting data source. There are certainly other approaches to overcome this kind of problem.

To make a centralized *DAG* obsolete and bring it on a distributed environment (i.e. store it into a cloud infrastructure) is another field of further development. The distribution of a *DAG* eliminates a single point of failure.

Further, we think there is much space to improve the current implemented architecture of versioning key graphs. As the evaluation results show in figures 6.3 and 6.4 , the overhead for searching a database holding revisions increases nonlinear with the number of updates on the key graph. This affects the generation time of *Key Trails* directly. To speed up the *Key Trails* generation and to avoid the overhead, more efficient storing and restoring mechanisms and algorithms are needed for versioning key graphs by reducing the nodes needed to store per update. The use of hypergraphs instead of a *DAG* is an alternative. We already mentioned and sketched hypergraphs in chapter 4.1. Hypergraphs allow to reach multiple nodes over one hyperedge directly. Each hyperedge directs to multiple versions of a node. Hereby, only one edge holding the corresponding secret materials is stored per version, instead of the entire node data including its corresponding incoming and outgoing edges.

Bibliography

[1] S. Rafaeli, and D. Hutchison, „A survey of key management for secure group communication“, *ACM Computing Surveys*, vol. 35, no. 3, pp 309-329, Sep. 2003.

[2] M.J. Moyer, J.R. Rao, and P. Rohatgi, “A survey of security issues in multicast communications,” *IEEE Network*, vol. 13, no. 6, pp. 12–23, Nov.-Dec. 1999.

[3] C. K. Wong, M. Gouda, and S. S. Lam, “Secure Group Communications Using Key Graphs,” Department of Computer Sciences, The Univ. of Texas at Austin, *Tech. Rep.* TR-97-23, July 1997.

[4] D. M. Wallner, E. J. Harder, and R. C. Agee, “Key management for multicast: issues and architectures,” *Informational RFC*, July 1997.

[5] 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, August 1999.

[6] W. Diffie, "Authenticated key exchange and secure interactive communication", in *Processdings of 8th Worldwide Congress on Computer and Communications Security and Protection: SECURICOM ’90*, 1990, pp. 300-306.

[7] S. Graf. "Treetank, a native xml storage". *XML-Prague 2011*, Posterpaper, Prag.

[8] S. Graf, P. Lang, S. A. Hohenadel, M. Waldvogel. "Versatile Key Management for Secure Cloud Storage". Submitted at *EuroSys '11*. 11.4.2012 - 13-04-2012, Bern.

[9] S. Banerjee, and B. Bhattacharjee, "Scalable secure group communication over IP multicast," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 8, pp. 1511-1527, Oct. 2002.2004.

[10] M. G. Gouda, H. C. Tser, and E. N. Elnozahy, "Key trees and the security of interval multicast," *Proceedings of 22nd International Conference on Distributed Computing Systems*, pp. 467–468, 2002.

[11] Y. Kim, A. Perrig, G. Tsudik, "Tree-based group key agreement," *ACM Transactions on Information and System Security*, vol. 7, no. 1, pp. 60-94, Feb. 2004.

[12] A. Penrig, D. Song, and D. Tygar, "ELK: A new protocol for efficient large-group key distribution," *Proceedings of IEEE Symposium on Security and Privacy*, pp. 247–262, May 2001.

[13] Y. Sun, W. Trappe, and K. J. R. Liu, "A scalable multicast key management scheme for heterogeneous wireless networks," *IEEE/ACM Transactions on Networking*, vol. 12, no. 4, pp. 653-666, Aug. 2004.

[14] X. Zhang, S. Lam, D. Lee, and Y. Yang, "Protocol design for scalable and reliable group rekeying," *IEEE/ACM Transactions on Networking*, vol. 11, no. 6, pp. 908-922, Dec. 2003.

[15] Y. Amir, et. al., "Secure group communication using robust contributory key agreement," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 5, pp. 468-480, May 2004.

[16] K. C. Chan, and S. H. G. Chan, "Key management approaches to offer data confidentiality for secure multicast," *IEEE Network*, vol. 17, no. 5, pp. 30-39, Sep. 2003.

[17] L. R. Dondeti, S. Mukherjee, and A. Samal, "DISEC: a distributed framework for scalable secure many- to-many communication," *Proceedings of 15th IEEE Symposium on Computers and Communications*, pp. 693–698, 2000.

[18] O. M. Erdem, "Efficient self-organized key management for mobile ad hoc networks," *IEEE Global Telecommunications Conference (GLOBE- COM)*, vol. 4, pp. 2185–2190, Dec. 2004.

[19] J. Raymond, and A. Stiglic, "Security Issues in the Diffe- Hellman Key Agreement Protocol," *IEEE Transactions on Information Theory*, pp. 1-7, 1998.

[20] M. Steiner, G. Tsudik, and M. Waidner, "Diffe-Hellman key distribution extended to group communication," proceedings, *The 3rd ACM Conference on Computer and Communications Security*, pp. 31–37, 1996.

[21] M. Steiner, G. Tsudik, and M. Waidner, "Key agreement in dynamic peer groups, *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 8, pp. 769-80, Aug. 2000.

[22] W. Trappe, Y. Wang, and K. J. R. Liu, "Resource-aware conference key establishment for heteroge- neous networks," *IEEE/ACM Transactions on Networking*, vol.

13, no. 1, pp.134-146, Feb. 2005.

[23] Y. Sun, and K. J. Ray Liu, “Scalable hierarchical access control in secure group communications” *IEEE INFOCOM*, vol. 2, pp. 1296–1306, March 2004.

[24] J. Wang and X. Qi, “Key management for differentially secure multicast,” *2nd IASTED International Conference on Communications, Internet & Information Technology*, pp. 226–231, Nov. 2003.

[25] X. Zou, "Secure Group Communications and Hierarchical Access Control", PhD. Thesis, University of Nebraska-Lincoln, USA, 2000.

[26] Qiong Zhang, Yuke Wang, and Jason P. Jue, "A Key Management Scheme for Hierarchical Access Control in Group Communication", Vol. 7, No. 3, 2008, pp. 323-334.

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

[28] C. K. Wong and S. S. Lam, "Keystone, A Group Key Management Service", *International Conference on Telecommunications*, 2000.

[29] H. Ragab Hassen and A. Bouabdallah and H. Bettahar, "A New and Efficient Key Management Scheme for Content Access Control within Tree Hierarchies", *Advanced Information Networking and Applications Workshops*, 2007.

[30] Johnson, Donald B., "Efficient algorithms for shortest paths in sparse networks", *Journal of the ACM* 24 (1): 1–13.

[31] Russell, Stuart J.; Norvig, Peter (2003), *Artificial Intelligence: A Modern Approach (2nd ed.)*, Upper Saddle River, New Jersey: Prentice Hall, ISBN 0-13-790395-2.

[32] Pohl, I., "Bi-directional Search", in Meltzer, Bernard; Michie, Donald, *Machine Intelligence*, 6, Edinburgh University Press, pp. 127–140.

[33] Pearl, J., "Heuristics: Intelligent Search Strategies for Computer Problem Solving". Addison-Wesley, 1984. p. 48.

[34] Lowerre, Bruce. "The Harpy Speech Recognition System", Ph.D. thesis, Carnegie Mellon University, 1976.

Statement of Authorship

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Konstanz, January 13, 2012

Patrick Lang