A Secure Cloud Gateway based upon XML and Web Services

Sebastian Graf
Distributed Systems Group
University of Konstanz
Email: sebastian.graf@uni-konstanz.de
supervised by Prof. Dr. Marcel Waldvogel

***Abstract*—Storing data in the cloud offers a scalable and easy way to handle large amounts of data guaranteeing availability and scalability by the hosting *Cloud Service Providers*. The price for the gained availability is uncertainness about the integrity and confidentiality of the data. Even if common approaches provide high availability and end-to-end encryption necessary to achieve *Availability* and *Confidentiality* as security goals, other security requirements like *Integrity* and *Accountability* are neglected. The key management of those clients for encrypting data to satisfy *Confidentiality* must furthermore support join-/leave-operations within the client set. This work presents an architecture for a secure cloud gateway satisfying the common security goals *Availability*, *Confidentiality*, *Integrity* and *Accountability*. Mapping these security goals, XML as storage base is equipped with recursive integrity checks, encryption and versioning based on the native XML storage *Treetank*. A *Key Manager* extends this approach to provide the deployment of multiple clients sharing keys to the storage in a secure way. New key material is pushed to a server instance deployed as Platform-as-a-Service (*PaaS*) propagating this update to the clients. The server furthermore applies integrity checks on encrypted data within transfer and storage. Any communication between client, server and *Key Manager* relies on fixed defined workflows based upon web services. The proposed architecture called *SecureCG* thereby enables collaborative work on shared cloud storages within multiple clients ensuring confidentiality, consistency and availability of the stored data.**

I. Motivation of Secure Cloud Storage

The flexibility of *Cloud Based Services* offers great possibilities to store any data in a guaranteed available and scalable manner. All data peculiarities ranging from block based byte chunks to *Microsoft Word*-documents are thereby persisted in the cloud. *Cloud Storage Gateways* represent convenient applications mapping such interfaces to common *Cloud Service Providers*. Most *Cloud Storage Gateways* thereby appear as standalone clients or centralized web applications.

Even if *Cloud Storage Gateways* enable the power of *Cloud Based Services* to different kinds of clients, the gained *Availability* comes at a price. While *Availability* and *Confidentiality* are provided by the hosting of the encrypted primary data on *Cloud Service Providers*, *Integrity*, *Confidentiality* and *Accountability* are not sufficiently considered in existing approaches. Another drawback of current approaches is the dependency of *Cloud Storage Gateways* to the platform they run on, e.g. they are shipped as complete operating system images, hard coded in routers or delivered as platform dependent applications. Flexible usages representing libraries, server based infrastructures or user specific clients are not supported. A more flexible usage includes collaborative use cases and related distributed environments. Even if *Availability* and *Confidentiality* are already provided, the related key management must be adapted to satisfy this usage. Common centralized access controls use the same key for all authorized clients which complicate modifications on the set of authorized clients. Besides the necessary protection of the data from unauthorized access of excluded clients, updated key material must be propagated to the valid clients in a scalable and secure manner.

An XML based architecture as *Cloud Storage Gateway* named *SecureCG* satisfies all security requirements, platform independence and flexible key management. Based upon the native XML storage *Treetank* [3], any data is wrapped on demand into XML[13] and persisted afterwards in the cloud. The underlaying tree structure enables *SecureCG* to provide easy ways of integrity checks supporting the security goal of *Integrity*. The *Accountability* is guarded through the native versioning of XML within *Treetank*. Encryption on the data satisfies the goal of *Confidentiality* whereas the underling tree structure of the XML plus the provided versioning functionality is exposed. *SecureCG* supports flexible handling of multiple *Treetank* clients with unique de-/encryption keys using a standalone *Key Manager* and the *VersaKey* approach[12]. Equipped with the platform independent, block based interface *jSCSI* [7], and the adaptive *REST* based interface *JAX-RX*[4], the client provides a flexible and secure storage interface. The storage itself is encapsulated by a server deployed as *Platform as a Service*(*PaaS*) implementation. The *PaaS* implementation provides own integrity checks to ensure *Integrity* within the transmission and the storage of the data. Even if the server only stores encrypted data to ensure *Confidentiality* on *Cloud Based Services*, the server also checks and propagates new encrypted keys within the authorized client set. Communication between client, server and *Key Manager* relies on web services with defined workflows. Within this distributed architecture and as a consequence thereof the scattered functionality, *SecureCG* fulfill the security requirements even on untrusted storage while working with all different types of data.

Treetank as the base of *SecureCG* implements already the *Accountability* and partly the *Confidentiality* and the *Integrity*.

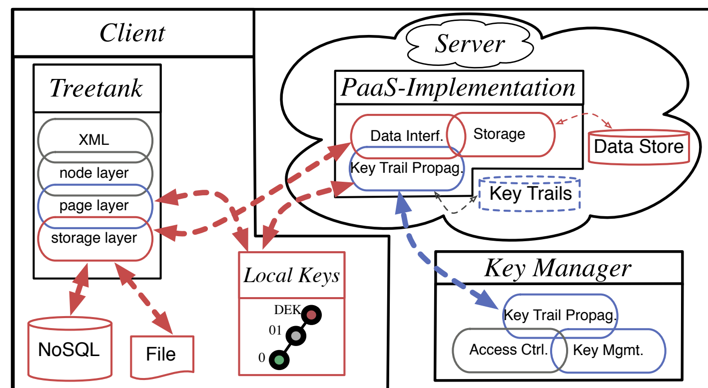


Fig. 1: Overview of proposed architecture

Parts of the client were already released to the open source community[3]. Results on local machines show constant access time regarding read- and write-access of any node within any version of the XML which is important since requests on single substructures must scale in the cloud. *SecureCG* is provided throughout as plain *Java* implementation which enables flexible usages independent from the underlaying platform.

Current approaches making use of cloud storages either use web services or pull the access to file system level based upon iSCSI or NFS. By utilizing web services as primary communication within a distributed architecture and by offering additional interfaces with the ability to store data on block level based upon *jSCSI*, *SecureCG* provides a flexible toolset for secure storage on untrusted third party applications. The distributed architecture consisting out of client, server and *Key Manager* allows scalable and flexible usage without any restrictions regarding security requirements or functionality.

II. Applying security to *Cloud Storage Gateways*

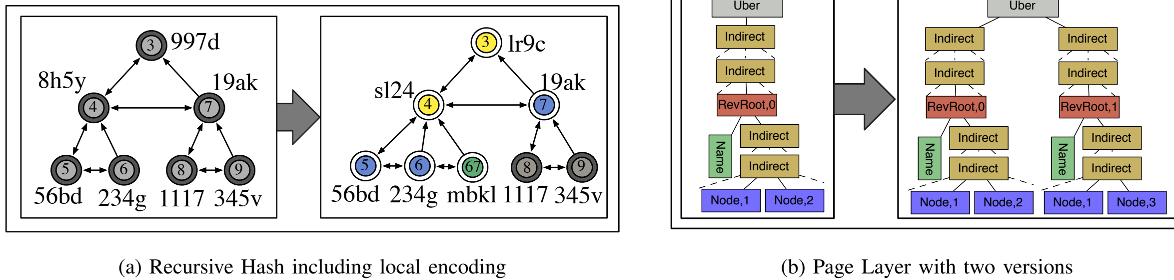
Common *Cloud Storage Gateway*-approaches claim to store data "secure" on untrusted third party *Cloud Service Providers* by encrypting it. Since the encryption operations take place either on client side or on trusted applications within the *Cloud Storage Gateway* providers, the stored data indeed satisfies the requirement of *Confidentiality*. Hence, the key management offers no adaptive key handling supporting an evolving set of accessing clients. Besides the *Confidentiality* gained through the encryption of the storage, other common security requirements[10] namely *Availability*, *Integrity* and *Accountability* are not satisfactorily considered. *Availability* includes the prohibition of unauthorized modifications or deletions of any data whereas it is only partly considered in current *Cloud Storage Gateway*-approaches. Any secure *Cloud Storage Gateway* must offer fault tolerance data handling like roll-back-operations of any undesirable data modifications.

Most common cloud storages use only web services as interfaces restricting any error handling. Error handling is however mandatory to achieve *Integrity* within a system. *Integrity* is thereby divided into *Data Integrity* and *System Integrity*. *System Integrity* requires constant consistency of the data against any malfunctions regarding the architecture. This requirement can easily be satisfied within hash-based checks of the transferred and stored data. A direct usage of any cloud storage via web services inhibits this requirement since no consistency checks on the transferred data are performed. Additional to *System Integrity*, *Data Integrity* tracks the consistency of authorization within each modification on the data. User bound signatures satisfy this requirement if bound to a version and the corresponding modifications.

Extending these goals and thereby the common definition of security, the *NIST Definition about Underlaying Technical Models for Information Technology Security*j[11] adds two additional requirements. *Accountability* describes the ability to trace any changes within the system. Regarding *Cloud Storage Gateways*, versioning or logging of any modifications on the data enable the *Accountability* and thereby support additional security features like non-repudiation.

Assurance, the second requirement defined within the *NIST*-definition, represents the necessity to define an overall workflow to provide security not only within a system but also within the using environment. This requirement covers social and personal aspects out of focus regarding *SecureCG*.

Satisfying all requirements denoted above, *SecureCG* consists out of the components shown in Fig. 1: The server performs computations on the data wherefore direct storages within the cloud like the *Amazon S3* are not used. Due to the variety of services in the cloud[8], the storage of the data instead is represented by a server instance deployed as *PaaS* (e.g. *Google AppEngine*, *Amazon Beanstalk*, *Windows Azure*, ...). The inevitable non-confidential data handling upon *Cloud Service Providers* requires the stored data to be encrypted. Besides storing the encrypted data, the server furthermore propagates changes regarding encryption keys even though the server is not aware about the keys. Section II-C describes the server component including consistency checks upon encrypted data and propagation of new key material.



(a) Recursive Hash including local encoding

(b) Page Layer with two versions

Fig. 2: *Treetank* satisfying *Integrity* and *Accountability*

The purpose of any client instance is the providing of user centric interfaces (e.g. web services or iSCSI) to access the server within *SecureCG*. Even if the client can have flexible representations fitting the requirements of the use cases, it must be deployed in a trusted environment. The trusted environment of the client maps the requirement of *Confidentiality* within the system. The clients additionally sign the modifications as well to provide *Accountability* and *Integrity*. The functionalities within the client are described in section II-A.

To improve the key handling regarding distributed clients and fine-granular access management, the key management is performed within an externalized *Key Manager*. The *Key Manager* organizes key material based upon *VersaKey*[12] for de-/and encryption as well as for signature issues. Since the key propagation within the proposed architecture is performed by the server, the *Key Manager* can be shutdown while no updates on the key management is performed. A detailed description about the role of the *Key Manager* is given in section II-B.

The synchronization of key material results in push messages based on *REST* between the *Key Manager* and the server. The necessity to use sessions motivates the usage of *SOAP* for transferring data between the client and the server since the transfer of primary data needs a preceding authorization workflow. The concrete workflows including the different techniques for data transfer are described in section II-D.

A. Outline of the Client

The client within *SecureCG* pursues the same target like common *Cloud Storage Gateways*: Different interfaces mapping the requirements of accessing tools are provided within the client. The ability to wrap any content for secure storage in the cloud is archived by using XML as common storage format. As the de-facto "lingua franca" in *WWW*-environments, most content is not only available as XML, additionally XML has the ability to offer flexible representations in a structured way. Interfaces to web services as well as iSCSI wrap any alien data into XML as denoted in Fig 1. The client relies, for handling XML in a scalable way, on the own implemented native XML database *Treetank* [3]. *Treetank* offers different features on node level (including versioning) based upon a layered architecture: Node based operations are provided within the node layer supported by a local node encoding. Independently of the position of nodes within the tree, all nodes are stored in pages representing the page layer. The page layer offers versioning inspired by *ZFS*[1] and encrypts any data based on the local keys. The de-/serialization process of the pages is independent from the concrete storage backends which include at the moment plain files and the *BerkeleyDB*. The data is within all storages only appended so no data is deleted based upon the versioning of the page layer.

The node based operations of the node layer, enabled by the local node encoding, include structural integrity checks. Based upon a recursive hash, each node guards the integrity of the corresponding subtree. Figure 2a shows an evolving tree within the insertion of node "67". The green node is inserted while the yellow ones are updated with the help of reading the blue nodes. The arrows between the nodes represent the pointers stored to each node within *Treetank*. The insertion of a node updates all nodes on the *ancestor-or-self*-axis while all siblings on this axis must be read to recompute the new hashes. This functionality can be used not only to provide consistency within the storage but also while the data is in process e.g. regarding consecutive *REST* requests[5]. Equipped with an incremental hash function[2], the ancestor-sibling traversal becomes redundant which improves the performance of recomputing the hashes.

Next steps include the improvement of this approach by using cryptographic hashes since the incremental hash function is not aware of isomorphic tree structures. Using cryptographic hashes, the recursive integrity check guards the *System Integrity* within storage and transfer of any data. Extending the page layer with additional checksumming of pages similar to *ZFS* supports the *System Integrity* within the storage and the transfer. Regarding the necessity to provide *Data Integrity*, the hash structures will be extended to sponsor user bound signatures. The data itself will contain thereby the fingerprint of the last authorized user which modified the node and the version.

Security includes non-repudiation whereas *Integrity* represents a necessary requirement. A second requirement to gain non-repudiation is the *Accountability* to track changes within the system. *Treetank* supports, based on the page layer,

versioning functionality. Figure 2b shows an evolving page layer within two revisions. All modified nodes are stored in new node pages, denoted by the blue color, created under the related, red colored revision-root-page representing a new version. The indirect pages, colored yellow, multiply the fanout of the overall root (the uber-page) and the single revision-root-pages. Tag names are stored separately in name pages, colored green within Fig. 2b. All new versions result in new subtrees of pages including the modified nodes. Different versioning algorithms are supported within this hierarchy satisfying the security requirement of *Accountability*. Access is based upon transactions enabling atomic modifications on multiple nodes within one revision.

Further work to improve *Accountability* includes the research on adaptive versioning. Whereas incremental versioning offers the write-optimal way to represent changes between two versions, differential versioning represents the read-optimal approach. Since the read- and write-performance should be balanced over the time, a combined approach consuming more storage but resulting in predictable workloads is desirable.

All versioned data stored in the cloud must be encrypted to ensure *Confidentiality*. This encryption must take place in a trusted environment. Encryption in the "untrusted" cloud would violate the *Confidentiality* since *Cloud Service Providers* would have the ability to copy and store the unencrypted data. *Treetank* offers the ability to encrypt the nodes directly in the page layer. Since the encrypted pages are serialized to the *Cloud Service Provider*-infrastructure, the data in the cloud can only be accessed within an authorized client. The recursive integrity structure upon XML guards thereby the unencrypted data including any signatures whereas the integrity check on the page layer cares about the *System Integrity* based on the encrypted pages. The management of the key material for encryption- and signing-operations is organized within an external *Key Manager*. This supports the access of multiple clients to the same storage as described in section II-B.

The current encryption approach will be extended by a fine-granular access control mechanism respecting the recursive relationships within the tree structure. Based upon the hierarchy, access control on higher level nodes includes automatically the corresponding subtrees. Therefore, access control with different levels of ability must be introduced. A

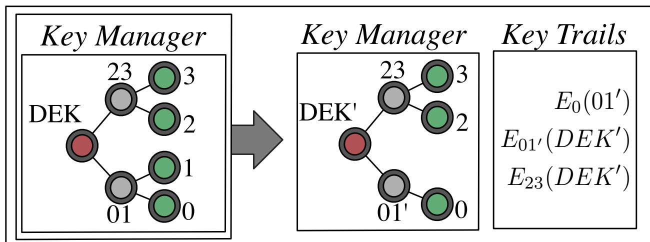


Fig. 3: Encryption tree within the *Key Manager* and related *Key Trails*

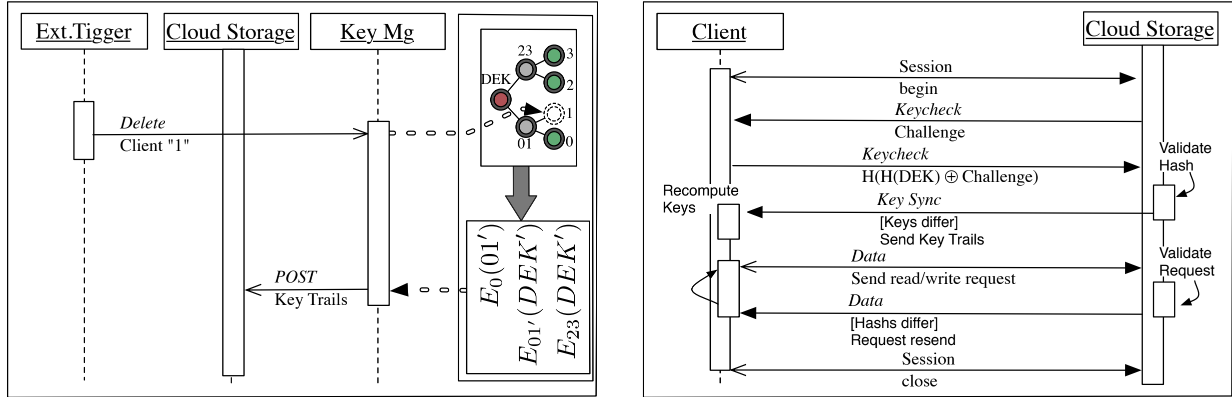
further extension includes the awareness of former versions within changing keys. Older versions should stay accessible within all valid clients at this given time.

Using XML as storage format generates synergies due to the underlying tree structure. The tree structure supports recursive consistency-checks and encryption ensuring *Integrity* and *Confidentiality*. Since *Treetank* consists out of flexible layers supporting versioning and different backends, interfaces to the storage must only wrap XML around any content. Available interfaces include *REST* [4] whereas extensions will cover even block based communications relying on *jSCSI* [7]. *Treetank* offers fine-granular authorization on subtrees[6] to support *Confidentiality* even on interface level. Available as pure Java implementation, *Treetank* and its extensions offers multiple possibilities of usages e.g. as third party library or deployed as an own web service within a trusted environment

B. *Standalone* Key Manager

If working in a distributed collaborative environment based upon multiple clients, key management becomes a critical issue. Encryption keys must be shared within all clients to access the same data. This influences the security within the system since the keys have to evolve with ongoing modifications within the client set. To support join-/leave-operations of clients, *SecureCG* externalizes the key management and equips all clients with disjunct keys. An established approach to encrypt network traffic called *Versakey*[12] supports the joins and leaves of clients. An hierarchy organizes disjunct keys whereas the leafs represent concrete clients, denoted by the green nodes while the root represents the encryption key called DEK represented by the red nodes in Fig. 3. Using an unique key, each client has the ability to access all data. The purpose of the internal nodes is the combination of the keys in the related subtrees. Any leave or join within the client set results in an adaption of all keys on the path to the root. These changes result in encrypted logs named *Key Trails*. The *Key Trails* represent the new keys encrypted with the valid former keys of the authorized clients. The distinction between keys and *Key Trails* results in a distribution of the functionality regarding the management of the keys on the *Key Manager* and the propagation of changes within the server.

Figure 3 shows an example regarding the key management. Client "1" leaves the architecture consisting out of four clients. The *Key Manager* generates new keys for $01 \rightarrow 01'$ and $DEK \rightarrow DEK'$ where DEK' is the new encryption key. The new keys are encrypted with the old, valid keys e.g. $E_0(01')$ denotes the new key $01'$ encrypted with the key of client "0". The encrypted keys $E_0(01')$, $E_{01'}(DEK')$ and $E_{23}(DEK')$ represent the *Key Trails* and are pushed to the server in order to be propagated to the clients. Afterwards, all valid clients, accessing the storage, are triggered to update their keys and decrypt the new keys with their former ones. To access this data, each client holds not only its own key but also all keys to the path to the DEK. Client "3" has thus the ability to access the new DEK' by decrypting the trail $E_{23}(DEK')$ with its stored key 23.



(a) Adaption of *Key Manager* related to join operations of clients

(b) Writing data including key propagation

Fig. 4: Communication Workflows

With this approach and the distributed functionality of adapting keys and propagating changes, flexible joins and leaves on the client site are possible. The combination of the versioning of the data and the traceability of modifications within the encryption keys based upon the *Key Trails* enables time aware access policies without re-encrypting any data. Additionally, the *Key Trails* increase the *Availability* due to the storage of the encrypted modifications on the server itself.

C. *Storing onto any* Cloud Service Providers

While common *Cloud Storage Gateways* map user required interfaces to common web service calls onto the cloud storage directly, the proposed server component of *SecureCG* consists out of a *PaaS* implementation since it has to extend the storage of the data with additional functionalities. These additional functionalities act as hooks before and after transferring data guarding *Integrity* and *Confidentiality* on the data. Using the hashes of the pages, *System Integrity* is ensured while the data is in transfer or in storage. With each transfer of an encrypted page, the corresponding hash is delivered. Any difference related to the delivered hash and the one computed on the received page results in a request to resend the data. Any page retrieved from the cloud storage by the server is checked against the deposed hash as well to prohibit any unforeseen changes on the data. Before any transfer of data, the keys are synchronized. In the case of outdated keys, the server propagates the related *Key Trails* to the client. This results in an on-demand service of the *Key Manager* increasing security and *Availability* due to the highly available and encrypted storage of the *Key Trails* on the server.

D. *Communication between Server, Client and* Key Manager

Communication within the proposed architecture takes place over common web services. *SecureCG* uses in this context three different kinds of communication workflows:

1) The first communication workflow is triggered because of changes on the set of clients and includes, besides an external trigger, the server and the *Key Manager*.
2) The second communication workflow represents the propagation of new key material from the server to the client.
3) The last communication workflow denotes the transfer of primary data between client and server.

The first communication workflow is based upon *REST* requests shown in Fig. 4a. An external trigger starts the key changing workflow based on a modifying *REST* request on the authorized client key. Note that the encryption keys are not retrievable over *GET* as resource. Even though identifiable as substructures, the only valid operations on the keys are *DELETE*- or *PUT*-operations on authorized client keys. A deletion request of client “1” results thereby in a new DEK' and related *Key Trails*. The *Key Trails* are afterwards posted to the server to become available for all clients. The only invariant the *Key Manager* holds within this transmission workflow is the order: Each request coming in is handled in an atomic manner resulting in the generation of the *Key Trails* followed by the push of the *Key Trails* to the server instance. The parts communicating with each other within this workflow are identifiable as blue components in Fig. 1.

The second and the third communication workflow are combined as denoted in Fig. 4b. This combination results in a session-based order of requests. An authorization workflow checks if the DEK on the client is up-to-date before any transmission of primary data starts. At the begin of the authorization workflow, the server sends a challenge to the client. The client performs a concatenation of the challenge and the hash of its DEK, $H(DEK)$, and sends back the result to the server.

Since the server also holds the actual $H(DEK)$, the server reproduces the concatenation and compares both results. The server only stores $H(DEK)$ and not the DEK itself since this would violate the *Confidentiality*.

If the hash differs, the *Key Trails* are send to the client to be decrypted. The decryption operations suitable to the own keys generate the actual DEK (if the client was not excluded

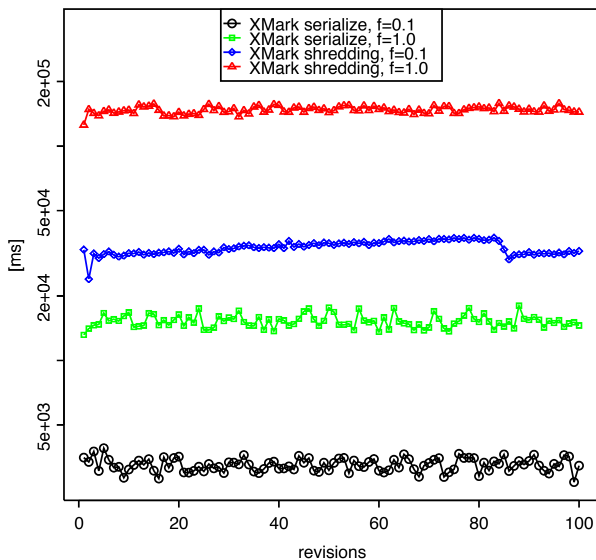


Fig. 5: XMark Shredding within different versions

from the architecture). After the optional synchronization of the DEK, the client requests the primary data. The requests are thereby checked against the hashes of the transferred pages to prohibit transmission errors. If the hashes differ, a resend request is returned. Despite the first communication workflow using *REST*, any request from the clients to the server rely on *SOAP* due to the prefixed authorization workflow. The components of this communication workflow are colored red in Fig. 1.

III. Conclusion and Results

The proposed architecture combines common techniques from the area of XML databases, web services and security to provide an useable framework for secure cloud storage. The resulting framework will be extensible regarding interfaces to the client and flexible with focus on different possibilities of appliance. Furthermore, the resulting approaches are applied directly into an open-source reference implementation proving the suitability as well as the scalability.

Fig. 5 shows the shredding of two different XMark[9]-instances into *Treetank* ($f = 0.1$,$f = 1.0$). Within each revision the tree is deleted before inserting a new instance. Inserting and retrieving data related to any revision takes approximately the same amount of time since *Treetank* scales due to its page layer. Based upon the page layer, the position of nodes in the tree are independent from their position within the storage and therefore within the revision. This independence ensures the scalability within the proposed architecture.

The *Treetank*-based client and the *PaaS*-based server ensure *Confidentiality*, *Accountability*, *Integrity* and *Availability*. Additionally, the independent *Key Manager* supports individual keys between disjunct clients. Already partially released as open source, the proposed architecture consisting out of a combination of complementary approaches offers great benefits for the secure usage of *Cloud Based Services* within a collaborative environment.

IV. Final Steps

Even if encryption is already implemented within the client, the access control needs further development to offer fine-granular, recursive-aware access on any subtree. This access control must be adapted within *Versakey* to fit the versioning approach to ensure access on older revisions for former clients. Besides the encryption which respects the versioning, the versioning itself must be adapted to balance between incremental- and differential versioning. This adaption, which might result in intermediate full-dumps of the entire data, must respect the encryption by extending the client based key handling with additional versioning functionality. The current consistency check are either performant or not aware of isomorphic subtrees. A combination of both approaches will satisfy the requirements of a cryptographic hash without consuming as much time as the current approach. To ensure *System Integrity* even in the cloud, a second hash structure will be introduced to guard the encrypted pages. Both hashes are combined with user bound signatures to combine *System Integrity* and *Data Integrity* on the XML within the structure (node layer) and the encrypted storage (page layer).

V. Acknowledgements

I would like to thank my supervisor, Marcel Waldvogel, for his guidance. Furthermore I would like to thank Anna Dowden-Williams for her more than valuable input.

References

[1] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum, "The zettabyte file system," in *FAST 2003: 2nd Usenix Conference on File and Storage Technologies*.
[2] S. Graf, S. K. Belle, and M. Waldvogel, "Rolling boles, optimal XML structure integrity for updating operations," in *Proceedings of the 20th international conference on World wide web*, ser. WWW '11. New York, NY, USA: ACM, 2011.
[3] S. Graf, M. Kramis, and M. Waldvogel, "Treetank: Designing a versioned XML storage," in *XMLPrague'11*, 2011.
[4] S. Graf, L. Lewandowski, and C. Grün, "JAX-RX, unified REST access to XML resources," University of Konstanz, Tech. Rep., 2010.
[5] S. Graf, L. Lewandowski, and M. Waldvogel, "Integrity assurance for RESTful XML," in *Proceedings of the 2010 international conference on Advances in conceptual modeling: applications and challenges*, ser. ER'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 180–189.
[6] S. Graf, V. Zholudev, L. Lewandowski, and M. Waldvogel, "Hecate, managing authorization with restful xml," in *Proceedings of the 2nd Workshop on RESTful Services*, ser. WS-REST '11, 2011.
[7] M. Kramis, V. Wildi, B. Lemke, S. Graf, H. Janetzko, and M. Waldvogel, "jscsi - a java iscsi initiator," in *Paper for: Jazoon'07 - Internationale Konferenz für Java-Technologie*. Universitt Konstanz, 2007.
[8] P. Mell and T. Grance, "The nist definition of cloud computing," *National Institute of Standards and Technology*, vol. 53, no. 6, 2009.
[9] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "Xmark: A benchmark for xml data management," in *International Conference on Very Large Data Bases*, 2002.
[10] B. Schneier, *Secrets and lies: digital security in a networked world*. John Wiley, 2000.
[11] G. Stoneburner, "Underlaying technical models for information technology security," *National Institute of Standards and Technology*, 2001.
[12] M. Waldvogel, G. Caronni, D. Sun, N. Weiler, and B. Plattner, "The VersaKey framework: Versatile group key management," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 9, pp. 1614–1631, Sep. 1999.
[13] E. Wilde, "Putting things to REST," Tech. Rep., 2007.