Scalable Key Management for Distributed Cloud Storage

Mathias Björkqvist
IBM Research - Zurich
Rüschlikon, Switzerland
mbj@zurich.ibm.com

Christian Cachin
IBM Research - Zurich
Rüschlikon, Switzerland
cca@zurich.ibm.com

Felix Engelmann
Ulm University
Ulm, Germany
felix.engelmann@uni-ulm.de

Alessandro Sorniotti
IBM Research - Zurich
Rüschlikon, Switzerland
aso@zurich.ibm.com

***Abstract*—As use of cryptography increases in all areas of computing, efficient solutions for key management in distributed systems are needed. Large deployments in the cloud can require millions of keys for thousands of clients. The current approaches for serving keys are centralized components, which do not scale as desired.**

This work reports on the realization of a key manager that uses an untrusted distributed key-value store (KVS) and offers consistent key distribution over the Key-Management Interoperability Protocol (KMIP). To achieve confidentiality, it uses a key hierarchy where every key except a root key itself is encrypted by the respective parent key. The hierarchy also allows for key rotation and, ultimately, for secure deletion of data. The design permits key rotation to proceed concurrently with key-serving operations.

A prototype was integrated with IBM Spectrum Scale, a highly scalable cluster file system, where it serves keys for file encryption. Linear scalability was achieved even under load from concurrent key updates. The implementation shows that the approach is viable, works as intended, and suitable for high-throughput key serving in cloud platforms.

I. Introduction

Encryption plays a fundamental role for realizing secure networked computing environments. Key management ensures reliable and secure distribution of cryptographic keys to legitimate clients, which are then able to encrypt data or to establish secure communication channels. Key management for cloud-scale distributed installations poses additional challenges over classical, centralized systems, due to the vastly bigger systems and the higher demands for resilience and security.

Maintaining the confidentiality of encryption keys is extremely important, especially for encrypting data in storage systems, where losing access to the encryption key implies losing the data itself. A communication system, in contrast, may just restart the session if a key is lost.

As key management is critical for many environments, industry standards have been introduced to separate key-management functions from the components that consume keys, and to consolidate key lifecycle management at centralized, well-protected systems [1]. Key management can be seen as an essential service of an IT infrastructure and especially for cloud platforms, similar to network connectivity, computing, and storage. The most prominent standard for distributed key management today is the OASIS Key Management Interoperability Protocol (KMIP) [2], which specifies operations for managing, storing, and retrieving keys at a remote server. For local key management using library-style access PKCS #11 [3] is the prevalent interface. In the context of cloud services, where service interactions are REST calls, the open-source Barbican [4] key manager provides keys to all services of OpenStack. Commercial cloud platforms use proprietary protocols inside their infrastructure.

Key managers differ according to the operations they support and in terms of their performance, resilience, and security. Prominent commercial key servers often put emphasis on the needs of enterprise environments, such as fine-grained authentication and support for hardware security modules (HSMs). For example, governmental standards for handling health data dictate a reliable audit trail to reconstruct all operations accessing cryptographic keys. Enterprise key managers are also designed for high availability to allow uninterrupted service. They must support the complete lifecycle of cryptographic secrets, with operations for creating, importing, storing, reading, updating, exporting, and deleting keys.

Designing and operating a key-management service in a distributed system with many entities running cryptographic operations is challenging because it must balance between the conflicting goals of performance and security.

A. Contribution

In this paper we present a solution for scaling a key management service to cloud applications with thousands of clients and possibly millions of keys. This includes the capability to scale dynamically while maintaining security. Our distributed key management solution should handle all core key lifecycle-management tasks and scale in a linear way. Existing enterprise-grade key managers do not provide such scalability because they rely too heavily on centralized components, such as HSMs or strongly consistent relational databases.

In particular, we address key management for an enterprise environment with a scalable cloud platform. The *clients* or *endpoints* accessing the key manager primarily perform data-at-rest encryption, but could be any other application that executes cryptographic operations. The key server has to provide a consistent view of the available keys when accessed from multiple clients.

The key manager consists of three components with different security assumptions:

Trusted storage: A *master key* resides on a trusted medium for persistent storage across power cycles. When the system is turned off, no component apart from the trusted storage medium maintains any relevant cryptographic information; in other words, *only* the root of trust must be guarded. For supporting lifecycle management and key rotation, the medium should support secure erasure. Potential implementations are hardware security modules (HSMs), USB-attached smart cards, or also cheap removable USB drives that can be physically destroyed for key erasure. For better scalability, multiple instances of trusted storage may be utilized.

Metadata key-value store: For supporting scalable operation, all cryptographic material apart from the master key is held in an untrusted distributed storage system modeled as a key-value store (KVS). All data stored in the KVS is protected with the master key, i.e., by wrapping or through a key hierarchy. The distributed KVS supports high-throughput and scalable access to the stored key material. The KVS platforms available today often do not support atomic operations but only eventual consistency, however. This can lead to issues caused by simultaneous access to the same key by multiple clients.

Key server: Keys are delivered to the clients in cleartext and are available in the memory of the key server. The key server is trusted not to disclose keys to unauthorized clients. It overwrites keys no longer needed according to standard practice.

The KVS used for scalability does not permit strongly consistent operations on keys, such as key rotation. We resolve this through a novel design for concurrent key-management operations that use a weakly consistent data store.

In particular, we describe an implementation of a key manager using hierarchically protected keys, where only the master key resides on trusted storage. The important features of this design are:

1) The key server permits continuous access to keys in the presence of ongoing key rotation operations;
2) The key server scales linearly with the available server resources.

An implementation for the IBM Spectrum Scale cluster files system (http://www.ibm.com/systems/storage/spectrum/scale/, formerly known as General Parallel File System or GPFS) was developed and evaluated for this purpose. Experiments show that the design and prototype deliver the expected performance, in particular, linear scalability with the server resources.

B. Products and related work

Every cryptographic system must manage keys. Several standalone key managers have been developed to provide a generic service and support the standard protocols, such as KMIP [2]. Key-lifecycle management operations are important for enterprise deployment [1] and for satisfying regulatory requirements [5].

Two prominent products addressing this space are the *Vormetric Data Security Management (DSM)* server [6] and the *IBM Security Key Lifecycle Manager (ISKLM)*. Vormetric DSM offers interoperability with KMIP and facilitates integration with database systems, such as Microsoft SQL Server. It is certified at several FIPS 140-2 levels, depending on the hardware on which it operates. With an appropriate HSM, certification ranges up to FIPS 140-2 Level 3. Due to its centralized architecture it is inherently not scalable. ISKLM runs on standard servers atop a software stack that includes IBM WebSphere and DB2. For key storage, ISKLM may use a PKCS #11-attached HSM, and for resilience and high availability, it can run distributedly on a cluster. It provides a web interface for management, control, and auditing, and a KMIP interface serving keys to endpoints. Due to its dependence on the database, ISKLM is also centralized. Although these and other similar solutions are complex and incur high operational expenses, they do not scale as well for distributed platforms.

Barbican [4] inside OpenStack is a scalable key-management component for cloud platforms. It supports asynchronous operations on symmetric and private keys, public keys, and certificates. As all services in OpenStack, it is accessed through a REST API and uses the *Keystone* [7] component for authorization. Its operations are not as general as those of KMIP, for instance. The backend of Barbican is modular, designed as a plug-in system, so that it currently supports an SQL database or a KMIP client for storing secrets remotely, or an HSM accessed through PKCS #11 for local key storage. In Barbican, multiple workers run concurrently and on different hosts, but they all interact with the central backend database through a message queue. This limits its scalability. Considering also its restricted functionality in the API, Barbican doesn't offer the desired scalable support for key storage in combination with key rotation and secure erasure.

Several key management services are available in cloud platforms today, such as Amazon KMS, CloudHSM, and IBM Key Protect [8], [9], [10]; they hide the complexities of in-house key management but rely on partial trust in the cloud operator.

II. Objectives

A. Security goals

In a deployment of the key manager, there are two actors: the key manager and the client. Only the key manager may access the master key and the trusted storage. Any other entity is subsumed into an *adversary*, an actor with complete access to the metadata KVS, communication links, and so on. The adversary might be an untrusted storage provider, the client itself, or the operator of the network between the client and the server.

Clients can authenticate themselves to the key manager and establish a secure connection using TLS, with server- and client-side certificates. The key manager authorizes access to

keys based on the identity in the client certificate and delivers keys to the client over the secure channel. There are two concrete goals addressing security.

Key confidentiality: The adversary should not be able to learn any useful information about a key to which it does not have access, i.e., keys for which it is not authorized, neither by interacting with the server, by observing the KVS, nor by listening on the network.

Key erasure: Clients can request that a key is deleted permanently. This supports the *secure deletion* of data protected by this key, since physical deletion of stored data from disks or solid-state storage is no longer possible in practice [11]. Key deletion supports the approach to secure deletion introduced by Di Crescenzo et al. [12] and recently extended by Cachin et al. [13], where only a single key at the root of a hierarchy must actually be erasable. All other metadata in the hierarchy can be exposed to the adversary. From the perspective of the key manager, secure deletion is supported through *key rotation* of the master key and *deletion* of expired keys.

B. Other goals

Scalability: The service performance should scale linearly with the available resources and not disrupt client operations. To achieve this, the key manager is implemented by parallel stateless workers that have access to the master key. The key server processes have to operate in a consistent way, to serve the current version of a key consistent with the data protected by it. It should also be possible to dynamically adapt the number of servers to the load of the system.

Availability: Associated with scalability, the service must also be resilient to failures of individual nodes. Problems with the consistency of different key management servers can result in loss of data, when different clients encrypt with different keys.

Usability: In any security system, the weakest link is the erroneous handling of sensitive material by users or operators (as the reliance on user-memorable passwords shows). The system should support operations in a straightforward and transparent way.

III. Design

A. Architecture

The architecture of the key server is shown in Figure 1. Every node essentially runs the same *key server* independently of the others, but has access to the distributed *metadata KVS* and to the *trusted storage*. Having multiple key servers work in parallel provides *scalability*. The *core server* locally caches the most recently accessed keys in memory.

The KVS for metadata must support the usual *put* and *get* operations on *key/value* pairs, with the additional requirement that it supports *conditional put*. More precisely, the KVS supports

$$get(key) \rightarrow (value, version)$$

$$put(key, value, version) \rightarrow success,$$

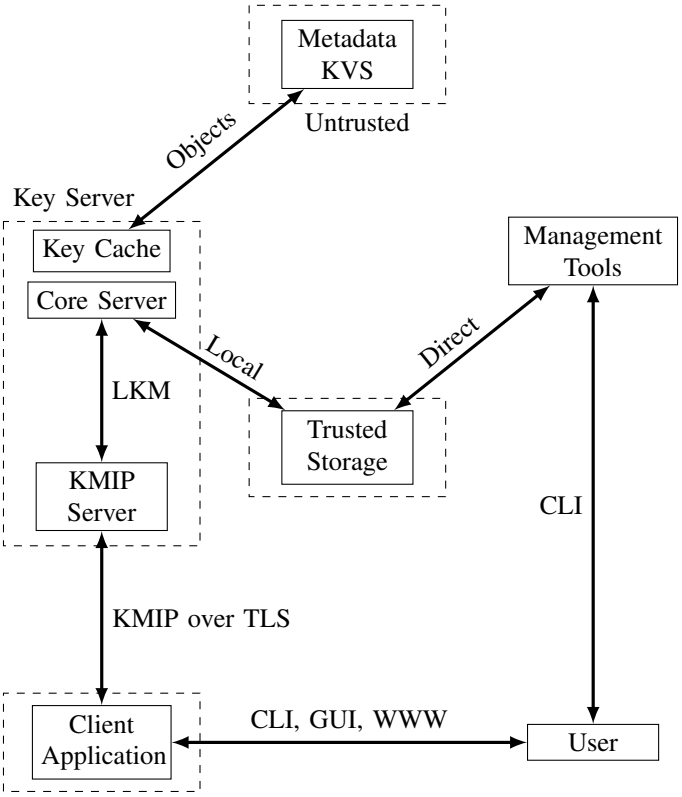

Fig. 1. Overview of the key manager components. Cryptographic material is only stored and handled inside the dashed regions.

where *put* takes effect only when *success* = TRUE. Every *put* operation monotonically increments the version associated to a *value*. The specification requires that *put* returns TRUE if and only if the version of *key* that is replaced by the operation was *version*. Multiple platforms can provide such operations, i.e., *Comet* [14], *redis* (https://redis.io), or Amazon DynamoDB (https://aws.amazon.com/dynamodb/). The prototype that targets file encryption in IBM Spectrum Scale uses the *cluster configuration repository (CCR)* as the distributed KVS implementation, which is a Spectrum-Scale-internal distributed data store with high availability that offers versioned put/get operations.

Clients address keys through a unique, randomly generated *identifier* in the KMIP interface. More user-friendly attributes such as a *name* may be used to associate a key with the function that the key plays in the context of the application, and KMIP permits multiple keys to have the same name in the scope of a KMIP server. The name, along with other attributes, are part of the metadata of the key itself, and these attributes can be used to retrieve the identifier of a key, which is then used for subsequent operations. The communication between the client and the key server is performed using the tag-type-length-value (TTLV) transport of the KMIP protocol, which in turn is communicated over a mutually-authenticated TLS connection.

Cryptographic keys stored in the KVS are protected using a *master key* retrieved from trusted storage. More precisely,

every key in the KVS is *wrapped* with the master key using AES and authenticated encryption with Galois Counter Mode (GCM). For every wrapped key (in the KVS), also the identifier of the wrapping key is stored in the metadata. This is needed for key rotation.

The trusted storage accessed by every node is initialized with the master key. It can be evolved through key rotation. A *user* interacts with the system through *management tools* for the provisioning of the master key in trusted storage.

The core of the key server provides a *local key manager (LKM)* interface towards a *KMIP proxy server*, which interfaces to the clients. This ensures compatibility with many existing endpoint clients in cloud and enterprise storage systems that access a key manager over KMIP. The key management service itself is completely stateless, and can flush its cache at any time.

A client accesses a key over KMIP (via *Create*, *Locate*, *Get*, *Destroy*, *Rekey* ... operations), using the identifier, which is stored e.g., in the metadata of the data unit protected by this key. For a cloud object store, this could be in the metadata of the *account/container/object* of an object as in OpenStack Swift; for a distributed file system, this could be in the *inode* of a file.

Storing keys in wrapped form ensures the *key confidentiality* requirement. No data on persistent storage, apart from the trusted storage, reveals any information about the keys. Decrypted keys only exist in volatile memory. Furthermore using authenticated encryption for wrapping also guards against accidental or intentional alteration of cryptographic keys or their attributes in the KVS.

The goals of *scalability* and *availability* are achieved by the design of stateless key server nodes, which can be placed into different failure zones, and by the use of a highly available KVS for storing the key data. The inherent scalability and availability of the distributed KVS is further enhanced by deploying separate trusted storage on each key server node, thus improving the characteristics of the entire system.

A potential drawback of this design is the overhead related to the management of the trusted storage. The added complexity comes from the fact that the master keys are now managed internally by the system, and not delegated to an external key management service. The overhead varies depending on e.g., the choice of trusted storage media type, the degree of integration of the trusted storage management with the rest of the system, and regulatory compliance requirements.

B. Operations

Cryptographic material is stored in a format that is a simplification of the network-level representation of KMIP. This has the advantage that much of the needed functionality is already available from existing KMIP libraries.

Clients authenticate to the KMIP server during the TLS handshake using a client certificate, following the standard practice that has been available and deployed with many KMIP servers. After authentication, they may exchange data in the KMIP format over the secure channel.

At system initialization time, the core server generates a fresh master key and its identifier, and writes them to the trusted storage over the local interface. Every key is stored in the KVS as a separate data object under its identifier. During operation the core server responds to client operations, arriving through the KMIP interface, as follows.

Create: The key material is generated from a cryptographically strong random source. The key itself is wrapped with the master key; the master-key identifier together with the key identifier are added as attributes. The data is serialized in the KMIP-derived format and stored as an object in the KVS, according to its key identifier. Key and key identifier are returned to the client.

Get: For retrieving keys, the server uses the identifier provided in the request by the client to locate the key. To obtain the key material from the KVS, the server retrieves the object containing the wrapped key, unwraps the key material with the master key specified in the attributes, and verifies the integrity of the unwrapped data. Then it returns key and key identifier to the client.

Referencing a key with the unique identifier instead of the name supports the implementation of client-side key rotation; note that the name may be the same across different versions.

The key server additionally maintains a cache with the recently served keys in cleartext. The server only has to query the KVS when it has no cached copy of the key or when a cached key reaches its refresh age.

Destroy: The server deletes the corresponding object with the wrapped key in the KVS. Note this does not yet securely erase the key according to the security model, as the KVS may still keep a copy of the object and the master key has not yet changed. This is addressed below.

Rekey: The KMIP *Rekey* operation for a given key creates a fresh key with the same name and other attributes as the existing key. It is implemented by creating a new key and destroying the old one, and by removing the name from the old key. The returned key material is fresh and has a new identifier. The old key may still be accessible until the master key is changed as well.

Secure deletion: The key server provides secure deletion in the sense that the master key is *rotated*. Any key material that was stored on the untrusted KVS wrapped with the previous master key then becomes inaccessible. To support this, master keys contain a *version* in their attributes. The challenge lies in rotating the master key without disrupting the other operations of the key server.

For the moment, assume that the rotation is triggered by a dedicated *agent*, which may be a special management node or an administrator client. For rotating one particular key in a hierarchy, we call the keys that it wraps *children* (i.e., a client-visible key) and the wrapping key the *parent* (i.e., master key).

To rotate the parent (i.e., master) key, fresh key material with a higher version is first chosen and written to trusted storage. From this moment on, any operation on the parent key that does not specify a version or identifier returns the new key. On the other hand, for any children wrapped with

the old key, the appropriate version can be retrieved with the key identifier available in the metadata of the wrapping. This ensures continuous operation while rotation is in progress, in particular, key creation, retrieval, and destruction operations can be served by other key-manager nodes. The agent now cycles over all children of the parent, unwraps the child key, rewraps it with the new parent key, and stores the outcome in the corresponding object. When this is complete, the old parent key is securely erased from the trusted storage.

This design can be generalized to key-wrapping hierarchies of depth larger than one in a future extension. When the key-rotation agent invokes a rotation operation for a key, the rotation recursively trickles down to all its children. Key rotation would then progress over the tree of children, until they are all wrapped with the new key.

The only limitation regarding concurrent operation is that the agent performing key rotation must be unique, as otherwise, two rotation operations might occur concurrently and leave the data in the KVS in an inconsistent state. With an arbitrary, weakly consistent KVS, one can implement this by partitioning the key space across the agents or alternatively by using an external locking mechanism. With a versioned KVS, however, the KVS-level objects can be replaced conditionally on the old version containing the previous wrapping key identifier. This ensures that each key is rotated atomically, with the same parent key as for all of its siblings.

The operations for rotating the master key ensure the security goal of *key erasure*.

IV. Evaluation

To evaluate the key manager we have developed a prototype and benchmarked it using the IBM Spectrum Scale distributed cluster file system, formerly known as General Parallel File System (GPFS). Spectrum Scale offers encryption at the level of files, and some of the authors have already been involved in the design and realization of the encryption feature (available since GPFS V4.1). Each node in the cluster accesses the encryption keys individually as a KMIP client; typically keys are served using the IBM Security Key Lifecycle Manager (ISKLM).

The benchmark was performed in a realistic environment with a Spectrum Scale cluster (version 4.1) on two physical servers. Each server was equipped with dual Intel Xeon E5630 processors and 40GB memory, running RedHat enterprise Linux version 7. A Spectrum Scale cluster with a varying number of clients was created on the physical servers. For every file system node, a dedicated key manager node runs on the same physical server. In production deployment we envisage that every file-system/key-manager node pair is co-located on each physical server; this enhances the reliability as it spreads key management across different failure zones.

For the untrusted data store, we used the *cluster configuration repository (CCR)* integrated with Spectrum Scale; it offers a versioned KVS interface (in fact, its implementation supports even stronger consistency). We created a file system in the cluster with a policy to encrypt new files. Each Spectrum Scale client was configured with the KMIP URL of its dedicated key server, including a fail-over address at another key-server node. The file system policy triggers the encryption process whenever a new file is created. The default cache expiration time was set to 15 min (but is irrelevant for the evaluation).

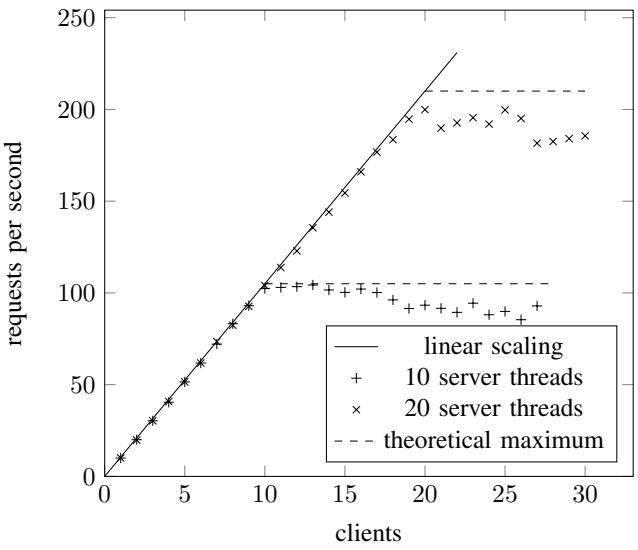

Fig. 2. Scalability of the service with client-side load balancing, showing linear increase.

A. Scalability

A reasonable measure for the performance of the key manager is *throughput*, measured as the number of keys served per second. An artificial capacity limit was imposed by restricting the size of the server thread pool that handles concurrent client accesses.

We queried one key from a single key server with an increasing number of clients running natively. Figure 2 shows that performance scales linearly when the system operates below its capacity limit (all measurements were taken over an average of 10 seconds). By adapting the thread pool size, we can show that the service scales as expected. The scaling is independent of the distribution of threads over the servers.

In Figure 3 the independence of the threads from physical servers is shown. Running six threads on one server yields almost the same result as dividing them evenly among two machines. In the case with two servers, the clients randomly chose a server for each key. With the increasing number of concurrent clients, we observe a degradation of the performance as clients fight for resources. To resolve such a situation, additional nodes would be added to the cluster. Spectrum Scale clients configured in high-availability mode and unable to reach a key manager would then automatically fail over to other key-manager nodes.

B. Latency

During normal file-system operation the key server is rarely exposed to high load. In this sense the throughput is more

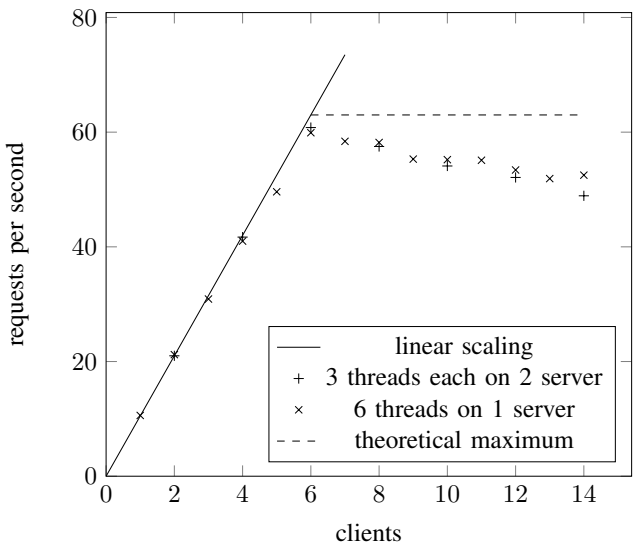

Fig. 3. Scalability of the key manager with client-side load balancing, showing linear increase until reaching the capacity bound.

relevant for administrative tasks, such as booting the cluster or mounting a file system. For the perceived performance the *latency* until a requested key is available matters much more. For instance, this cost occurs whenever an application opens a new file, and the key is not cached.

To reliably measure the latency of the key retrieval, we captured packet traces on the querying machine. In these traces we can measure the total delay between the TCP handshake packets, the KMIP response, and TLS-session tear-down. The total latency is composed of two parts. First, a noticeable fraction of time is spent to create the TLS channel. The second part corresponds to the key manager's operation. We only describe results for key-manager operations, as the TLS contribution is independent of it.

The client requests a key by its identifier. We measured the time between the request and the response packet, which depends on whether the key server has cached the key. Fetching the key from the distributed KVS obviously comes at a cost and results in about 90ms average latency. The timings are shown in the following table, including the standard ISKLM key server product running on the same hardware for comparison:

Measurement	Mean [ms]	Stddev [ms]
Get key from KVS	90.06	$\pm$0.76
Get key from key-server cache	0.30	$\pm$0.05
Get key from ISKLM	200	$--$

C. Concurrency and high availability

A second set of experiments was performed, where another client concurrently stored new keys in the key server every 100 ms (which involves a write-through to the KVS). The experiments of Section IV-A and IV-B, where clients only retrieve keys, were executed. We saw no impact on the retrieval rate or latency for the other client(s).

For testing high availability scenario, we performed a synthetic test from a client stub to retrieve keys, using the same native code as for key retrieval by the Spectrum Scale file system. The parameters were set to time out after 20 seconds and to retry once before selecting the next server. We queried a key repeatedly and then shut down the primary key server while monitoring the network traffic. The resulting behavior matched the expected behavior.

V. Conclusion

As encryption of data at rest becomes more prevalent, the challenge of managing the encryption keys also surfaces for diverse systems. The scalable key-management design presented in this work targets cloud-scale deployments. It is compatible with storing the master keys in an HSM, but achieves better performance than a solution exclusively relying on a centralized key manager or a HSM.

The key manager is built on top of an untrusted key-value store (KVS) and demonstrated in the context of the IBM Spectrum Scale cluster file system. It serves file-encryption keys using the KMIP standard. A key-hierarchy and key rotation operations supporting secure deletion of critical data have been described and prototyped.

The evaluation shows that the key manager was able to scale linearly even under load from key updates, and performance measurements conducted on the individual components indicate that the throughput and latency are mostly limited by the performance of the distributed KVS.

Acknowledgments

This work has been supported in part by the European Commission through the Horizon 2020 Framework Programme (H2020-ICT-2014-1) under grant agreement number 644579 ESCUDO-CLOUD and in part by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contracts number 15.0087.

References

[1] M. Björkqvist, C. Cachin, R. Haas, X. Hu, A. Kurmus, R. Pawlitzek, and M. Vukolić, "Design and implementation of a key-lifecycle management system," in *Proc. Financial Cryptography and Data Security (FC 2010)*, ser. Lecture Notes in Computer Science, R. Sion, Ed., vol. 6052. Springer, 2010, pp. 160–174.

[2] OASIS Key Management Interoperability Protocol Technical Committee, "Key Management Interoperability Protocol Version 1.2," 2015, oASIS Standard, available from http://www.oasis-open.org/committees/documents.php?wg_abbrev=kmip.

[3] OASIS PKCS 11 Technical Committee, "PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 2.40," 2016, oASIS Standard, available from https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=pkcs11.

[4] "OpenStack Barbican," https://wiki.openstack.org/wiki/Barbican, 2018.

[5] E. Barker, "Recommendation for key management — Part 1: General," National Institute of Standards and Technology (NIST), NIST Special Publication 800-57 Part 1 Revision 4, 2016, available from http://csrc.nist.gov/publications/PubsSPs.html.

[6] "Vormetric Data Security Management," https://www.thalesesecurity.com/products/data-encryption/vormetric-data-security-manager, 2018.

[7] "OpenStack Keystone," https://wiki.openstack.org/wiki/Keystone, 2018.
[8] "Amazon CloudHSM," https://aws.amazon.com/cloudhsm/, 2018.
[9] "Amazon Key Management Service," https://aws.amazon.com/kms/, 2018.
[10] "IBM Key Protect," https://console.ng.bluemix.net/catalog/services/key-protect/, 2018.
[11] J. Reardon, S. Capkun, and D. Basin, "SoK: Secure data deletion," in *Proc. 34th IEEE Symposium on Security & Privacy*, 2013.
[12] G. Di Crescenzo, N. Ferguson, R. Impagliazzo, and M. Jakobsson, "How to forget a secret," in *Proc. 16th Symposium on Theoretical Aspects of Computer Science (STACS)*, ser. Lecture Notes in Computer Science, C. Meinel and S. Tison, Eds., vol. 1563. Springer, 1999, pp. 500–509.
[13] C. Cachin, K. Haralambiev, H. Hsiao, and A. Sorniotti, "Policy-based secure deletion," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 259–270. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516690
[14] R. Geambasu, A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy, "Comet: An active distributed key-value store," in *Proc. 9th Symp. Operating Systems Design and Implementation (OSDI)*, 2010.