Managing Personalized Cross-cloud Storage Systems with Meta-code

Magnus Stenhaug, Håvard D. Johansen and Dag Johansen
UIT The Arctic University of Norway, Norway

Keywords: Privacy, Cloud Computing, Federated Cloud, Autonomic Data Management, Hybrid Cloud, Multi-cloud, Data Policies, Security, Edge Computing.

Abstract: Providing fine-level and customized storage solutions for novice cloud users is challenging. At best, a limited set of customization options are provided, often related to volume of data to be stored. We are proposing a radical different customization approach for cloud users, one where personalization of services provided is transparently managed and supported. Our approach to building personalized cloud storage services is to allow the user to specify data management policies that execute in a user space container transparently to the user. In this paper we describe Balva, a cloud management system that allows users to configure flexible management policies on their data. To support legacy applications, Balva is implemented at the file system level, intercepting system calls to effectuate dynamic and personalized management policies attached to files.

1 INTRODUCTION

The convergence and application of IoT sensors, wireless, mobile, and cloud computing enables new and disruptive ways to quantify information on individuals and our environment. Small, wearable, non-invasive, and sensing *edge devices* are becoming household items, with small manufacturing cost and a vast array of different uses. When coupled with advanced machine learning techniques and an abundance of cheap computational infrastructures in the cloud, the data streams generated by these edge devices have the potential to generate technological innovation at a scale never witnessed before (Schwab 2016).

However building storage systems that reliably and over time can capture, store, and curate data from a heterogeneous and dynamic pool of cloud-connected edge devices is not trivial. Examples include solipsistic lifelogging (Gurrin, Smeaton, and Doherty 2014), athlete quantification systems (Johansen et al. 2013), or medical research databases (Stenhaug, Johansen, and Johansen 2016). Requirements for management policies, security policies, and access interfaces are likely to change over time and cannot be all predicted at system design time. Systems currently available for such applications provide little opportunity to customize or individualize governing management and security policies, or data interfaces.

This paper describes Balva, a cross-cloud data service that federates input from edge devices into a logical cross-cloud abstraction. Data hosted by Balva is governed by expressive management and security policies, set by the end users themselves. The set of policies can be adapted over time, in concert with the changing needs and requirements of individual users and their applications. The flexibility in Balva stems from small code snippets attached to individual data objects. Each such code snippet provides security and management functionality that can extend or change various meta properties of the data objects it is attached to, including how the data is replicated or how it can be transformed. We therefore refer to such code snippets as *meta-code* (Johansen and Hurley 2011; Johansen et al. 2015). Balva interpositions meta-code execution in the data access path of hybrid multi-cloud file systems (Dobre, Viotti, and Vukolić 2014; Ardagna 2015). Execution is transparent to the applications and does not require alteration of existing data APIs. This enables Balva to adapt policies for legacy applications, an important requirement for long term data storage and curation. The Balva multi-cloud architecture automates resource management across multiple clouds, which may include autonomic solutions for fine-grained security and privacy policy enforcement, data lock-in avoidance, fault-tolerance, scalability, and performance optimization. We design Balva to capture these rapid changes while still being able to preserve and curate this data for the long term.

Stenhaug, M., Johansen, H. and Johansen, D.
Managing Personalized Cross-cloud Storage Systems with Meta-code.
DOI: 10.5220/0006377607220728
In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, pages 694-700
ISBN: 978-989-758-243-1

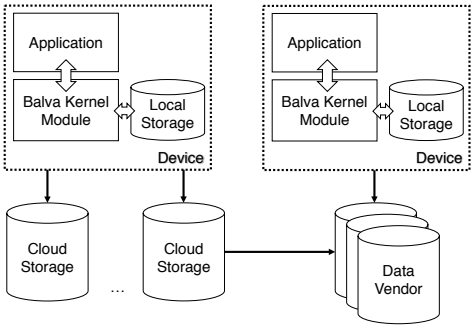

Figure 1: Multi-cloud deployment with Balva.

2 THE BALVA ARCHITECTURE

Recent trends in cloud computing include federating and orchestrating cloud services spanning multiple cloud services and providers. By not committing to a single cloud provider, but rather a collection of service providers, application developers can potentially avoid data lock-in and stay available if the cloud service becomes unavailable. The trade-off is (often) an increase in cost. This cost can be used to the benefit of the programmer as the service spanning multiple providers can greedily choose the cheapest solution. SpanStore (Wu et al. 2013) builds a storage service spanning multiple cloud providers and optimizes for low latency at a modest increase in cost. Similarly, Supercloud (Jia et al. 2016) uses the Virtual Machine spot market to choose the cheapest available computing resource at any time, migrating virtual machines when cheaper resources become available.

Another benefit of federating cloud services is that companies can utilize in-house computing resources while expanding into the cloud when in-house capacity is reached. While dynamically moving the execution and storage of an in-house resource is compelling, it increases the complexity of managing and developing such services. Typically, a federation layer implementing an IaaS cloud platform is added to the software stack to alleviate some of the programming effort. This comes at the cost of having to rely on a set of middleware services and APIs defined by the provider of the federation software, with the potential risk of relying on a software provider that may stop supporting the software.

2.1 Object Model

Balva operates on objects consisting of a data layer and a meta layer. We have designed Balva to impose few restriction on what objects it can handle, their granularity, and how they are stored and processed. An object may be stored as a file, database record, or any sequence of bytes that can be interpreted by applications accessing that object. Balva does, however, require that each object o has a unique identifier $o.id$, for example when stored as a file: an UUID or a host and path name combination. The identifier does not have to be static, and may change over time. We also require that all Balva storage backends support a meta-layer that can store textual or binary key-value metadata with each object. The object storage backend must always keep meta-data together with the data, even if the object is renamed or transformed by applications. Each object can potentially exist in infinitely many discrete states between its creation and its deletion.

In Balva, the meta-layer of each data object o includes a lists a of named and well-defined meta-code modules $o.metacode = [m_0, m_1, \ldots, m_n]$ that applications and Balva can invoke. Invocation of a meta-code module may include parameters and may return some value to the caller. Meta-code modules are executed by the Balva runtime in the isolated context of the object they are attached to, and may manipulate contained data and metadata. As such, the meta-code functions in $o.metacode$ constitute o's public and private interfaces. Balva supports objects with different and dynamic $o.metacode$ members, adapted to how each object will be used. Each meta-code module $m \in o.metacode$ is set by some principal $m.u$, identified by the hash of u's public key. Whenever the meta-code m is executed, it will run with the privileges of its governing principal $m.u$ and isolated within the execution context of o.

We take a minimalist approach to federating cloud storage resources. Balva requires only a few meta-code modules on objects for its own operations, depending on what particular backend objects are stored in or its indented usage. For instance, in our file system implementation of the Balva architecture, as will be described below, the meta-code array includes modules for read, write, and delete operations. In addition, Balva provides meta-code modules for manipulating an object's meta layer, including adding and removing elements of the $o.metacode$ array. We adapt a programming model where users and their applications can extend, replace, or remove elements of $o.metacode$ over time, avoiding the need to define in advance a static set of data-access operations for all cloud backend systems that are to be supported. The ability to extend objects with operations implementing various management policies enables users to build a powerful and personalized storage system tailored for their individual needs.

2.2 Access Tokens

Balva is designed to be deployed across multiple cloud computers or even hybrid and multi-clouds spanning different vendors, as illustrated in Figure 1. Balva therefore may run in multiple administrative domains: from trusted cloud vendors to individual user devices and community owned IoT devices. A program executing locally on a client device must still be able to transparently access protected data through the local file system or a shared library. With traditional Access-Control Lists (ACLs), users would need to maintain accounts with each individual federated service, and have little opportunity to delegate their rights beyond sharing their passwords. Bearer tokens are seemingly better suited for federated cloud operations, as individual services do not need to keep local state on access rights. In particular, Balva grants access to invoke meta-code operations based on a mechanism similar to the Codecaps as described by Renesse et al. (2013).

Balva access tokens are signed sequences meta-code predicates, or heritages, $h_n = [p_0, p_1, \ldots, p_n]$ that are evaluated in the context of an object o whenever $m \in o.metacode$ is invoked. Given a heritage h, a meta-code operation $m \in o.metacode$, Balva only admits an invocation if

$$\forall p \in h \mid eval(p, m, o) = true$$

Thus, having some heritage h_n, a principal can create a derived heritage h_{n+1} by appending new predicates:

$$h_{n+1} = h_n | p_{n+1}$$

Doing so attenuates the rights granted by h_n with additional restrictions imposed by p_{n+1}. Because heritages are cryptographically sealed so that h_n cannot be retrieved from h_{n+1}, principals can safely share subset of their rights with others. Each heritage have a root predicate p_0 that only returns true for some objects. By convention, a principal holding a heritage h such that $eval(h[0], m, o) = true$ is said to be the owner of o, and is typically granted full access.

3 IMPLEMENTATION DETAILS

Balva is a cloud storage service designed to allow configuration of advanced management and security meta-code policies on stored objects. Balva targets supporting the wide-range of existing data-driven application stacks available today, and hence meta-code based policy configuration and enforcement must be transparent to existing legacy applications accessing

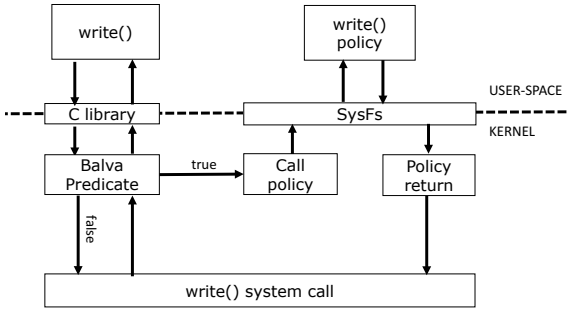

Figure 2: Control flow of write operations.

data. Balva is therefore placed at the Operating System (OS) and file system level within existing Linux-based software stacks. As such, files and blocks are our unit of storage and data access; open, read, and write are our main operations. We assume the underlying file system has the traditional semantic of uniquely identifiable and hierarchical organized file names and the ability to attach metadata to files, a common function in most Linux files systems (e.g., ext-4, the journaling file system for Linux).

Balva meta-code policies are executed based on up-calls from the kernel into a user space daemon, in response to file system calls. As such, the core components of our runtime is a Linux kernel module and a collection of processes that implements file system policies. This enables Balva to take advantage of hooks inserted into the file system, and execute code in the critical path of regular file system operations such as open, read, and write. The Balva kernel module communicates with a user process, which in turn invokes the appropriate callback function. We have implemented hooks for several system calls as required to interposition meta-code in the file system access path. These includes the *sys_open*, *sys_read*, and *sys_write calls*.

The Balva kernel module makes scheduling and configuration decisions on how to execute the meta-code policies associated with the data according to a configuration set by the user. In addition to the kernel module, Balva manages a set of processes running inside containers in user space (Felter et al. 2015; Soltesz et al. 2007). The runtime launches a kernel module that listens for specific system calls, contacting the user space processes if there is a policy set to execute given the system call parameters and the current configuration. From this, we can construct and combine modular mechanisms and policies to implement a personalised data store that can capture specific user needs.

When Balva initializes, the kernel module locates the system call table and replaces specific system

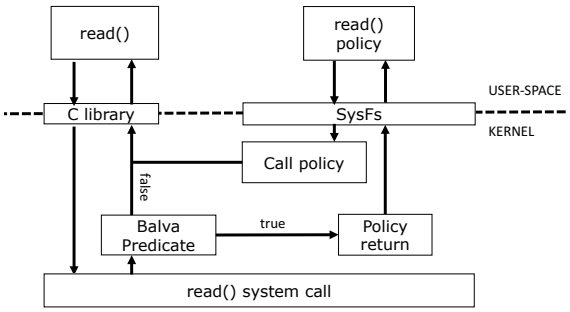

Figure 3: Control flow of read operations.

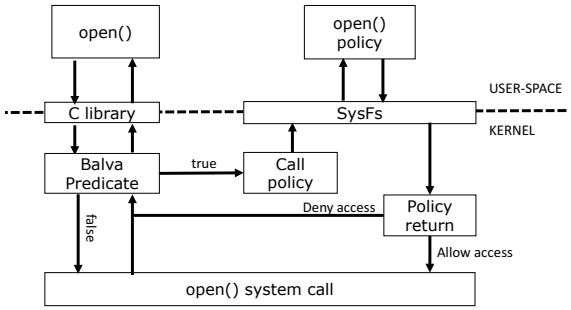

Figure 4: Control flow of open operations.

calls with Balva prototype functions. These prototype functions perform additional logic in addition to normal system call execution. To execute the read system call, the runtime will identify any meta-code attached to the file access operation, and invoke the associated code before executing the original system call. Access token predicates are handed to Balva through the metadata layer of the files, which ensures API compatibility. To protect the Linux kernel, we have decoupled the predicate and meta-code execution by handling the former within the kernel and the latter in user space. Since predicate functions are designed to be simple regular expression evaluations, the penalty of executing a predicate function for every file is minimal.

Figure 2 depicts how writes are executed within the kernel space and the user space. Balva intercepts the initial write and passes the intermediate buffer to our meta-code interface or the system call depending on the output of the predicate evaluation. If the predicate returns true, the kernel thread handling the system call will block while waiting for the user level meta-code callbacks to be completed. The user level meta-code will read the intermediate buffer and pass it back to the kernel with any potential changes. Once the user-level execution has completed, the Balva kernel module will invoke the original system call with the modified buffer. Once the routing completes, the kernel module will return to the original calling application. Similarly, Figure 3 shows the control flow of a read operation. The difference between handling read and write operations is that we call the original system call prior to the user space policy for reads, while write operations call the policy prior to the system call.

Figure 4 shows how an open system call is executed. The user space policy is called if the predicate function returns true, and if the return value of the policy is to deny access the kernel module will immediately return an appropriate error value to the calling program. If the meta-code policy returns successfully, the kernel module will call the file system implementation of sys_open, and return the file handle returned by the system call. The sys_open call creates a new file descriptor handle for the calling process. Access control is implemented by associating specific files with access control mechanisms. If a mechanism denies access to a file, the mechanism can simply return the appropriate error message without invoking the original system call.

3.1 Policy Examples

Adding policies to be executed on data accesses is done by incrementally adding meta-code modules in Balva.

Data Transformation We implement data transformation policies by adding a transformation step to reads and writes. For a write operation, we relay the data d from the system call to the callback function implementing the transform. The callback function will then transform the input data and the runtime will return the transformed data d' to the kernel module which then completes the system call. The data d' is viewed by the file system and mirrored to the local hard drive. Conversely, we can retrieve the original data by calling the read callback for a reverse transform policy with the transformed data d'.

Using this technique, we can implement an encryption policy. The callback routine for write can encrypt the data d and write the encrypted data d' to disk. An application reading the data will invoke the callback routine for the read operation, and transform the encrypted data d' into the decrypted data d viewed by the application. With an encryption policy, we can retrofit encryption to existing systems without having to modify the application or file system.

Implementing filtering functionality in Balva can be done during both reads and writes. Filtering data during a write operation persists the data in the filtered state. Filtering can also be implemented by transfor-

ming the output data according to the policy for each read, with the unfiltered data residing on disk. Balva supports writing filtering and declassification policies that can adapt according to specific parameters, such as user and device privileges.

Access Control We can implement fine-grained access control policies by encrypting the data during writes and relying on a decryption key that is fetched remotely and stored temporarily. This enables us to write policies that reason with high-level concepts such as user roles within a corporation in relation with the contents of the files. To revoke access to the data, the key owner can revoke the access to the remote encryption key. By leveraging trusted computing platforms such as Intel SGX (McKeen et al. 2013), these policies can be enforced by executing within a secure environment.

Data Management Replication policies can be constructed by monitoring files for changes, and perform replication to an offline location according to the consistency and durability needs of the policy. We have implemented mechanisms for replicating data to the major cloud vendors: Amazon S3, Windows Azure Storage and Google Storage. With these mechanisms, we can construct replication specific policies where data is replicated to one or more cloud storage services in different geographical regions. For example, a user may be restricted to storing offsite backups within EU or US for certain types of files. We can also construct auditing policies to track the provenance and changes in files by logging file accesses to a remote location.

4 EVALUATION

We perform a series of benchmarks to identify additional latency incurred when interpositioning meta-code policies in the access path of data. When a file is associated with a specific policy, the runtime will execute user space code during a system call execution in the kernel, and the resulting additional context switch can be an expensive procedure. This context switch overhead is measured with a series of benchmarks to evaluate the validity of the meta-code execution model.

We deployed Balva on a Linux server equipped with an Intel Xeon E5-1620 processor running at 3.70 GHz, 64 GB of memory and a 500 GB Samsung 840 EVO SSD. We are running Ubuntu server 16.04, and we use a standard deployment of Linux Server.

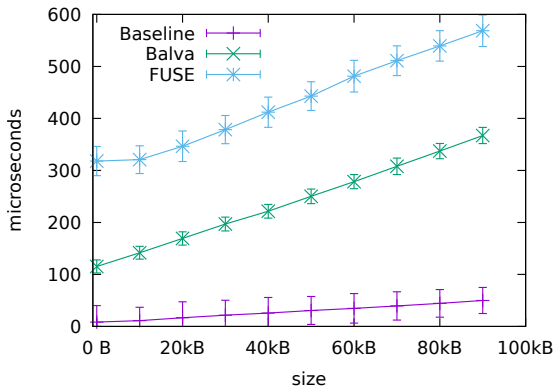

Figure 5: Read latency.

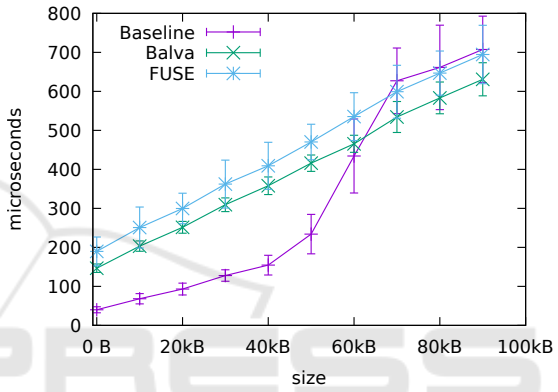

Figure 6: Write latency.

When the Balva kernel module is loaded into the running Linux kernel, all subsequent invocation of system calls will trigger evaluation of policy predicates. The overhead of such predicate evaluations are negligible as they do not trigger any context switch out from the kernel space.

4.1 System Call Latency

We calculate the system call latency by measuring the entry and exit difference in a user space program invoking the open, read and write system calls, averaging the measurements over a million individual calls to the kernel. We open three different files that will behave differently. The first will not invoke any meta-code policies. This provides us with a baseline measurement to compare the different implementations. The second will invoke a Balva meta-code policy that mirrors the input, and represents a minimal no-op policy. Additionally, we compare our kernel module to an implementation using FUSE to capture file system calls.

Figure 5 shows the average latency of a read system call. We observe that Balva incurs an additio-

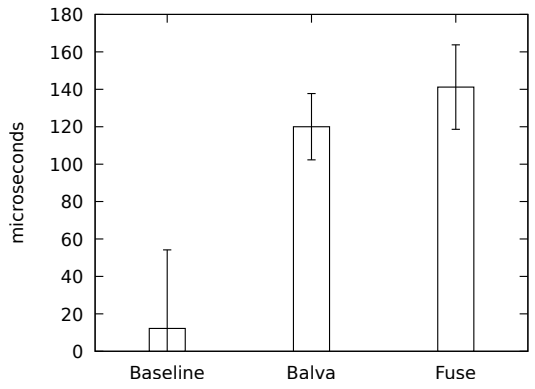

Figure 7: Open latency.

nal cost for our setup. With a buffer size of 1 byte, Balva has a 13.7 times higher read latency compared to the baseline. For 100 kB buffer sizes, Balva is 7.4 times slower. The write system call latency can be seen in Figure 6. Compared to the FUSE implementation, Balva performed better for all sizes. Compared with the baseline system call, Balva incurs an additional latency cost of approximately two to three times the baseline latency. The results for the open system call can be seen in Figure 7, and show comparable results to the read and write latency.

Our experiments reveal that even with the incurred overhead of additional policy execution, Balva is able to perform comparably to the base Linux file system. Invoking a user defined policy is not a costly operation, but the overall additional cost of executing these depends on the individual meta-code functionality. Note that we do not make any specific assumptions on the underlying file system mechanisms such as caching and prefetching. We do not specify that the file system should bypass the cache for our experiments.

4.2 Replication Policy Latency

In this experiment, we measure the write latency of a policy where each write is replicated to Amazon S3. Specifically, the meta-code policy writes a copy of all data updates to an Amazon S3 instance before allowing the application to continue. The S3 instance is located in Ireland (eu-west-1), and we measured an average ping latency of 74.0 milliseconds between the local machine and the Amazon S3 front ends. Since S3 does not allow incremental updates, we need to rewrite the entire file each time. To avoid reading and writing to a large file for each operation, we partition the file with a fixed block size of 4096 bytes. With a partitioned file, we can perform incremental updates by only updating the respective blocks. We average the time it takes to complete a million random write operations of 1 byte to a 1 GB file, and compare the results with writing to the local hard drive. The results can be seen in Table 1.

Table 1: Average latency of write system calls when replicating to Amazon S3.

Policy	latency
None (Baseline)	6.8 μs
No-op policy	14.7 μs
Replicate to S3 (Ireland)	188.0 ms

We observe as expected that the latency cost of the additional remote cloud write is quite high compared to only writing to the local disk. However, the incurred cost comes with the added benefit of having the data replicated to a remote location with a strong consistency model. The meta-code policy is implemented with approximately 300 lines of C. A more refined solution would be to relax consistency requirements, and cache data locally before writing to the cloud asynchronously. This solution would mask some of the write latency at the cost of additional code complexity and a weaker consistency model.

5 RELATED WORK

FUSE (Singh 2006) allows users to implement custom file systems without having to reason with the OS kernel. However, a FUSE implementation of Balva limits us to operate on files within a subdirectory (Hurley and Johansen 2014). Additionally, we would need to implement the file system itself or simply mirror the local file system. By interpositioning policies between the application and the I/O system calls, we do not have to reason with implementing a file system or the details of the underlying file system.

Data capsules (Song et al. 2012) resemble our model of adding functionality to data items. However their data protection as a service paradigm does not appear to extend beyond the reach of a single cloud; Balva is built for hybrid and multi-cloud environments where security policies are implemented throughout. Associating code with individual data items also has similarities to the active documents work (Dourish et al. 2000) and Monitoring-Oriented Programming (Chen and Roşu 2007). Our scope is a bit broader than verifying code execution, but some of the techniques employed are similar.

Guardat (Vahldiek-Oberwagner et al. 2015) takes a similar approach to enforcing policies at file accesses. Their approach is to allow users to specify data access policies using the Guardat policy language. Balva allows users to write policies in fami-

liar programming languages, only relying on a minimal library that communicates with the kernel module. Executing the policies in user space isolates the OS kernel from faulty policies while supporting a rich set of policies.

6 CONCLUSION

This paper describes Balva: a cloud storage service that transparently interpositions meta-code between applications and file system calls. By implementing security and management policies as meta-code attached to objects, Balva enables users to extend the storage system at runtime to enforce customized policies for encryption, replication, auditing, and access control. The experimental evaluations of the Balva kernel module show that our flexible security and management policies can be supported with little overhead on the I/O performance.

ACKNOWLEDGMENTS

This work was supported in part by the Norwegian Research Council project numbers 231687/F20. We would like to thank the anonymous reviewers for their useful insights and comments.

REFERENCES

Ardagna, Danilo (2015). "Cloud and Multi-cloud Computing: Current Challenges and Future Applications". In: *Proceedings of the Seventh International Workshop on Principles of Engineering Service-Oriented and Cloud Systems*. PESOS '15. Florence, Italy: IEEE Press, pp. 1–2.

Chen, Feng and Grigore Roşu (2007). "Mop: an efficient and generic runtime verification framework". In: *ACM SIGPLAN Notices*. Vol. 42. 10. ACM, pp. 569–588.

Dobre, Dan, Paolo Viotti, and Marko Vukolić (2014). "Hybris: Robust Hybrid Cloud Storage". In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. Seattle, WA, USA: ACM, 12:1–12:14.

Dourish, Paul et al. (2000). "A programming model for active documents". In: *Proceedings of the 13th annual ACM symposium on User interface software and technology*. ACM, pp. 41–50.

Felter, Wes et al. (2015). "An updated performance comparison of virtual machines and Linux containers". In: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pp. 171–172.

Gurrin, Cathal, Alan F. Smeaton, and Aiden R. Doherty (2014). "LifeLogging: Personal Big Data". In: *Foundations and Trends in Information Retrieval* 8.1, pp. 1–125.

Hurley, J. and D. Johansen (2014). "Self-Managing Data in the Clouds". In: *2014 IEEE International Conference on Cloud Engineering*, pp. 417–423.

Jia, Qin et al. (2016). "Smart spot instances for the supercloud". In: *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*. ACM, p. 5.

Johansen, Dag and Joseph Hurley (2011). "Overlay cloud networking through meta-code". In: *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*. IEEE, pp. 273–278.

Johansen, Håvard D. et al. (2013). "Combining Video and Player Telemetry for Evidence-Based Decisions in Soccer". In: *Proc. of the Int. Congr. on Sports Science Research and Technology Support*.

Johansen, Håvard D et al. (2015). "Enforcing privacy policies with meta-code". In: *Proceedings of the 6th Asia-Pacific Workshop on Systems*. ACM, p. 16.

McKeen, Frank et al. (2013). "Innovative instructions and software model for isolated execution." In: *HASP@ ISCA*, p. 10.

Renesse, Robbert van et al. (2013). "Secure Abstraction with Code Capabilities". In: *Proc. of the 21st Euromicro Int. Conf. on Parallel, Distributed, and Network-Based Processing*.

Schwab, Klaus (2016). "The fourth industrial revolution". In: World Economic Forum Geneva.

Singh, Sumit (2006). *Develop your own filesystem with FUSE*. Developer Works. https://www.ibm.com/developerworks/library/l-fuse/. IBM.

Soltesz, Stephen et al. (2007). "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors". In: *Proc. of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*. EuroSys '07, pp. 275–287.

Song, Dawn et al. (2012). "Cloud data protection for the masses". In: *Computer* 45.1, pp. 39–45.

Stenhaug, Magnus, Håvard Johansen, and Dag Johansen (2016). "Transforming Healthcare through Life-long Personal Digital Footprints". In: *Proc. IEEE Conference on Connected Health: Applications, Systems and Engineering Technologies: The 1st International Workshop on Cloud Connected Health*. CHASE '16. IEEE.

Vahldiek-Oberwagner, Anjo et al. (2015). "Guardat: Enforcing data policies at the storage layer". In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM, p. 13.

Wu, Zhe et al. (2013). "Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, pp. 292–308.