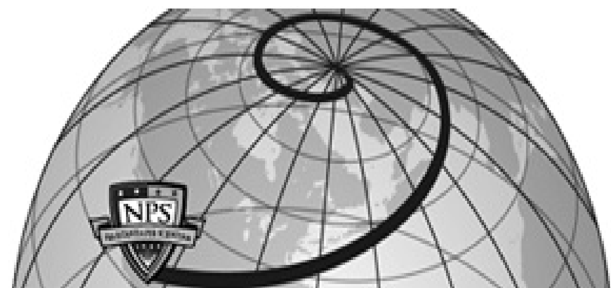
Calhoun
Institutional Archive of the Naval Postgraduate School



Calhoun: The NPS Institutional Archive

DSpace Repository

Faculty and Researchers | Faculty and Researchers' Publications

2015

Re-thinking Kernelized MLS Database Architectures in the Context of Cloud-Scale Data Stores

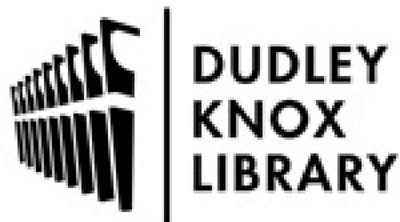
Nguyen, Thuy D.; Gondree, Mark; Khasalim, Jean; Irvine, Cynthia

Springer

Nguyen, T.D., Gondree, M., Khosalim, J. and Irvine, C., 2015, March. Re-thinking Kernelized MLS Database Architectures in the Context of Cloud-Scale Data Stores. In International Symposium on Engineering Secure Software and Systems (pp. 86-101). Springer, Cham.

http://hdl.handle.net/10945/60184

Downloaded from NPS Archive: Calhoun



Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

Re-thinking Kernelized MLS Database Architectures in the Context of Cloud-Scale Data Stores

Thuy D. Nguyen, Mark Gondree, Jean Khosalim, and Cynthia Irvine

Department of Computer Science, Naval Postgraduate School
Monterey, California 93943
{tdnguyen,mgondree,jkhosali,irvine}@nps.edu

Abstract. We re-evaluate the kernelized, multilevel secure (MLS) relational database design in the context of cloud-scale distributed data stores. The transactional properties and global integrity properties for schema-less, cloud-scale data stores are significantly relaxed in comparison to relational databases. This is a new and interesting setting for mandatory access control policies, and has been unexplored in prior research. We describe the design and implementation of a prototype MLS column-store following the kernelized design pattern. Our prototype is the first cloud-scale data store using an architectural approach for high-assurance; it enforces a lattice-based mandatory information flow policy, without any additional trusted components. We highlight several promising avenues for practical systems research in secure, distributed architectures implementing mandatory policies using Java-based untrusted subjects.

1 Introduction

Resource sharing exists at several layers in the cloud. For example, platform-as-a-service usually employs virtualization with shared hardware; software-as-a-service may provide multi-tenant database services (*e.g.*, [37]). Given this shared resource environment, information leakage is a major concern in a multi-customer cloud [30,26]. Further, a variety of sensitive data is being managed by community and private clouds in governments and industries across the globe, *e.g.*, healthcare organizations in the U.S. and EU [12]. The U.S. government is using a community cloud infrastructure for processing and sharing intelligence data [25] and is planning different tactical cloud environments [6,24] for ground and afloat operations. The output of tactical sensors to these clouds has been dubbed a "Data Flood" problem [29]. These Big Data challenges go beyond the need for new analytics: a leak resulting from this flood may pose grave danger to both human intelligence sources and national security. The need to manage sensitive and classified data in shared cloud infrastructures motivates enforcing strict, mandatory policies for information flow control, through the use of systems following the same rigor applied to a security kernel [32].

F. Piessens et al. (Eds.): ESSoS 2015, LNCS 8978, pp. 86–101, 2015.
© Springer International Publishing Switzerland 2015

It is in this context that we re-explore the design of high-assurance multilevel secure database systems, adapted for cloud-scale data stores. Prior work has considered relational database management systems (RDBMS) preserving mandatory information flow policies (*i.e.*, the Bell-LaPadula model). At the time, relational databases appeared to be a multi-purpose "one size fits all" solution; this perspective, however, has substantially waned [35]. An emerging trend is to select the data model and query model one's application requires, then to select a storage back-end appropriate for the situation. Experience has begun to show that often the resultant model is not relational, nor does the application require ACID properties (atomicity, consistency, isolation, durability). As a result, various high-availability, massively-scalable, non-relational ("NoSQL") databases have found adoption in a cloud context. These new databases tend to not guarantee ACID properties, instead relaxing consistency in favor of availability and network partition tolerance. The success of non-relational databases to support a variety of cloud services has demonstrated that many natural and important applications—*e.g.*, content distribution, content management systems, massively parallel data mining—are not efficiently maintained as relational models.

Our work makes the following contributions:

- We formulate the problem of multilevel security for cloud data stores—prior work considered only MLS relational models and MLS transactional systems.
- We propose the design of a scalable data store following BigTable's design, capable of enforcing an MLS policy; the design uses a variant of the kernelized architecture approach, requiring no trusted components external to the OS.
- We implement a prototype of our design using Apache HBase and HDFS, requiring only small modification to run as MLS-aware untrusted subjects.
- We experimentally evaluate our prototype, verifying compatibility using several popular cloud applications (e.g., Titan, Apache Storm) and assessing performance using known cloud benchmarking tools.

We identify limitations in adapting a large class of cloud applications—*i.e.*, those making extensive use of in-memory data structures, employing languages like Java—for kernelized systems. Our performance experiments call into question the viability of the kernelized design in the context of cloud-scale systems; we discuss these findings and suggest possible directions for future work.

Our prototype follows the Hinke-Schaefer variant of the kernelized database architecture pattern. This design pattern allows the entire application to be executed without privileges in an MLS environment, while supporting all allowable access patterns, *e.g.*, read-down. This is motivated by trusted computing base (TCB) minimization requirements [1,23]. Untrusted applications built around this pattern are called *MLS-aware* [21]. We select this MLS database architecture for exploration as it is credited as best facilitating the "retrofit" of existing code to run on high-assurance systems [15]. It is known to be inefficient when tuple-level labels are required, as data must be divided among many different operating system objects [17,15]; thus, our prototype only supports labels at the coarsest (table-level) granularity. From a security perspective, the only major

criticism of Hinke-Schaefer relates to support for transactions, which our design avoids by adopting a weakened consistency model.

2 Related Work

The problem of information flow control has received growing attention in infrastructure-as-a-service (IaaS) cloud service models. In particular, some projects have explored the threat of placing co-resident VMs in shared clouds [30,5] for side-channel attacks [39,40]. The Xenon VMM [26] is a hardened version of Xen satisfying a separation policy appropriate for controlling these flows. Relatedly, Wu *et al.* [38] design a proof-of-concept IaaS system based on Eucalyptus, implementing Chinese Wall rather than a strict separation policy. Watson [36] proposes a more general set of rules for information flow control between sets of nodes performing a joint computation in a federated setting. Information flow control in storage-as-a-service models, however, has not been well-explored, nor have the lessons of MLS RDBMS research been re-evaluated in this new domain.

Some non-relational data stores support native mechanisms for access control. Apache Accumulo [2] is a column-store that extends the BigTable design to support cell-level access control. Each cell is assigned a security label encoding non-discretionary, attribute-based access control rules; these are not equivalent to MLS labels, *i.e.* they are not used to enforce an information flow control policy. In particular, users with permissions to write a cell can modify its label, or write this data to a new cell with less restrictive visibility (in MLS terms, either violating tranquility or performing a downgrade). Relatedly, Apache HBase implements access control lists (ACLs) at the table- and column-granularity. As of v0.98, HBase features both cell-level visibilities, like Accumulo, and ACLs on cells [3]. These application policies are orthogonal to those considered by our approach, and could be incorporated for more expressive policies.

Roy *et al.* [31] present Airavat, a Hadoop-based MapReduce framework with enhancements for controlling information flow. Airavat runs on SELinux, using its type enforcement for domain isolation. Airavat modifies HDFS to manage its own security labels. Using these, it implements a custom policy based on differential privacy, to minimize leakage of private data during MapReduce computations. In particular, the MapReduce framework (including Airavat) and reducer implementations are trusted subjects. In contrast, in MLS-BTC there are no trusted subjects external to the OS.

3 MLS Architectures Overview

Before discussing a proposed design for an MLS cloud data store, we briefly review architectures for MLS RDBMS and cloud data stores, generally.

MLS Database Architectures. Several secure architectures have been previously identified for multilevel databases, *i.e.*, the Woods Hole architectures [9]. Of these designs, the kernelized architecture provides the basis for our work.

The reader is directed to existing survey work for an in-depth description of other MLS database architectures, *i.e.*, the trusted subject, integrity lock and distributed architectures [28]. For a *kernelized architecture*, multilevel relations are decomposed into single-level relations managed external to the TCB. Different ways to decompose relations, and different ways of managing the resultant single-level data, lead to variants of the kernelized design. In the Hinke-Schaefer architecture [18], there are no trusted components outside the kernel; other variants include SeaView [13] and Lock Data Views [34]. In all variants, multiple single-level untrusted subjects manage the (decomposed) single-level relations.

MLS Cloud Data Stores. No prior MLS database work applies to data stores with relaxed ACID properties, to databases that do not encode relational models, or to databases lacking fixed schemas. We find mandatory access control (MAC) to be orthogonal to the transactional properties of relational databases, and believe MLS non-relational stores to be a new and interesting domain.

Indeed, certain design patterns for distributed, cloud data stores seem synergistic with architectures for multilevel relational databases. For example, in MLS systems, information flow restrictions require some data and single-level services to be inaccessible to clients based on its level; data store designs that accommodate partition tolerance and availability in the presence of failures seem to accommodate adaptation to these environments.

Data store designs that employ append-only, log-structured storage managed by the underlying TCB can be implemented using a lock-free design, possibly allowing access to high-readers while a low-update is in progress. Such concurrent access comes at the expense of replacing strong consistency by eventual consistency, which for many stores is part of the intended design. Thus, systems whose data structures use write-ahead logs (*e.g.*, to support a crash-only design [7]) may, in practice, enable eventually-consistent, read-down operations.

For some MLS relational databases, clients access a single database front-end. Data sharding allows a client to independently determine the location of nodes in the cluster holding its data, to contact each node directly. In a replicated architecture—in which trusted front-end agents mediate access to untrusted backend databases [28]—sharding may entirely eliminate the need for trusted front-ends, allowing single-level subjects to interact with the services at their level, directly, to access data at or below their level.

The most common criticism of the Hinke-Schaefer architecture is the difficulty of implementing transactions, due to the need for read-locks and, thus, the possibility of flows that violate MAC policy [15]. Some NoSQL stores, however, sacrifice transactions for scalability, foregoing read locks and, thus, this problem.

4 Kernelized MLS Column-Store

We present the design for an MLS column-store following a kernelized architecture. We call this the *MLS-aware BigTable Clone* (MLS-BTC) design, as it is largely applicable to any data store following the published design of Google's BigTable [8]. To describe MLS-BTC, we adopt basic terminology employed by

Apache HBase, a popular open-source BigTable clone. In our design, *all* policy enforcement is performed by an underlying trusted operating system; it mediates access to all resources and enforces an MLS policy. Single-level clients interact with MLS-aware, single-level applications running on each server, which in turn may access resources using interfaces exposed by the trusted OS. A benefit of this approach is that applications are not involved in MLS policy enforcement and, thus, reside outside the TCB and do not need to be engineered to be trustworthy.

4.1 MLS-BTC Design

In MLS-BTC, each table holds timestamped data, organized by rows and columns, and grouped by column families. Table data is partitioned into *regions* of contiguous rows. A *region server* (RS) manages a set of regions, handling all operations on its assigned regions, and splitting regions that have grown above the configured region size. For persistence, each region is stored to a distributed file system which is, itself, an MLS-aware service following a kernelized architecture, *i.e.*, the underlying trusted OS enforces the policy for accessing stored objects.

Each MLS-BTC node holds multiple untrusted RS instances, one per level. Each RS stores its data to a directory associated with its level, using the MLS-aware distributed file system. There are no explicit labels in the MLS-BTC columnar data. Rather, following the Hinke-Schaefer design, data is stored to labeled operating system objects. This approach is known to be inefficient when database access patterns require data to be labeled at a fine-granularity [17]. In our design, object labels are coarse (per-level tables), the table namespace is partitioned per level, and the table's constituent objects are stored to different per-level file system directories. RS instances access table data at lower levels by explicitly reading from the appropriate per-level directory.

A dedicated per-level master is responsible for management of RS instances at its level. These duties include table creation, load-balancing regions across RS instances, and handling RS failures. As master instances require knowledge of tables at lower levels, each master is MLS-aware. The master instances and RS instances coordinate through a distributed locking/synchronization system.

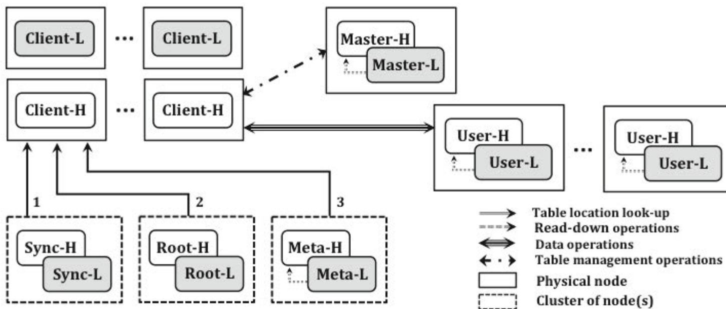


Fig. 1. MLS-BTC Component Relationship

MLS-BTC follows a BigTable architecture, using three special classes of region servers: the *Root* RS, *Meta* RS, and *User* RS. These servers help a client locate the RS hosting a specific table row, using the same region lookup mechanism used in BigTable (Fig. 1). A client locates the Root RS for its level via the distributed synchronization service at its session level. Next, the client contacts the Root RS to find the appropriate Meta RS for the request. The client contacts the Meta RS to find the location of the User RS managing the region for the requested row. Finally, the client contacts the User RS to access the row data.

Concept of Operations. In a typical MLS system, a user session must be associated with a sensitivity level, used to limit access to resources in accordance with MLS policy. For the MLS-BTC system, the sensitivity level of a user's session is associated, statically, with the level of the network interface on which the request is received[1]. A client communicates with the per-level region servers to manipulate table data (*e.g.*, get, put, delete, multi-row scan). A single-level client can write to tables at its session level, and read tables at or below its session level. A client communicates with a per-level master server to issue certain administrative functions (*e.g.*, create, list, or delete tables, add to or drop from column families). The *list-tables* operation returns data for any tables at or below the client's session level.

Design Features. The primary design goals of the MLS-BTC system are: (a) to defer all MLS policy enforcement to the underlying TCB; (b) to use no trusted subjects external to the OS, avoiding extending the TCB boundary, *e.g.*, no trusted proxies or trusted front-ends to communicate between processes at different levels; (c) re-use existing code for untrusted subjects, minimizing the modifications required to make these MLS-aware; (d) expose a familiar client API. We ensure all MLS functionality is deferred to the underling OS by re-designing MLS-BTC components as untrusted subjects following a Hinke-Schaefer design. Re-using existing code for untrusted subjects with only small modification allows us to leverage complex, feature-rich server behavior, and future upgrades to that code, without extending the TCB boundary. A familiar API—such as one compatible with an existing, column-oriented store—will allow MLS-BTC to support legacy applications and simplify new application development.

4.2 Prototype Implementation

Each node in an MLS-BTC cluster is a platform running a trustworthy operating system enforcing an MLS policy. The prototype currently implements this component using SELinux, configured to enforce a MAC policy based on the Bell-LaPadula confidentiality model [16]. The prototype's untrusted subjects are based on a number of existing open-source components, running either unmodified or with small modification:

[1] We admit labeling interfaces imposes some deployment inflexibility, adopting it for simplicity; in Sect. 4.2 (Limitations), we suggest more flexible and complex designs.

- *MLS-aware Master & Region Servers.* The prototype re-uses the Master Server and Region Server (RS) components of HBase [14], modified to be MLS-aware HBase (MA-HBase) components. This provides clients with a cross-domain read-down capability, constrained by the SELinux MAC policy.
- *MLS-aware Distributed File System.* The prototype re-uses components of the Hadoop Distributed File System [33] (HDFS), modified to produce an MLS-aware HDFS (MA-HDFS) component. Details for the design and architecture of the MA-HDFS component are reported in prior work [27].
- *Per-Level Locking/Synchronization Services.* The prototype re-uses components of Zookeeper [19] to implement a distributed synchronization and locking service, available at each level. We configure and re-use these components, wholesale, as single-level subjects on separate nodes.

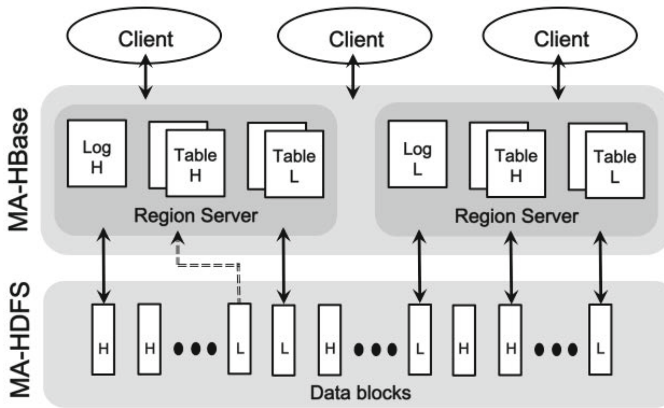


Fig. 2. MLS-BTC Table Storage

Table and Directory Organization. The MA-HBase cluster jointly manages a set of per-level tables (Fig. 2). As in HBase, there are three types of table: the root table, the meta table, and the user tables; each are maintained by the Root RS, Meta RS and User RS, respectively. The root and meta table naming conventions are unchanged. The user table namespace, however, is divided by level (*level.TableName*), to signal to an RS when a read-down operation is required to access table data at a lower level.

HBase stores all table data under a configurable root location in HDFS. The directories under this root (*e.g.*, */hbase*) include a directory tree holding write-ahead log (HLog) data, and a tree holding per-region table (HFile) data. The constituent HLog and HFile objects managed by MA-HDFS are stored under a per-level root location (*e.g.*, */level/hbase*).

For both MA-HBase tables and MA-HDFS directories, the *level* indicator, used to partition the namespace and invoke read-down logic, is a human-readable string administratively associated with an SELinux sensitivity level.

Information Flow. The information flow between a client application (local or remote) and an MA-HBase server process is constrained by the system's MAC policy (see Fig. 3). An application can only communicate with an MA-HBase process running at its session level. When the application requires read-access to a table at a lower level, it must request the RS running at its session level to perform a read-down on its behalf. If the application attempts to contact an RS running at some lower level, the underlying trusted OS prohibits this.

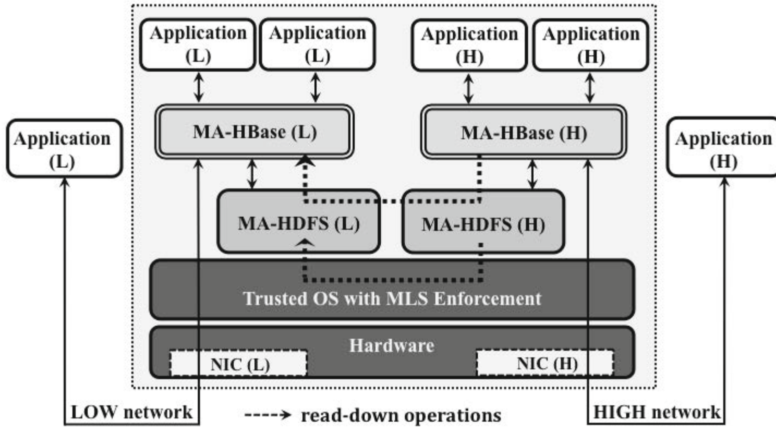


Fig. 3. Information Flow in MLS-BTC

The information flow between an MA-HBase RS process and an MA-HDFS server process is similarly contained: An MA-HBase RS may only communicate with MA-HDFS processes running at the same level. Thus, RS requests to read HLog or HFile data at lower levels must be issued from the RS to an MA-HDFS process at the same level, which in turn performs the read-down operations.

Read-Down Requests. Next, we explain how MLS-BTC handles write, read and read-down requests. This involves two steps: locating the appropriate User RS by the client, and handling the request by the User RS. In MA-HBase, we distinguish between two types of RS: the *authoritative* RS and the *surrogate* RS. The authoritative RS is the "owner" of an allocated region. It runs at the sensitivity level of the corresponding table and updates the MA-HDFS files for storing the row data associated with its regions. The surrogate RS runs at the client's session level and is responsible for handling read-down requests for table data managed by an authoritative RS at a lower level. This is required since a client cannot communicate directly with any lower-level authoritative RS instance. The number of authoritative and surrogate RS instances running on a node is defined administratively through MA-HBase configuration files.

When a client requests access to a row in a table at its session level, it locates the authoritative User RS associated with the row, via the Root and Meta RS.

When the client requests access to a row in a table at a lower level, *i.e.*, a read-down operation, the process is slightly different. The Meta RS recognizes the difference between the client's session level and table's level, and responds to the client with the location of an appropriate surrogate RS, rather than the authoritative RS. In turn, when the client contacts the surrogate RS with a request to read a row at a lower level, the surrogate RS performs a read-down operation to the MA-HDFS resources storing the row data. Since the Meta RS does not read-down on every meta table scan to retrieve region information at lower levels, client-side caching of meta table data poses a problem: prior scans of tables at its session level will not include all available lower regions. Thus, the MA-HBase client does not cache data obtained from the Root RS and Meta RS.

In HBase, an RS process maintains a database of all active storage objects associated with its region, called the *onlineRegions* database. This database is created during initialization, expanded when a new region is allocated to the RS, and modified when a region change is made (*e.g.*, when a row is modified or deleted). During a read request, an RS uses this database to locate the HDFS resources associated with the requested row. The database is held in private memory and is not visible to other RS processes.

In MA-HBase, each authoritative RS maintains a new data structure, the *onlineRegionsCache*, to expose its region data to surrogate RS instances at higher levels (see Fig. 4). The *onlineRegions* database is a complex data structure: a map of maps of lists of complex nested objects. This structure is located in the Java heap, and it grows and shrinks dynamically, in each of its dimensions. To expose its contents to higher levels, some form of IPC must be employed. Using shared memory (*i.e.*, re-implementing it as a library using the Java Native Interface) would be non-trivial. For example, POSIX shared memory sizes are defined at creation time, limiting the dynamic growth of the structure. Further, such a library would require new, custom logic for memory management; the lack of coordination between the memory managers—*i.e.*, Java's garbage collection and the management of the shared memory pool under the native library—would be especially problematic. Instead, the data is exposed using file-based IPC.

On a write request, the RS services the request, updates the *onlineRegions* database, flushes all recent modifications to MA-HDFS, then serializes the in-memory *onlineRegions* database to an *onlineRegionsCache* file. This file is stored to a RAM disk, accessible to surrogate RS instances at higher levels. The surrogate RS can read-down to the RAM disk, to de-serialize and interpret the data structure in response to read-down requests. Using the *onlineRegionsCache* database, a surrogate RS locates the MA-HDFS objects associated with table data at lower levels, and requests these from MA-HDFS processes running at the surrogate's level. Serializing the *onlineRegions* database required developing a custom serialization class, as standard Java object serialization mechanisms could not be used: the database contains inner classes with non-serializable attributes. Concurrent access to the *onlineRegionsCache* by multiple processes is synchronized using a lock-free, read-and-retry consistency mechanism.

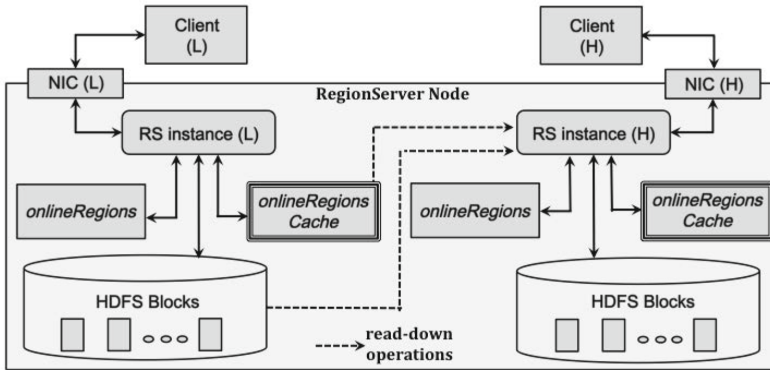


Fig. 4. Authoritative and Surrogate RS Detail

Limitations. The current MLS-BTC prototype system has a number of practical limitations, stemming from our objective to develop a functional, proof-of-concept, non-relational data store that closely follows a kernelized architecture. We summarize those limitations here.

Scalability. The current prototype accommodates policies with a relatively small number of sensitivity levels. For simplicity, the client's session level is associated with a level assigned to the receiving NIC; thus, the number of levels available for the system's policy is constrained by the number of NICs supported by the underlying trusted platform. To support more complex lattice structures, *i.e.*, the "gazillion problem" in MLS design [20], MLS-BTC could be extended with custom trusted components to associate a remote client with a session level and to start services dynamically on behalf of those subjects. The MYSEA system uses such an approach to implement its multilevel LAN concept [22].

Caching. Serializing objects to shared memory and maintaining a consistent image of in-memory objects accessible to subjects at higher levels comes with a performance penalty, discussed further in Sect. 4.3

Surrogate RS. To locate surrogate RS instances, each Meta RS uses a static look-up table providing the authoritative-to-surrogate mappings. For a large MLS-BTC cluster, a runtime mechanism for constructing and managing this mapping should be introduced, allowing authoritative and surrogate servers to enter and leave the system, dynamically.

Meta RS. The current prototype requires all Meta RS instances be co-located, to facilitate read-down to lower meta tables. For a large MLS-BTC cluster, this cannot be guaranteed: a low Master may elect to assign a low meta region to alternate nodes to load-balance. The Master at higher levels could recognize this, and migrate high meta tables in response. This workaround, however, poses a problem when Masters at incomparable levels migrate meta regions to different nodes, forcing higher-level Masters to make an irreconcilable choice regarding with whom they co-locate.

Implementation Complexity. The source lines-of-code (SLOC) metric provides an intuitive measure commonly associated with development cost and software complexity. We compare Hbase and HDFS with their MLS-aware counterparts (see Table 1) using the CLOC utility[2]. Summing across the total lines of source that changed, ∼3% of the untrusted codebase, and none of the untrusted codebase (*i.e.*, SELinux), required modification for MLS-BTC.

Table 1. SLOC Comparison

	Original	MLS-aware	Δ	% Δ
MA-HDFS [27]	89615	92263	3314	3.70%
MA-HBase				
Master	8624	8736	116	1.35%
RS	17170	18829	1715	9.99%
Client	7184	7420	270	3.76%
Other	66313	66732	411	0.62%
Total	188906	193980	5826	3.08%

Compatibility. To determine that our prototype data store is functionally compatible with legacy web-applications, while constraining these according to MLS policy, we tested three applications: Titan, Storm and AppScale.

Titan. Titan[3] is an open-source, distributed graph database designed for storing and querying large-scale graphs. We configured Titan to use our prototype as its storage backend. Titan's Gremlin tool was able to manipulate sample graphs stored in the data store: read/write graphs held in tables at the client's level, and read graphs held in tables at lower levels.

Storm. Storm[4] is an open-source, distributed stream processing platform. Storm does not run on Hadoop; however, using an HBase connector[5], Storm can be configured to use HBase as a storage back-end. We configured Storm to store processed data in a table at the client's level. Theoretically, other applications could read this Storm-processed data, either at or below their level.

AppScale. AppScale[6] is an open-source re-implementation of Google's App Engine platform. AppScale supports HBase as a storage back-end, to store a variety of persistent data used by the platform (*e.g.*, user-uploaded content, system metadata). We modified AppScale (v1.7.0) to use our prototype as its primary datastore, rather than the precompiled HBase distributed with AppScale. A sample GAE application, the *guestbook* program, was used to test AppScale's use

[2] Count Lines of Code, `http://cloc.sourceforge.net`
[3] Titan, `https://thinkaurelius.github.io/titan/`
[4] Storm, `https://storm.incubator.apache.org/`
[5] `https://github.com/jrkinley/storm-hbase`
[6] AppScale, `http://www.appscale.com/`

of the HBase API. The program could read and write to the HBase tables containing user messages at the client's level. Theoretically, the *guestbook* program could be modified to perform explicit read-downs to other table data.

4.3 Prototype Evaluation

To measure the performance of the MLS-BTC prototype, we used the Performance Evaluation (PE) benchmark distributed with HBase and the Yahoo! Cloud Serving Benchmark (YCSB) suite [11].

The PE benchmark implements the same tests used to evaluate performance for BigTable: a sequential read test (Seq-R), random read test (Rand-R), scan test (Scan-R), sequential write test (Seq-W) and random write test (Rand-W); see Chang *et al.* [8] for details. The benchmark employs a hard-coded table name in its tests; we added an option to specify the table to use, for testing read-down.

YCSB is a benchmark framework for evaluating the performance and elasticity of cloud storage systems, and has been employed to benchmark systems like Cassandra, HBase and PNUTS. YCSB provides a set of test workloads, to evaluate different aspects of a system's performance. All six workloads use a similar set of records as test data. For details on the test workloads, see Cooper *et al.* [10]. We followed the recommended test order (A, B, C, F, D, E), which keeps a consistent store size. Test data were loaded prior to running YCSB-A. Before starting YCSB-E, all tables from previous workloads are removed and new test data loaded. In YCSB, all workloads require writes before or during each run; thus, no read-down operation was tested.

Each test in the PE and YCSB benchmarks is executed in one of three scenarios: using 100,000 rows (100K), 500,000 rows (500K) and 1 million rows (1M) workload sizes. Before each trial, all stored HDFS/HBase data are removed, to start each trial from a comparable initial state.

Test Environment. The test environment consists of twelve nodes evenly distributed across four server blades in one of two racks. Each node is a virtual machine hosted on VMware ESXi 5.0.0. One rack contains three server blades (each, a Dell PowerEdge R710, with 8 CPUs x 2.925 GHz with hyper-threading active, 48GB RAM and Gigabit Ethernet). The other rack holds the remaining server blade (a Dell PowerEdge R610, with 8 CPUs x 2.26 GHz with hyper-threading active, 24GB RAM and Gigabit Ethernet).

Results. Benchmark results for the MLS-BTC prototype are summarized in Fig. 5. We normalize each trial by the average HBase performance—*i.e.*, mean of three trials under same test conditions with HBase—to obtain an "overhead factor," a positive multiplicand expressing performance relative to HBase. In general, all tests experience performance degradation, which is expected. The degree of degradation, however, varies significantly, impacted by the size and mixture of the workload.

The PE write-tests show substantial degradation, even for the relatively small 100K-row workload. In contrast, the PE read-tests exhibit overheads associated with both table creation and read performance. For most YCSB workloads, the

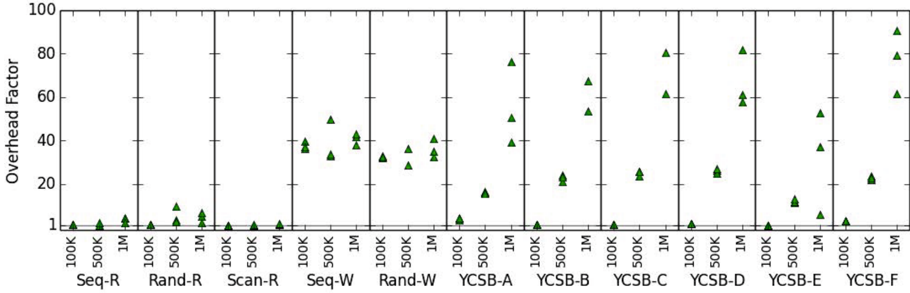


Fig. 5. Prototype performance, normalized by HBase performance

prototype is more than 50× slower than HBase. In all write workloads, performance degradation is the result of both caching the *onlineRegions* database (anytime a row is created or modified) and caching the HDFS namespace (anytime an HBase object stored in HDFS is created or modified). During read-down, performance degradation is the result of reading the cached *onlineRegions* database to handle each read-down request (see Fig. 6). During other reads, the degradation is attributable to lack of client-side caching of server metadata. In general, the most significant performance bottlenecks are associated with the caching of the data structures that maintain the HDFS namespace (FSImage) on the name node and the location of HDFS blocks (BlockMap) on the data nodes [27].

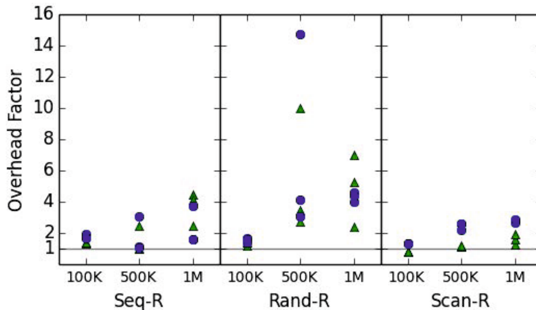


Fig. 6. Highlight of Fig. 5, including read-down performance (blue circle)

We note that more data is required for a rigorous characterization of system performance, but the observed data suggests a rough order-of-magnitude degradation: writes are processed ∼40× slower than HBase; reads are ∼1.5–8× slower, and mixed read/write workloads can experience ∼10–90× slowdown. In Sect. 5, we discuss some more general outcomes and lessons learned.

5 Discussion

We find the general approach of caching objects for Hinke-Schaefer is not appropriate for large, distributed systems in Java. In particular, the lack of efficient IPC mechanisms for object sharing calls into question the viability of the kernelized approach for managing MLS-aware Java applications. Most methods for IPC in Java are bi-directional (*i.e.*, socket-based IPC) or limited by small buffer sizes (*i.e.*, I/O-stream based IPC). As Java lacks an API for shared memory, MLS-BTC re-uses file-based IPC for OS-enforced data sharing. We find this is inefficient for sharing complex, in-memory objects across levels. We discuss some alternatives to get more acceptable performance, based on this observation.

The kernelized approach could be explored after enhancing Java with more flexible OS-enforced IPC interfaces. For example, Kaffe [4] is a research JVM with a process-level abstraction, allowing different Java processes to communicate via shared memory in a controlled way. Supporting HDFS/HBase on Kaffe under SELinux may be promising for kernelized MLS designs with Java-based systems, although Kaffe appears to no longer be maintained.

In the extreme, our experiences could be interpreted as evidence that the Hinke-Schaefer approach should be abandoned, and others explored. For example, using the trusted front-end variant of the kernelized architecture, a carefully-engineered trusted proxy may significantly improve performance. In MLS-BTC, such a trusted proxy can forward client requests at different levels to the appropriate RS processes, eliminating the need to cache the *onlineRegions* data structure. The challenge is to design a small, covert channel-free subject whose responsiveness and efficiency removes significant bottlenecks; this is challenging given that “responsive” and “channel-free” tend to be mutually exclusive.

In the integrity lock architecture—in which an untrusted DBMS stores all multilevel objects [28]—cryptographic protection on objects prevent untrusted subjects from processing aggregate queries. This requires the trusted front-end to be more complex, to perform extra post-query processing. Many key-value models, however, do not support an API with queries returning aggregate objects: simple put, get, delete semantics return individual objects. The integrity lock architecture can be re-explored in this context, re-evaluating all prior criticisms. In particular, the architecture’s (known) signaling channel could be bounded with respect to some popular datastore API.

6 Conclusion

We have presented a design for an MLS-aware column-store, faithfully following the kernelized design pattern. The resulting system, MLS-BTC, constrains access to resources at different levels, enabling read-down without any trusted subjects outside the TCB. Our prototype evaluation questions the practicality of the kernelized design approach to manage MLS-aware, untrusted Java-based applications. MLS-BTC is a distributed system based on HBase using SELinux for MLS policy enforcement; it is the first cloud-scale data store following a high-assurance design.

References

1. Anderson, J.: Computer security technology planning study. Technical Report ESD-TR-73-51, The Mitre Corporation, Air Force Electronic Systems Division, Hanscom AFB, Bradford, MA (October 1972)
2. Apache Accumulo Project. Apache Accumulo user manual version 1.5 (2014)
3. Apache HBase Project. The Apache HBase reference guide (2014)
4. Back, G., Hsieh, W.C.: The KaffeOS java runtime system. ACM Trans. Program. Lang. Syst. 27(4), 583–630 (2005)
5. Bates, A., Mood, B., Pletcher, J., Pruse, H., Valafar, M., Butler, K.: Detecting co-residency with active traffic analysis techniques. In: Proc. of the ACM Workshop on Cloud Computing Security, pp. 1–12 (2012)
6. Buxbaum, P.: Clouds at the edge: Army intel program deploys first tactical cloud computing node in Afghanistan. Geospatial Intelligence Forum 11(2), 8–12 (2013)
7. Candea, G., Fox, A.: Crash-only software. In: USENIX Workshop on Hot Topics in Operating Systems, pp. 67–72 (2003)
8. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst. 26(2), 4:1–4:26 (2008)
9. Committee on Multilevel Data Management Security. Multilevel data management security. Technical report, Air Force Studies Board (1983)
10. Cooper, B.: YCSB core workloads (2010), http://goo.gl/NJBV4L
11. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proc. of the ACM Symp. on Cloud Computing, pp. 143–154 (2010)
12. Currie, W., Seddon, J.J.: A cross-country study of cloud computing policy and regulation in healthcare. In: Proc. of the 22nd European Conf. on Information Systems (2014)
13. Denning, D.E., Lunt, T.F., Schell, R.R., Shockley, W.R., Heckman, M.: The SeaView security model. In: Proc. of the IEEE Symp. on Security and Privacy, pp. 218–233 (1988)
14. George, L.: HBase: The Definitive Guide. O'Reilly Media (2011)
15. Graubart, R.D.: A comparison of three secure DBMS architectures. In: Database Security III: Status and Prospects, pp. 167–190 (1989)
16. Hanson, C.: SELinux and MLS: Putting the pieces together. In: Proc. of the Annual SELinux Symp. (2006)
17. Hinke, T.: Secure database management system architectural analysis. In: 2nd Aerospace Computer Security Conf., pp. 65–72 (1986)
18. Hinke, T.H., Schaefer, M.: Secure data management system. Technical Report RADC-TR-75-266, System Development Corp. (November 1975)
19. Hunt, P., Konar, M., Junqueira, F., Reed, B.: Zookeeper: Wait-free coordination for internet-scale systems. In: Proc. of the USENIX Annual Technical Conf. (2010)
20. Irvine, C.: A multilevel file system for high assurance. In: Proc. of the 1995 IEEE Symp. on Security and Privacy, pp. 78–87 (May 1995)
21. Irvine, C.E., Acheson, T., Thompson, M.F.: Building trust into a multilevel file system. In: Proc. 13th National Computer Security Conf., pp. 450–459 (1990)
22. Irvine, C.E., Nguyen, T.D., Shifflett, D.J., Levin, T.E., Khosalim, J., Prince, C., Clark, P.C., Gondree, M.: MYSEA: The Monterey security architecture. In: Proc. of the ACM Workshop on Scalable Trusted Computing, pp. 39–48 (2009)
23. Jaeger, T.: Operating System Security. Morgan and Claypool Publishers (2008)

24. Killion, T.: Future naval capabilities. In: NDIA 15th Annual Science and Engineering Technology Conf. (April 9, 2014)
25. Konkel, F.: Intelligence community builds cloud infrastructure. In: FCW (September 2013), http://goo.gl/mfYjV9
26. McDermott, J., Montrose, B., Li, M., Kirby, J., Kang, M.: Separation virtual machine monitors. In: Proc. of the Annual Computer Security Applications Conf., pp. 419–428 (2012)
27. Nguyen, T., Gondree, M., Khosalim, J., Irvine, C.: Towards a cross-domain MapReduce framework. In: IEEE MILCOM 2013, pp. 1436–1441 (2013)
28. Notargiacomo, L.: Architectures for MLS database management systems. In: Information Security: An Integrated Collection of Essays, pp. 439–459 (1995)
29. Porche III, I.R., Wilson, B., Johnson, E.-E., Tierney, S., Saltzman, E.: Data_flood: Helping the Navy Address the Rising Tide of Sensor Information. Rand (2014)
30. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In: Proc. of 16th ACM Conf. on Computer and Communications Security, pp. 199–212 (2009)
31. Roy, I., Setty, S.T.V., Kilzer, A., Shmatikov, V., Witchel, E.: Airavat: Security and privacy for MapReduce. In: Proc. of the USENIX Conf. on Networked Systems Design and Implementation (NSDI), p. 20 (2010)
32. Shockley, W., Schell, R., Thompson, M.F.: The importance of high assurance computers for command, control, communications, and intelligence systems. In: Aerospace Computer Security Applications Conf., pp. 331–342 (December 1988)
33. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Proc. of the 26th IEEE Symp. on Mass Storage Systems and Technologies (MSST), pp. 1–10 (2010)
34. Stachour, P.D., Thuraisingham, B.: Design of LDV: A multilevel secure relational database management system. IEEE Trans. Knowledge and Data Engineering 2, 190–209 (1990)
35. Stonebraker, M., Cetintemel, U.: One size fits all: an idea whose time has come and gone. In: Proc. of the Intl. Conf. on Data Engineering, pp. 2–11 (2005)
36. Watson, P.: A multi-level security model for partitioning workflows over federated clouds. In: Proc. of the IEEE Conf. on Cloud Computing Technology and Science (CloudCom), pp. 180–188 (November 2011)
37. Weissman, C.D., Bobrowski, S.: The design of the force.com multitenant internet application development platform. In: Proc. of the 2009 ACM SIGMOD Conf., pp. 889–896 (2009)
38. Wu, R., Ahn, G.-J., Hu, H., Singhal, M.: Information flow control in cloud computing. In: Proc. of the Conf. on Collaborative Computing (CollaborateCom), pp. 1–7 (October 2010)
39. Xu, Y., Bailey, M., Jahanian, F., Joshi, K., Hiltunen, M., Schlichting, R.: An exploration of L2 cache covert channels in virtualized environments. In: Proc. of the ACM Workshop on Cloud Computing Security, pp. 29–40 (2011)
40. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-VM side channels and their use to extract private keys. In: Proc. of the ACM Conf. on Computer and Communications Security, pp. 305–316 (2012)