On the formalization, design, and implementation of component-oriented access control in lightweight virtualized server environments

Kirill Belyaev, Indrakshi Ray*
Department of Computer Science, Colorado State University, Fort Collins, CO, USA

ARTICLE INFO
Article history:
Available online

Keywords:
Access control
Service and systems design
Tuple spaces
Data and application security
Denial of service protection
Security architectures

ABSTRACT
In modern day operating systems, such as Linux, it is now possible to handle a large number of concurrent application services on a single server instance. Individual application components of such services may run in different isolated runtime environments, such as chrooted jails or application containers, and may need access to system resources and the ability to collaborate and coordinate with each other. We formalize the access control requirements of such components; our model allows access to OS resources on a need-to-know basis and also controls collaboration and coordination among service components running in disjoint containerized environments under a single Linux OS server instance. Such access control is managed and enforced through a Linux Policy Machine (LPM) that acts as the centralized reference monitor and provides a uniform interface for accessing system resources and requesting application data and control objects. We present the design of the LPM and provide an implementation to demonstrate the feasibility of our approach.

© 2017 Elsevier Ltd. All rights reserved.

1. Introduction

The advancements in contemporary multi-core CPU architectures and the increase in main memory capacity have greatly improved the ability of modern server operating systems (OS) such as Linux to deploy a large number of concurrent application services on a single server instance. Such deployments become increasingly common as large data centers and cloud-centric application services become more popular.

The emergence of application containers (Docker Developers, 2016; Linux Containers Developers, 2016), introduction of support for kernel namespaces (Linux Programmer’s Manual, 2016a), allows a set of loosely coupled service components to be executed in isolation from each other and also from the main operating system. Executing the individual service components in isolated containers has its benefits. If a single containerized application runtime is compromised by the attacker, the attack surface is limited in its scope to a single component. Moreover, such an approach also simplifies the management and provision of service components (Belyaev and Ray, 2016a). Even in traditional UNIX environments, applications can be deployed in isolated containers, such as chrooted jails (Kamp and Watson, 2000; Solaris Zones Developers, 2016), for reasons of security and performance, and each application can access files only in some designated directory tree. We need to provide such applications or service components running in isolated environments with a privilege to access the
system resources on a need to know basis and also provide mechanism for controlled sharing and coordination among components running in containerized environments.

In our earlier works (Belyaev and Ray, 2016a, 2016b), we proposed an access control mechanism that provides each component with access to OS resources on a need to know basis and also allows controlled sharing and coordination among components. Specifically, we advocated the idea of a reference monitor (Krohn et al., 2007) which we refer to as the Linux Policy Machine (Belyaev and Ray, 2016a) middleware (and shorten it to just LPM in the rest of the paper) to address the identified challenges. Our LPM allows the management and enforcement of access to OS resources for individual service components and it also allows regulated inter-component communication. In a subsequent work (Belyaev and Ray, 2017), we provided further details on the performance of our communication architecture. Our current work extends our earlier results in several ways. First, we formalize our ideas in a more rigorous manner. Second, we provide details pertaining to the formulation of access control policies, demonstrate how the policies are stored and referenced in the context of our framework.

In this work, we formalize two types of policy classes, namely, capabilities classes and communicative classes. Capabilities class helps us categorize applications on the basis of their needs to system resources. Each capability class is associated with a set of Linux capabilities (Linux Programmer’s Manual, 2016b). The capabilities classes differ from each other on the basis of capabilities they possess. Each service component is placed in at most one capabilities class. The OS resources the component can access depends on the Linux capabilities associated with that class. Communicative classes allow controlled sharing and coordination of components across different isolated environments. Components that need to communicate are placed in a communicative class and their allowed interactions are outlined. Specifically, it enumerates which two components in a communicative class can coordinate with each other. The communicative class also specifies the data of a component that can be shared with another component belonging to the same communicative class.

We also provide the implementation details for the policy classes which forms the core of our LPM. Specifically, we provide the details about how the policy classes are created and their instances stored. The implementation of communicative classes is adapted from the generative communication paradigm introduced by Linda programming model (Gelernter, 1985) which uses the concept of tuple spaces for process communication. The traditional tuple space is implemented as a shared space in the main memory which allows asynchronous communication – this lacks any security features and also have operational limitations (Belyaev and Ray, 2016a, 2016b). We, on the other hand, implement tuple space on the filesystem of an individual component. The communication between service components is mediated through a module of the LPM called a Tuple Space Controller (TSC) which is allowed limited access to a component’s tuple space. TSC performs the coordination and collaboration between components using well designed tuple space transactions. Note that the TSC allows only those coordinations and collaborations that are specified in the communicative policy classes.

Our uniform framework provides a coherent business logic interface for administrative personnel to manage access control for a set of components both at the level of OS resources and at the level of inter-component interaction. Such a centralized LPM construct that is resident in user-space acts as a reference monitor (Krohn et al., 2007) for a set of administered application services deployed on a single Linux server instance. We incorporated the access control modeling and decision control in user-space with robust and expressive persistence layer. This allows high interoperability and usage of the framework on any general-purpose Linux OS without a requirement for custom kernel patching (Krohn et al., 2007).

The rest of the paper is organized as follows. Section 2 gives an overview and formalizes the framework, including the two types of policy classes. Section 3 provides the details of the inter-component communication architecture. Section 4 provides the semantics of tuple space transaction between service components. Section 5 gives the system architecture. Section 6 provides an overview of the database layer used to store the access control records. Section 7 gives samples of access control policies used in practice. Section 8 provides details on security of our framework. Section 9 demonstrates the feasibility of our approach by describing the implementation experience with a focus on tuple space paradigm. Section 10 gives additional information on the architectural properties of our framework. Section 11 covers a large body of related work. Section 12 concludes the paper.

2. Component-oriented access control framework

We first present a real-world motivating example where a single application/data service can consist of several components, each deployed in isolation (Belyaev and Ray, 2016a, 2016b).

2.1. Motivating example

Consider a real-world service deployment scenario illustrated in Fig. 1 that is taken from the actual telecom service provider (n-Logic Ltd., 2016). A Linux server has three applications, namely, Squid Web Cache Server, Squid Log Analyzer, and HTTP Web Server, deployed in three separate isolated environments (chrooted jail directories), each under a distinct unprivileged user identifier (UID). Combined all three applications represent individual components of a single service – ISP web caching that caches Internet HTTP traffic of a large customer base to minimize the utilization of ISP’s Internet backbone. Squid Web Cache Server component needs access to network socket I/O resources and some advanced networking capabilities of the Linux kernel (Linux Kernel Developers, 2017) in order to operate. Ideally, we would like to give this component only the set of capabilities needed to accomplish its task and not confer the super-user privileges.

In addition to providing the least set of capabilities to each component, we must also allow controlled sharing and coordination among them as the example illustrates. Squid Web Cache Server component generates daily operational cache logs in its respective runtime environment. Squid Log Analyzer...
component needs to perform data analytics on those operational log files on a daily basis. It then creates analytical results in the form of HTML files that need to be accessible by the HTTP Web Server component to be available through the web browser for administrative personnel. Note that the service components may not have access to a common centralized database management or a key-value store for the purpose of communication. Usual Inter-Process Communication (IPC) primitives such as message queues, memory-mapped files and shared memory may cause unauthorized access or illegal information flow. In fact, IPC could be disabled to minimize such security breaches (Belyaev and Ray, 2016a; Kamp and Watson, 2004). Moreover, legacy byte-level constructs, such as IPC, often do not fit under the development framework of modern applications that operate at the granularity of higher-level message objects. In such enterprise environments, a novel paradigm is needed for efficient and secure communication among isolated processes. Such coordination and sharing of data must take place in a controlled manner.

### 2.2. Component formalization

We begin by providing a definition of a component and then give the formalisms for the notion of policy classes that forms the basis of our access control framework.

#### Definition 2.1. [Component]

A component is an application or a micro-service process in the Linux environment executing with non super-user privileges (UID ≠ 0) and having access to program executable resources such as memory, network resources, CPU cycles, and files in its directory structure.

Let \( C = \{c_1, c_2, \ldots, c_n\} \) where \( C \) is the set of components and \( c_j \) denotes a component where \( 1 \leq j \leq n \). Note that two components \( c_i \) and \( c_j \) where \( 1 \leq i, j \leq n \) may have the same UID if they are executing in the same isolated containerized environment and will have access to the same set of resources.

In the following subsections we describe how we confer fine-grained access control to a component with regard to accessing system resources through the policies of the component’s capabilities class. We then describe how components belonging to isolated containers can coordinate their execution and get access to each others’ resources through the policies specified in the component’s communicative policy class.

#### 2.3. Capabilities policy class

The individual containerized components of an application service may need access to OS resources. In the Linux environments, the application runtime access control to the underlying OS resources has been traditionally regulated by root privileges which provides all permissions on system and user resources. The applications regulated by root privileges run with a special user identifier (UID = 0) that allows them to bypass access control checks. However, giving root permissions to an application violates the principle of least privilege and can be misused.

Subsequently, in Linux kernels starting from version 2.1, the root privilege was partitioned into disjoint capabilities (Linux Developers, 2016). Capabilities provide more fine-grained access control and allow a subset of root privileges to be granted to processes accessing some OS resources. For example, a web server daemon needs to listen to port 80. Instead of giving this daemon all root permissions, we can set a capability on the web server binary, like \( \text{CAP_NET_BIND_SERVICE} \) using which it can open up port 80 (and 443 for HTTPS) and listen to it, like intended. Some capabilities, such as \( \text{CAP_CHOWN} \) and \( \text{CAP_DAC_OVERRIDE} \), are very powerful; the former allows the components possessing
it to change file UIDs and GIDs and the latter allows the components to bypass access control checks.

The new emerging concept of Linux application containers such as Docker service (Docker Developers, 2016) and CoreOS (CoreOS Developers, 2016) relies on the Linux capabilities model. Despite the incorporation of capabilities in mainstream Linux and application containers, capabilities management in user-space is challenging (Hallyn and Morgan, 2008) and has been only sufficiently addressed in our work (Belyaev and Ray, 2016a). This is partly because kernel-space capabilities amendment does not provide persistent storage of state change in regard to capabilities being removed or added to a particular application. Consequently, such a deficiency leads to inability to track, identify and manage individual capabilities and their sets for a large number of applications and service components in a timely manner.

We now describe how to manage Linux capabilities using the notion of a capabilities class that we describe below.

Definition 2.2. [Capabilities Policy Class:] A capabilities policy class is one whose members are components that have access to the same Linux capabilities. Each capabilities policy class has an attribute called capability that lists the Linux capabilities associated with the class.

Let \( CP = \{cp_1, cp_2, \ldots, cp_m\} \) be the set of capabilities policy classes. Let \( cp_i(cap) \) be the set of Linux capabilities associated with class \( cp_i \). No two distinct capabilities classes can have the same set of Linux capabilities, that is, for \( i \neq j \), \( cp_i(cap) \neq cp_j(cap) \). Each capabilities policy class \( cp_i \) can have one or more components denoted as \( cp_i = \{c_{i1}, c_{i2}, \ldots, c_{im}\} \). Components \( c_{ij} \) belonging to the same capability class \( cp_i \), can access the set of OS resources as defined by \( cp_i(cap) \). Each component can belong to at most one capabilities class, that is, \( c_{ij} \in cp_{i} \Rightarrow c_{ij} \not\in cp_{j} \) where \( 1 \leq i \leq m \).

The following high-level operations are supported by our capabilities class model:

1. create a capabilities policy class,
2. add/remove capabilities to/from a policy class,
3. show capabilities in a policy class,
4. add/remove components to/from a policy class, and
5. show/count components in a policy class.

Note that the necessity to associate a single service component with at most one capabilities class is driven by the fact that a component in such a policy class inherits all the capabilities associated with such a class and can therefore access the same set of OS resources (Belyaev and Ray, 2016a). Since a single capabilities policy class can have multiple components, we can place all the service components into a single class depending on the service category (Belyaev and Ray, 2017). For instance, all components that require access to specific network socket I/O can be placed into a single capabilities class that grants them such a capability. Some service categories may require access to more capabilities than others. For instance, a web server may belong to a separate class that delegates a single \( \text{CAP_NET_BIND_SERVICE} \) capability to gain access permission to network sockets while a Network Time Protocol (NTP) server requires both \( \text{CAP_NET_BIND_SERVICE} \) capability as well as \( \text{CAP_SYS_TIME} \) capability to change system time on the machine. In such cases, we place the two components into different capabilities classes based on the principle of least privilege.

Such a separation is further supported by the fact that the capabilities enforcement is expressed through the notion of capabilities bits assigned to a binary on the filesystem (Belyaev and Ray, 2016a; Linux Programmer’s Manual, 2016b) – OS kernel cannot assign overlapping sets of capabilities to a single binary without overwriting some of them. Thus, each component has to belong to a distinct capabilities class in accordance with service categorization (classification).

2.4. Communicative policy class

In order to coordinate and collaborate across components executing in different isolated environments, we introduced the notion of a communicative class.

Definition 2.3. [Communicative Policy Class:] A communicative policy class is one whose members are components belonging to isolated environments that need to communicate to provide some service.

Let \( CM = \{cm_1, cm_2, \ldots, cm_t\} \) be the set of communicative policy classes. Each communicative policy class \( cm_i \) consists of two or more components that together offer some higher level service; this is denoted as \( cm_i = \{c_{i1}, c_{i2}, c_{i3}, \ldots, c_{it}\} \). Each component \( c_i \) cannot be a part of multiple communicative classes, that is, \( c_i \in cm_i \Rightarrow c_i \not\in cm_j \) where \( i \neq j \). If component \( c_i \) does not need to coordinate or collaborate with other components in different isolated environments, then it may not belong to any communicative class, denoted as \( c_i \not\in cm_i \) where \( 1 \leq i \leq t \).

Our notion of communicative class is different from the conventional notion of UNIX groups. The privileges assigned to a UNIX group are applied uniformly to all its members. Communicative class, on the other hand, allows both coordination and controlled sharing of private data objects of a container with other members. Such sharing of private data objects is done on a need-to-know basis and may be unidirectional.

Communicative classes are designed to support the following communication patterns among the components.

Coordination – Components forming part of a single service but belonging to different containerized environments may need an exchange of messages to perform coordinated invocation or maintain collective state (Belyaev and Ray, 2016a).

Collaboration – Components of a single data service may need to access data or runtime file objects to collaborate and perform joint or codependent measurements or calculations as illustrated in the description of the web caching service. Empowering a service component with the ability to obtain a replica of a data object that belongs to another component of a service in the same communicative class makes such collaboration possible.

The following high-level operations are supported by our communicative policy class model:

1. create a communicative policy class,
2. add/remove components to/from a communicative policy class,
A communicative policy class can be classified as a coordinative class or a collaborative class depending on whether it needs to coordinate or share data with other components. Coordinative class contains a set of coordination policies and a collaborative class contains a set of collaboration policies. Let \( CR \) and \( CC \) be the set of coordinative classes and collaborative classes respectively. Note that the set of communicative classes comprise the set of coordinative class and the set of collaborative class, that is, \( CM = CR \cup CC \). Moreover, if a communicative class \( cm \) supports both coordination and collaboration, then \( cm \in CR \cap CC \).

We now give the details of our access control policies in coordinative and collaborative policy classes. The access control rules for the coordinative policy class is defined by a partial function \( AF_c \) that takes in two components as arguments and returns either 1 or 0 signifying whether the coordination is allowed or prohibited. \( AF_c(c_1, c_2) = 1 \) signifies that components \( c_1 \) and \( c_2 \) are allowed to coordinate with each other; otherwise, such coordination is prohibited. Note that this function is commutative. As per the rules of the coordinative policy class, the two components \( c_1 \) and \( c_2 \) can coordinate only if they belong to some coordinative policy class \( cp \). This is denoted as \( AF_c(c_1, c_2) = 1 \) only if \( c_1, c_2 \in cp \), where \( cp \in CR \).

Next, we describe the access control rules for the collaborative policies. The collaborative policies are more complex. The components must be part of the same collaborative class and also there must be an explicit permission that gives a component access to replicas of specific objects in the collaborative class. Each private object \( o_i \) is owned by some component which is given by the function \( own(o_i) \). \( own(o_i) = c_o \) denotes that \( o_i \) is the private object of component \( c_o \). Let \( P \) be the set of explicit permissions, denoted as \( P = \{p_1, p_2, \ldots, p_n\} \), that grants components access to private object’s replica belonging to other components. Each permission \( p_i \in P \) is a tuple of the form \( \langle c_o, cm, o_i \rangle \) which denotes that component \( c_o \) has permission to read replica of object \( o_i \) that is owned by component \( c_m \). We are now ready to describe the access control function \( AF_c \). This is a partial function that takes in two arguments, namely, a component and a data object and returns 1 or 0 signifying whether such access is allowed or prohibited. \( AF_c(c_m, o_i) = 1 \) signifies that \( c_m \) is allowed to read a replica of \( o_i \); otherwise, such read is prohibited. The component \( c_m \) can read object \( o_i \) only by component \( c_m \), only if \( c_m \) and \( o_i \) belong to some collaborative policy class \( cl \) and there is a permission that allows \( cl \) access to \( o_i \) that is owned by \( c_m \). Formally, \( AF_c(c_m, o_i) = 1 \) only if \( \langle c_m, cm, o_i \rangle \in P \) where \( own(o_i) = c_m \) and \( c_m, c_m, o_i \in cl \) where \( cl \in CC \).

### 3. Communication architecture

We now discuss the enforcement architecture for communicative class model. We adapt the generative communication paradigm introduced by Linda programming model (Gelernter, 1985) that was proposed for communication among multiple threads of a single application. In this approach, processes communicate indirectly by placing tuples in a tuple space that is a shared main memory address space from which other processes can read or remove them. This programming model allows decoupled interaction between processes separated in time and space: communicating processes need not know each other’s identity, nor have a dedicated connection established between them (Vitek et al., 2003). In comparison to general-purpose message-passing that provides a rather low-level programming abstraction for building distributed systems and enabling inter-application interaction, Linda, instead, provides a simple coordination model with higher level of abstraction that makes it very intuitive and easy to use (Buravlev et al., 2016).

The lack of any protection mechanism in the basic model (Minsky et al., 2000; Vitek et al., 2003) makes the single global shared tuple space unsuitable for interaction and coordination among untrusted components. Consequently, we adapt the tuple space model that will satisfy our requirements for secure and reliable communication between service components within a single communicative policy class (Belyaev and Ray, 2016a). Each component in our model is associated with its own private tuple space which is implemented as an abstraction in the form of a filesystem directory.

We propose a tuple space calculus that is compliant with the base model introduced in Gelernter (1985) but is applied on dedicated tuple spaces of individual components instead of a global space. Our tuple space calculus comprises the following operations which are implemented in our Tuple Space Library (TSL):

1. create tuple space operation,
2. delete tuple space operation – Deletes tuple space only if it is empty,
3. read operation – Returns the value of individual tuple without affecting the contents of a tuple space,
4. append operation – Adds a tuple without affecting existing tuples in a tuple space, and
5. take operation – Returns a tuple while removing it from a tuple space.

We adhere to the immutability property – tuples are immutable and service components can either append or remove tuples in a tuple space without changing contents of individual tuples. The components and the LPM are allowed to invoke operations of the Tuple Space Library. A component is allowed to perform all the described operations in its own tuple space while LPM is restricted to read and append operations only. A component cannot perform any operation on another component’s tuple space. The take operation is the only manner in which tuples get deleted from a tuple space because the delete tuple space operation is allowed only on an empty tuple space. Note that the components may need to be modified in order to utilize the tuple space communication.

The LPM plays a mediating role in the communication between service components. The communication takes place through two types of tuples: control tuples and content tuples.
Control tuples may carry messages for coordination or requests for sharing. Content tuples the mechanism by which data get shared across service components. In order to avoid denial of service attacks, at most one control tuple and one content tuple may be appended into a component’s tuple space at any given time. The LPM polls using a round robin algorithm for control tuples in the tuple spaces for components that are listed in communicative policy classes to see if they wish to coordinate or collaborate.

The structure of the tuples is shown in Fig. 2. Control tuples are placed by a component into its tuple space for the purpose of coordination or for requesting data from other components. A control tuple has the following fields:

1. **Source ID** – This is specified as an absolute path of the component and it acts as the identifier of the communication initiator.
2. **Destination ID** – This is specified as an absolute path of the component and it acts as the identifier of the communication recipient.
3. **Type** – This indicates whether it is a collaborative or coordinative communication.
4. **Message** – This contains the collaborative/coordinative information. For collaboration it is the request for an absolute path of data object. Coordination message may be opaque or even encrypted as other entities may be oblivious of this inter-component communication.

Content tuples are used for sharing data objects across service components and they have the following fields:

1. **Destination ID** – This is specified as an absolute path of a service component and it denotes the ID of the recipient component.
2. **Sequence Number** – This indicates the sequence number of a data object chunk that is transported. ASCII objects in the form of chunks are the primary target of inter-component collaboration.
3. **Payload** – This contains the chunk of a data object. Content tuples are placed by the LPM reference monitor into corresponding tuple space of the requesting component that needs to receive content. Note that content tuples are designed for collaboration only.

Containerized service components are often not aware of whether they are deployed in an isolated runtime environment, such as a chrooted jail or not. Therefore, tuple fields, such as Source/Destination IDs and object paths that technically require the absolute path to the object on the filesystem, can be substituted with the isolated environment ID, such as a container ID. This permits the service deployment with individual components that are only aware of immediate containerized path locations or corresponding components’ service identifiers. For instance, the containerized identifier, such as /100/opt/bin/service-component-2, can be mapped to a system-wide path of /opt/containers/container-100/opt/bin/service-component-2 by the LPM reference monitor with a proper support for such a composite service mapping.

4. **Tuple space transactions**

We provide the sample transactional flow involved in tuple space operations, necessary to carry out collaborative and coordinative types of communication between isolated service components. Since loosely coupled processes cannot communicate directly due to isolation properties, the flow is conducted indirectly via the Tuple Space Controller (TSC) (Belyaev and Ray, 2016b).

4.1. **Coordinative transaction**

Coordinative communication between two components is depicted in Fig. 3. Intrinsically, coordination is bidirectional, since both endpoints need to obtain coordinative messages. Both
components need to create the corresponding tuple spaces in the isolated runtime environments. In the first phase, Component 1 delivers a message to Component 2.

- **[Step 1]**: Component 1 appends a control tuple (see the structure of tuples in Fig. 2) to its tuple space TS 1. This control tuple has to be subsequently delivered to Component 2.
- **[Step 2]**: TSC reads the control tuple from TS 1.
- **[Step 3]**: Component 1 retracts the control tuple via the take operation.
- **[Step 4]**: TSC appends the control tuple into tuple space TS 2 of Component 2.
- **[Step 5]**: Component 2 takes the appended control tuple from its tuple space TS 2.

In the next phase of coordinative communication, Component 2 has to deliver its coordination message to Component 1. Such a message could contain independently new coordinative information, or serve as the acknowledgment for the control tuple that has just been received. Such a decision is service-specific. However, we require that coordinative transactional flow is terminated through such a confirmatory control tuple from Component 2. The steps in the second phase are described next.

- **[Step 6]**: Component 2 appends a control tuple to its tuple space TS 2. This control tuple (denoted as message B) has to be subsequently delivered to Component 1.
- **[Step 7]**: TSC reads the control tuple from TS 2.
- **[Step 8]**: Component 2 retracts the control tuple via the take operation.
- **[Step 9]**: TSC appends the control tuple into tuple space TS 1 of Component 1.
- **[Step 10]**: Component 1 takes the appended control tuple (message B from Component 2) from its tuple space TS 1. This step completes the coordinative transaction.

Note that the coordination messages may be of any type. Therefore, our communication architecture allows full transparency in inter-component exchange and does not require proprietary formats. Most common formats that may be incorporated into the message field of a control tuple is XML, JSON or text strings. Such a choice is service-dependent. Moreover, the service components may utilize the serialization libraries such as XStream (XStream Developers, 2016) to represent class objects in the form of XML messages. In this case, isolated components that use our TSL library can perform complete object-based transport within a single service solely through provided tuple space communication.

### 4.2. Collaborative transaction

Collaborative communication is depicted in Fig. 4. Intrinsically, collaboration is unidirectional, since the workflow is only directed from a single requester to TSC and back in the form of content tuples. In contrast to a control tuple, a content tuple only has a Destination ID field, as depicted in Fig. 2. However, at the level of service logic, collaboration flow may very well be bidirectional. Both endpoints may obtain replicas of mutual data objects through TSC, if such sharing is explicitly permitted by collaboration policies. Such a scenario of collaboration is depicted in Fig. 4. The steps of collaborative transaction, on the left, are shown below.

- **[Step 1]**: Component 1 appends a control tuple to its tuple space TS 1 with indication of request for data object that is owned by Component 2.
- **[Step 2]**: TSC reads the control tuple from TS 1.
- **[Step 3]**: TSC reads the requested data object on the filesystem. Note that this step is not a part of the actual transactional flow, but represents the internal operations of TSL.
- **[Step 4]**: TSC appends the replica of a data object into tuple space TS 1. Note that if the size of the data object is greater than the size of the tuple, it may need to be fragmented and each fragment is loaded one tuple at a time. Moreover, the TSC can append the next content tuple only after the current one is taken from a tuple space.
[Step 5:] Component 1 takes appended content tuples, one tuple at a time and reconstructs the replica of the data object.

[Step 6:] Component 1 takes a control tuple from its tuple space TS 1. This step completes the collaborative transaction.

The flow of second collaborative transaction, on the right, is identical. Note that the communication starts with creation of a tuple space and ends with its deletion when the transactional flow completes.

4.3. Transactional API

The complexity for both types of communication is hidden from the components. TSL provides public Application Programming Interface (API) methods without exposing internal operations of tuple space calculus (Belyaev, 2016a).

The main API methods for tuple space transactions are depicted in Fig. 5. TSC executes the implementation of the ControllerTransactionManager class while the service component executes the implementation of the AgentTransactionManager class within the TSL library. ControllerTransactionManager implementation has the following public methods:

- `facilitate_BidirectionalPersistentCoordinativeTransaction()` – Performs the exchange of control tuples between corresponding tuple spaces of service components. The implementation of this method uses the private `facilitate_UnidirectionalPersistentCoordinativeTransaction()` method to append control tuples to individual tuple spaces involved in the coordination.
- `facilitate_PersistentCollaborativeTransaction()` – Performs the replication of a data object requested in the collaborative request issued by the component. AgentTransactionManager implementation has the following public methods:
  - `perform_ActivePersistentCoordinativeTransaction()` – Initiates a start of coordinative transaction by appending the initial control tuple in its own tuple space.
  - `perform_PassivePersistentCoordinativeTransaction()` – Initiates the ending of coordinative transaction by waiting for a control tuple from the counterpart component.
  - `perform_PersistentCollaborativeTransaction()` – Initiates and completes the collaborative transaction by assembling the replica at component’s end.
  - `get_ReplyControlTuple()` – Obtains the control tuple that has been appended by the TSC in its tuple space from the counterpart component.

Therefore, a single coordinative transaction (control flow) depicted in Fig. 3 that utilizes a set of operations of tuple space calculus is actualized through the invocation of the `facilitate_BidirectionalPersistentCoordinativeTransaction()` method within ControllerTransactionManager instance in TSC. It is coupled with the invocation of `perform_ActivePersistentCoordinativeTransaction()` method within AgentTransactionManager implementation on a side of the component that initiates coordination. It is then followed by the invocation of `perform_Passive-PersistentCoordinativeTransaction()` method within AgentTransactionManager implementation on a side of the component that receives initial coordination message.

Consequently, a single collaborative transaction (data flow) depicted in Fig. 4 that utilizes a set of operations of tuple space calculus is implemented through the invocation of the `facilitate_PersistentCollaborativeTransaction()` method within ControllerTransactionManager instance in TSC. It is coupled with the invocation of `perform_PersistentCollaborativeTransaction()` method within AgentTransactionManager implementation on a side of the component that requests a replica of a data object.

5. System architecture

LPM acts as a centralized enforcement point and a reference monitor for the application services deployed on the single OS server instance. The unified framework uses the embedded SQLite database library to store and manage policy classes abstractions and their policy records. The usage of embedded

Please cite this article in press as: Kirill Belyaev, Indrakshi Ray, On the formalization, design, and implementation of component-oriented access control in lightweight virtualized server environments, computers & security (2017), doi: 10.1016/j.cose.2017.06.004
database facility eliminates the dependency on a separate database server that is prone to potential availability downtimes and security breaches (Belyaev and Ray, 2016a). The LPM implemented in Java Standard Edition (SE) is deployed under unprivileged UID with elevated privileges using Linux capabilities within the same OS outside the containerized environments such as chrooted jails and application containers. Fig. 6 illustrates the components of the LPM. These are described below.

- **User Interface Layer:** This layer provides operator with command-line interface (CLI) to issue commands to manage the framework.
- **Parser Layer:** This layer parses the user input from the CLI shell and forwards the parsed input to the underlying layers for execution.
- **Enforcer Layer:** This layer enforces the capabilities on the given application using Linux LibCap (Linux Programmer’s Manual, 2016b) library and grants/denies access to OS resources depending on the capabilities class associated with the component. The layer also integrates a TSC (Minsky et al., 2000) responsible for tuple space operations for the enforcement of collaboration and coordination of service components in a single communicative class.
- **Persistence Layer:** This layer provides the Create/Read/Update/Delete (CRUD) functionality to manage policy records using tuple space transactions API.

**Fig. 5 – Tuple space transactions API.**

**Fig. 6 – LPM architecture.**
using embedded database facilities. This layer is discussed in detail in Section 6.

System implementation features numerous classes and packages associated with corresponding layers, with overall volume of current production code exceeding 12,000 lines, not counting the unit tests per individual class (Belyaev, 2016a).

6. Overview of the policies store

Persistence Layer of our LPM reference monitor provides the Create/Read/Update/Delete (CRUD) functionality to manage records using embedded database facilities. We now provide detailed description of the schema, the business logic, the relationship between main tables and the possible access control records involved in our component-oriented access control model. The schema of the embedded database (referred to as the Policies Store) for storing framework’s access control policies appears in Fig. 7. It shows the schema for three tables, namely, Components Table, Capabilities Classes Table, and Communicative Classes Table. We begin by describing the attributes of the Components Table that is described below.

- COLUMN\_COMPONENT\_PATH\_ID – Absolute path to the location of the service component is currently used as the component ID.
- COLUMN\_COMPONENT\_DESC – Description of the service component and its functionality.
- COLUMN\_COMPONENT\_ID – Reserved for future use – may be used for alternative ID mechanisms.
- COLUMN\_COMPONENT\_COMMUNICATIVE\_CLASS\_ID – A reference to the ID of the Communicative Class in which the component may belong.
- COLUMN\_COMPONENT\_CAPABILITIES\_CLASS\_ID – A reference to the ID of the Capabilities Class in which the component may belong.
- COLUMN\_COMPONENT\_CONTAINER\_ID – Reserved for future use with various container technologies.
- COLUMN\_COMPONENT\_TUPLE\_SPACE\_PATH – Reserved for possible use. It indicates the component’s tuple space location. Technically this should be determined automatically based on the component’s path ID – TS should be located immediately at the 1st level of component’s root directory.
- COLUMN\_STATUS – Not currently used but may be employed for specification of whether a corresponding row in the database table is active or not from the access control standpoint. Such a necessity may arise in the future when a record may still be kept in PS without a need to physically delete it. Therefore, changing its status from active to inactive may be beneficial for such an optimization of PS operation.
- UPDATE\_COLUMN – Used by Persistence Layer to indicate which field to update when calling write\_ComponentsTableRecord() method that belongs to Persistence Layer (Belyaev, 2016a).

The schema of the Communicative Classes Table is given below.

- COLUMN\_CLASS\_ID – Unique Communicative Class ID
- COLUMN\_CLASS\_NAME – Name of a Class

![Fig. 7 – Database schema of persistence layer.](image-url)
• **COLUMN_COORDINATION_RECORD** – A pair: COMPONENT\_1\_PATH_ID; COMPONENT\_2\_PATH_ID. If such a record exists, that signifies a permission for coordination
• **COLUMN_COLLABORATION_RECORD** – A pair: COMPONENT\_1\_PATH_ID; Data\_Object\_Absolute\_Path. If such a record exists, that signifies a permission for collaboration
• **COLUMN_STATUS** – Not currently used but may be employed for specifying whether a corresponding row in the database table is active or not from the access control standpoint. Such a necessity may arise in the future when a record may still be kept in PS without a need to physically delete it. Therefore, changing its status from active to inactive may be beneficial for such an optimization of PS operation.
• **UPDATE_COLUMN** – Used by internal Persistence Layer to indicate which field to update when calling write\_CapabilitiesClassesTableRecord() method that belongs to Persistence Layer (Belyaev, 2016a).

We now give some examples of coordination and collaboration records.

**Example 6.1.** Coordination Record Example 1.

```
[/s/missouri/a/nobackup/services/containers/container-1/bin/component-publisher; /s/missouri/a/nobackup/services/containers/container-2/bin/component-generator].
```

**Example 6.2.** Coordination Record Example 2.

```
[/s/missouri/a/nobackup/services/containers/container-2/bin/component-generator; /s/missouri/a/nobackup/services/containers/container-1/bin/component-publisher].
```

**Example 6.3.** Collaboration Record Example 1.

```
[/s/missouri/a/nobackup/services/containers/container-2/bin/component-generator; /s/missouri/a/nobackup/services/containers/container-1/bin/component-publisher].
```

Note that both coordination records shown in Examples 6.1 and 6.2 should exist in the table to allow bidirectional coordination between two components via TSC. Any record input is checked for validity before insertion – corresponding component IDs and objects paths should exist in the system. Such conditions are validated by the corresponding implementation logic in Persistence Layer. It also ensures that there are no duplicate rows in corresponding tables.

The schema of the Capabilities Classes Table is given below:

```
[ COLUMN\_CLASS\_ID, COLUMN\_CLASS\_NAME, COLUMN\_CAPABILITIES, COLUMN\_STATUS, UPDATE\_COLUMN].
```

The description of individual columns is as follows:

• **COLUMN\_CLASS\_ID** – Unique Capabilities Class ID.
• **COLUMN\_CLASS\_NAME** – Name of a Class.
• **COLUMN\_CAPABILITIES** – A list of one or more Linux Capabilities assigned to a class.
• **COLUMN\_STATUS** – Not currently used but could be employed for specification of whether a corresponding row in the database table is active or not from the access control standpoint. Such a necessity may arise in the future when a record may still be kept in PS without a need to physically delete it. Therefore, changing its status from active to inactive may be beneficial for such an optimization of PS operation.
• **UPDATE\_COLUMN** – Used by internal Persistence Layer to indicate which field to update when calling write\_CapabilitiesClassesTableRecord() method that belongs to Persistence Layer (Belyaev, 2016a).

### 7. Overview of the policies formulation

The access control policies for the introduced framework are managed via the Domain Specific Policy Language (DSPL) within LPM (a part of the Parser Layer depicted in Fig. 6) that allows the formulation of various component-oriented policies in a human, rather than machine-friendly form (Badger et al., 1995; Belyaev and Ray, 2016a). The language allows to perform Create/Read/Update/Delete (CRUD) functionality on individual policies and create both capability as well as communicative policy classes in the Policies Store (PS). In this section we demonstrate that meaningful component-oriented policies can be expressed completely in a form simple and concise enough to be administered at a reasonable cost.

#### 7.1. Capabilities classes management

The following samples of policies illustrate how capabilities classes can be managed in practice via the CLI provided to the administrative personnel:

```
[ COUNT\_CAPABILITIES\_CLASSES ] – Show how many Capabilities Classes exist in the PS
[ SHOW\_CAPABILITIES\_CLASSES ] – Show individual Capabilities Classes in the PS
[ CREATE\_CAPABILITIES\_CLASS 1 application_service _with\_ID\_1\_class ] – Create Capabilities Class for a specific application service
[ SHOW\_CAPABILITIES\_CLASS\_CAPABILITIES 1 ] – Show all Linux Capabilities associated with a particular Capabilities Class
[ ADD\_CAPABILITIES\_CLASS\_CAPABILITY 1 CAP\_DAC _OVERRI SE ] – Add CAP\_DAC\_OVERRI SE Linux Capability to a particular Capabilities Class
[ REMOVE\_CAPABILITIES\_CLASS\_CAPABILITY 1 CAP\_CHOWN ] – Remove CAP\_CHOWN Linux Capability from a particular Capabilities Class
[ SHOW\_CAPABILITIES\_CLASSES ] – Show all available Linux Capabilities
[ COUNT\_CAPABILITIES\_CLASS\_COMPONENTS 1 ] – Show how many service components are associated with a particular Capabilities Class
[ SHOW\_CAPABILITIES\_CLASS\_COMPONENTS 1 ] – Show all service components associated with a particular Capabilities Class
[ MOVE\_COMPONENT\_TO\_CAPABILITIES\_CLASS /opt/ containers/service-100/bin/log-analyzer 1 ] – Move a service component to a particular Capabilities Class
```
7.2. Communicative classes management

The following samples of policies illustrate how communicative classes can be managed in practice via the CLI provided to the administrative personnel:

- \( \text{COUNT_COMMUNICATIVE_CLASSES} \) – Show how many Communicative Classes exist in the PS
- \( \text{SHOW_COMMUNICATIVE_CLASSES} \) – Show individual Communicative Classes in the PS
- \( \text{CREATE_COMMUNICATIVE_CLASS 1 web_caching_service_class} \) – Create Communicative Class for a specific application service
- \( \text{SHOW_COMMUNICATIVE_CLASS_COMPONENTS 1} \) – Show all service components associated with a particular Communicative Class
- \( \text{COUNT_COMMUNICATIVE_CLASS_COMPONENTS 1} \) – Show how many service components are associated with a particular Communicative Class
- \( \text{MOVE_COMPONENT_TO_COMMUNICATIVE_CLASS /opt/containers/service-100/bin/log-analyzer 1} \) – Move a service component to a particular Communicative Class
- \( \text{SHOW_COMMUNICATIVE_CLASS_COLLABORATION_POLICIES 1} \) – Show all collaboration records associated with a particular Communicative Class
- \( \text{SHOW_COMMUNICATIVE_CLASS_COORDINATION_POLICIES 1} \) – Show all coordination records associated with a particular Communicative Class
- \( \text{ADD_COMMUNICATIVE_CLASS_COLLABORATION_POLICY 1 component_pathID object_path} \) – Add a collaborative policy to a particular Communicative Class
- \( \text{ADD_COMMUNICATIVE_CLASSCOORDINATION POLICY 1 component_pathID_1 component_pathID_2} \) – Add a coordinate policy to a particular Communicative Class
- \( \text{REMOVE_COMMUNICATIVE_CLASS_COLLABORATION_POLICY 1 component_pathID object_path} \) – Remove a collaborative policy from a particular Communicative Class
- \( \text{REMOVE_COMMUNICATIVE_CLASSCOORDINATION POLICY 1 component_pathID_1 component_pathID_2} \) – Remove a coordinate policy from a particular Communicative Class

8. Security aspects

In this section, we discuss the security of our component-oriented access control framework. Security issues with respect to the capabilities class is rather straightforward. LPM is a trusted reference monitor and components are given the capabilities on a need-to-know basis and placed in the corresponding capabilities class. LPM creates the communicative class based on the components’ requirements to interact. Only members of the same communicative class can coordinate and/or share data. The access control for the collaborative class is regulated by the LPM – each request for a data object is checked to ensure that both components belong to the same collaborative policy class before granting them replication privileges.

A malicious component cannot impersonate another component and take over its tuple space as the absolute path of the component is used as the identifier. Removing it from the associated communicative class will disable the collaboration with other components. The removal also precludes such a component from consuming computational resources utilized by our LPM reference monitor that has to serve concurrent interactions for a set of multi-component application services hosted on a single OS instance.

Extra protection mechanisms are also incorporated for each component’s tuple space. The components in the same isolated environment have a directory structure within the filesystem for their individual tuple spaces. Each component creates its own tuple space in this directory structure. Only the individual component can perform all the operations, namely, create tuple space, delete tuple space, read, append, and take. The LPM can only perform read and append operations on the respective tuple space. Thus, no one other than the component itself can remove anything from its tuple space. Moreover, the confidentiality and integrity are guaranteed by preventing other components from directly accessing its tuple space. The availability of the tuple space is also guaranteed by the manner in which the tuple space library prevents a component from using all the allocated filesystem space in the directory structure of the isolated environment. Coordination requires writing a single control tuple. Collaboration requires writing multiple content tuples; however, the implementation of append operation ensures that such an operation writes only a single content tuple at a given time and the component has to take the tuple before a new one is written in its tuple space. Such a strategy avoids overconsumption of filesystem space, alleviates disk/filesystem access loads with large numbers of concurrent transactions, and also serves as an acknowledgment mechanism before the next chunk of the replicated data object is written (Belyaev and Ray, 2016b).

9. Experimental results

The deployment of component-oriented access control framework in the real-world settings requires a thorough performance evaluation. The model for capabilities classes does not incur any significant performance overheads for the unified framework. This is because its enforcement is based on the calls to the LibCap library (Linux Programmer’s Manual, 2016b) that essentially updates the filesystem capabilities metadata information for a process (Wright et al., 2002). Such operations do not incur the performance overheads because library mediations do not require extra disk I/O aside from the I/O load of the base system (Badger et al., 1995, 1996). There is also no additional memory utilization required aside from the RAM consumption by the LPM reference monitor itself (Belyaev and Ray, 2016a).

The initial prototype of the TSL implemented in Java SE is publicly available through the LPM’s GitHub repository (Belyaev, 2016a). The specification of the machine involved in the benchmarking is depicted in Table 1. Memory utilization and time information has been obtained using JVM’s internal Runtime and System packages. Due to space limitations, we do not provide the benchmarking results for coordinative transaction. Despite its implementation complexity, such a transaction involves only exchange of two control tuples and therefore does
not incur any significant performance overheads in terms of CPU and RAM utilization. The unit test for coordination is also available at GitHub (Belyaev, 2016b).

For collaboration, the payload of individual content tuple is set at 1 MB. Therefore, for instance, it takes 64 content tuples to replicate a 64 MB data object. Six sizes of data objects have been chosen – 64, 128, 256, 512, 1024 and 2048 MB objects respectively. Collaborative transactional flow, as discussed in Section 3, is performed on the EXT4 filesystem, where the requesting service component creates a tuple space in its isolated directory structure and assembles the content tuples appended by the TSC into a replica in its isolated environment outside the tuple space directory.

Replication performance for sequential collaboration is depicted in Fig. 8. The create_ObjectReplica() method in Utilities package of the TSL library is a reference method that sequentially executes the collaborative transaction conducted between TSC and the service component within a single thread of execution. We can observe that the replication time progressively doubles with an increase of the object size. On average, it takes 0.625 second to replicate a 64 MB object, 1.065 seconds a 128 MB object, 1.955 seconds a 256 MB object, 3.950 seconds a 512 MB object, 8.550 seconds a 1024 MB object and 17.505 seconds to replicate a 2048 MB object. Java Virtual Machine (JVM) memory utilization during sequential collaboration has been observed to be negligible. That is largely due to the usage of Java NIO library (Java NIO Developers, 2016) in our Utilities package that is designed to provide efficient access to the low-level I/O operations of modern operating systems. On average, memory usage is 23 MB for replication of a 64 MB object, 34 MB for a 128 MB object, 56 MB for a 256 MB object, 305 MB for a 512 MB object (an outlier, repeatedly observed with this object size that might be specific to the garbage collector for this particular JVM), 58 MB for a 1024 MB objects, and 36 MB for replication of a 2048 MB object. Note that since the measured JVM memory utilization takes into account the processing of both TSC and requester components within a single thread of execution, the actual JVM utilization will be roughly twice lower for two endpoints in the collaborative transaction when endpoints execute in separate JVMs. This shows the practical feasibility of our collaborative implementation even for replication of large data objects. According to the obtained results, we can anticipate that TSC can handle a large number of concurrent collaborative transactions without consuming significant amounts of physical RAM. We observed partially full utilization of a single CPU core during replication of the largest data object (2048 MB). The unit test for sequential collaboration is available at GitHub (Belyaev, 2016c).

In real-world settings TSC and service component execute concurrently in separate threads perhaps in different JVMs. Replication performance for concurrent collaboration is depicted in Fig. 9, where TSC and service component execute as concurrent threads in a single JVM. In such settings, TCS thread performs a short sleep in its section of TSL library after every append operation to allow the service component thread to take a content tuple from its tuple space. That results in a longer replication time compared to sequential execution depicted in Fig. 8. Due to concurrent execution, two CPU cores have been partially utilized by the JVM during concurrent collaboration. The obtained results show that replication time is sufficient

<p>| Table 1 – Node specifications. |</p>
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel(R) Xeon (R) X3450 @ 2.67 GHz; Cores: 8</td>
</tr>
<tr>
<td>Disk</td>
<td>SATA: 3 Gb/s; RPM: 10,000; Model: WDC; Sector size: 512 bytes</td>
</tr>
<tr>
<td>Filesystem</td>
<td>EXT4-fs; Block size: 4096 bytes; Size: 53 GB; Use: 1%</td>
</tr>
<tr>
<td>RAM</td>
<td>8 GB</td>
</tr>
<tr>
<td>OS</td>
<td>Fedora 23, Linux kernel 4.4.9–300</td>
</tr>
<tr>
<td>Java VM</td>
<td>OpenJDK 64-Bit Server SE 8.0_92</td>
</tr>
</tbody>
</table>
for non-critical, non-real-time services where medium-sized data objects need to be replicated across service components. Further decrease in replication time is possible through the usage of faster storage media, such as Solid-State Drives (SSDs) and Non-Volatile Memory (NVM) (Chen et al., 2016). Again, we can observe that the replication time progressively doubles with an increase of the object size. On average, it takes 17.152 seconds to replicate a 64 MB object, 23.8 seconds a 128 MB object, 37.1 seconds a 256 MB object, 117.5 seconds a 1024 MB object and 246.505 seconds to replicate a 2048 MB object. In line with sequential collaboration, JVM memory utilization during concurrent collaboration also has been observed to be negligible. On average, memory usage is 7 MB for replication of a 64 MB object, 14 MB for a 128 MB object (an outlier, repeatedly observed with this object size that might be specific to the garbage collector for this particular JVM that is not related to the outlier depicted in Fig. 8 during sequential collaboration), 8 MB for a 256 MB object, 9 MB for a 512 MB object, 12 MB for a 1024 MB objects, and 19 MB for replication of a 2048 MB object. In fact, the utilization is much lower than in case of sequential collaboration. Again, when executed in separate JVMs, the memory footprint for every endpoint in the transactional flow will be further diminished. Therefore, TSC memory usage during real-world operations for handling multi-component collaborative transactions is expected to be minimal. Note that due to preliminary nature of conducted transactional benchmarks, the focus is on functionality, rather than availability. Therefore, no actual saturation of storage media has been attempted. The unit test for concurrent collaboration is available at: https://github.com/kirillbelyaev/tinypm/blob/LPM2/src/test/java/TSLib_UnitTests_Collaboration.java.

Table 2 – Server node specifications.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel(R) Xeon (R) E5520 @ 2.27 GHz; Cores: 8</td>
</tr>
<tr>
<td>Disk</td>
<td>SAS: 3 Gb/s; RPM: 15,000; Model: Fujitsu; Sector size: 512 bytes</td>
</tr>
<tr>
<td>Filesystem</td>
<td>EXT4-fs; Block size: 4096 bytes; Size: 40 GB; Use: 1%</td>
</tr>
<tr>
<td>RAM</td>
<td>24 GB</td>
</tr>
<tr>
<td>OS</td>
<td>Fedora 24, Linux kernel 4.7.5–200</td>
</tr>
<tr>
<td>Java VM</td>
<td>OpenJDK 64-Bit Server SE 8.0.102</td>
</tr>
</tbody>
</table>

Fig. 9 – Replication performance for concurrent collaboration.

9.1. Load simulation of tuple space controller

Our access control framework provides flexible types of information flow between infrequently communicating (not real-time) service components. Therefore, as already noted, it is not intended for High-Performance Computing (HPC) services and large data object transfers between components. Nevertheless, it is important to provide an accurate estimate of the resource consumption incurred by the TSC that is responsible for the information flow transport. In this part we provide the actual memory allocation (Resident Set Size (RSS)) info obtained through ps and htop utilities – the non-swapped physical memory that a replication task (a single collaborative transaction) has used. In contrast to the previously reported JVM memory allocation that is subject to arbitrary fluctuations related to garbage collection mechanism, RSS provides the real memory allocation estimates on the actual hardware. The TSC load simulation has been conducted on the enhanced server hardware. The specification of the machine is depicted in Table 2.

We have simulated the tentative resource consumption by the TSC conducting concurrent sequential collaborative DoS issues. In the future, we may consider to conduct explicit comparison of replication speeds between our solution and basic copy operation using cp command in user space.
transactions for a large number of service components – 1, 8, 16, 32, 64 and 128 concurrent transactions respectively, each executing in a separate JVM thread. Two object sizes have been used for replication – 64 MB for the lower bound and 2048 MB for the upper bound of the load assessment. No external processes unrelated to benchmarking were present during the simulation load on the system. Note that every individual experiment has been repeated for up to 10 times to verify the consistency of performance indicators. The unit tests for TSC load simulation are available at: https://github.com/kirillbelyaev/tinypm/blob/LPM2/src/test/java/TSLib_TSC_SequentialCollaboration_UnitTests.java.

We observed that the load has been evenly distributed on all CPU cores by the JVM and OS SMP facility and initially utilizes 100% on every core, gradually decreasing, as every thread executing collaborative transaction passes the peak work section. Our main focus is on RAM consumption since it is critically important to estimate its usage by potentially tens of concurrent transactions performed by TSC. The replication time and RSS info associated with a 64 MB data object is depicted in Fig. 10. For a single transaction executing within a single JVM thread the RSS is 159 MB with replication time at 0.78 s. For 128 concurrent transactions executing in independent JVM threads the cumulative upper bound for RSS is 3415 MB with last transaction to complete replication at the threshold of 110.858 s. The replication time nearly doubles with the proportional increase of the number of objects that need to be replicated. However, the memory usage does not generally double with load increase. The upper bound of 3415 MB represents only 14 to 15% of the available system memory depicted in Table 2. Therefore, the simulation shows that TSC may potentially be more memory-efficient on a larger scale with replication of small data objects for a large number of requesting service components.

The RSS info, associated with replication of a 2048 MB data object, is depicted in Fig. 11. Due to limitations of disk partition size we have not been able to run the 32, 64 and 128 thread experiments until completion to record the final replication time. The experiments terminate at the point of partition space saturation. However, two key indicators have been observed: peak RSS shows the highest memory allocation observed for a repeated number of simulations; End of Execution (EOE) RSS...
shows the highest memory allocation observed at transaction termination time for a repeated number of simulations. For a single transaction executing within a single JVM thread, the peak RSS was observed to be 552 MB with EOE RSS at 515 MB. We can see a consistent increase in peak RSS with progressive increase in the number of transactions. For 128 concurrent transactions executing in independent JVM threads, the peak RSS reaches 3298 MB with EOE RSS at 3170 MB. However, we do not see the drastic difference in memory consumption between 8 and 128 transactional threads. In fact, peak RSS for 8 concurrent threads is 2525 MB – a mere 700 MB difference observed with peak RSS for 128 threads. Such results, once again, empirically confirm that actual TSC implementation could be rather memory efficient at a larger scale, occupying only a fraction of available system RAM at peak load times on modern server hardware.

The important fact to observe is that the peak RSS with the same number of concurrent transactions has been nearly identical for two different data object sizes depicted in Fig. 11 and Fig. 10. In fact, the RSS is slightly larger for 128 transactions associated with a 64 MB data object. That shows, that object size does not have a strong impact on the real memory usage of individual collaborative transaction conducted through our TSL implementation. As already mentioned, this is largely due to the use of Java NIO library (Java NIO Developers, 2016) in our TSL implementation. It also shows that modern Java platforms could often times provide a viable alternative to compiled languages such as C++ for complex and secure enterprise-grade middleware implementations (Welsh et al., 2001).

For the sake of completeness we provide the actual replication time before disk saturation observed for the TSC load simulation associated with a 2048 data object depicted in Fig. 11. For a single transaction executing within a single JVM thread the average replication time is 22.406 s. For 8 threads the average replication time is 199.545 s. For 16 threads the average replication time to concurrently complete the replication of 16 2048 MB data objects is reported to be 431.72 s. Note that reported replication time might be nearly irrelevant in real-world settings where dozens of concurrently running service components could add additional I/O load on the server storage hardware with service-specific filesystem activity. That could significantly increase the completion time of a single collaborative transaction for a set of large data objects.

10. Discussion on user-space LPM

Our framework is supported through the user-space LPM reference monitor which becomes the main differentiator in contrast to most existing works (covered in Section 11) that are mainly based on kernel-space solution such as SELinux (SELinux Developers, 2016) and DTE (Badger et al., 1995). In contrast to such kernel-level solutions, LPM provides access control facilities that are mainly oriented toward containerized application services that operate in user space under unprivileged UIDs and may need incremental access to OS and hardware resources mediated through kernel. The isolated components of such services that require specific elevated privileges are given them through the notion of LPM’s capabilities classes abstraction (discussed in Section 2) without a need to incorporate direct kernel-level support into the reference monitor. At the same time, secure information flow between such components does not require kernel-space support because components exchange business-logic flows through customized tuple space abstraction in user-space. In fact, mediation of intensive data flows through kernel-level IPC has two inherent problems. First, its performance is architecturally limited by the cost of invoking the kernel and mediating cross-address space interaction for two communicating components. Inherently, the cost of context switching from kernel space to user space for large data transfers can be detrimental to overall system performance (Bershad et al., 1991). Second, service components execute in user space and therefore benefit from IPC mechanism implemented through user-level libraries (Bershad et al., 1991) such as our TSL implementation that avoids complexities of message passing that requires additional synchronization primitives (Belyaev and Ray, 2016a). For instance, user-space IPC daemons such as D-Bus (Havoc Pennington, Red Hat, 2016) have been specifically designed to leverage such user-space advantages. Moreover, having the kernel copy application-level data and coordination messages between address spaces of interacting service components is neither necessary nor sufficient to guarantee safety (Bershad et al., 1991; Krohn et al., 2007).

One of the main advantages of the user-space design for the reference monitor is portability, ease of implementation and in some sense correctness. LPM does not destabilize the kernel (Belyaev and Ray, 2016a; Krohn et al., 2007). At the same time, our user-space reference monitor may benefit from tighter integration with OS kernel through user-level interface such as Linux Security Modules (Wright et al., 2002) (LSM) hooks or related mediation layers. For instance, LPM already utilizes the capabilities management through calls to user-space LibCap (Linux Programmer’s Manual, 2016b) library that has direct interface to kernel. Note that enforcement of more fine-grained, execution-specific security policies for individual isolated service components is possible through existing kernel-space access control solutions such as SELinux that may be used in combination with our user-space reference monitor (Belyaev and Ray, 2017).

11. Related work

Traditionally, Linux environments supported Discretionary Access Control (DAC) which allows read, write, and execute permissions for three categories of users, namely, owners, groups, and all others for managing access to files and directories in the user-space. Another type of supported access control is based on the Mandatory Access Control (MAC) designed to enforce system policies: system administrators specify policies which are checked via run-time hooks inserted into many places in the operating system’s kernel. For managing access to system resources, typically superuser privileges are needed. Each file in the system is annotated with a numerical ownership UID. Applications needing access to system resources temporarily acquire the privilege of the superuser. The superuser is assigned UID = 0 – a process executing with this UID can bypass all access control rules. This simple model violates the principle of least privilege.
Researchers have proposed Domain and Type Enforcement (DTE) (Badger et al., 1995, 1996; Hallyn and Kearns, 2000) for Linux and UNIX environments. Type enforcement views a system as a collection of active entities (subjects) and a collection of passive entities (objects) (Badger et al., 1995, 1996). DTE is designed to provide MAC to protect a system from subverted superuser processes as the access control is based on enforceable rule sets. The DTE model, unlike the other Linux approaches, avoids the concept of users and only concentrates on applications (Badger et al., 1995, 1996). Our work, like DTE, also concentrates on access control requirements of applications and their interaction. We also express policies in a human readable form. However, our LPM is entirely resident in user-space in contrast to DTE that offers kernel level solution. Moreover, we target the access control requirements necessary for the manageable deployment of large numbers of localized isolated application services under unprivileged UIDs in isolated environments, such as chrooted jails and application containers. Such environments were outside the scope of DTE.

Security-Enhanced Linux (SELinux) (Loscocco, 2001; Spencer et al., 1999) allows for the specification and enforcement of MAC policies at the kernel level. SELinux uses the Linux Security Modules (LSM) (Wright et al., 2002) hooks in the kernel to implement its policy. The SELinux architecture is based on the Generalized Framework for Access Control (GFAC) proposed by Abrams et al. (1990) and LaPadula (1995) and supports multiple security models. In SELinux the policy server makes access control decisions and the object managers are responsible for enforcing access control decisions. It provides a policy description language for expressing various types of policies. SELinux supports the concepts of roles and users but is not intended for enforcing policies at the level of individual applications. Policy description and configuration in SELinux is non-trivial because of the relationships between multiple models of SELinux and consequently it is a little challenging to use (Xu et al., 2014). Our work complements the efforts of SELinux in that it provides access control for isolated applications in user-space.

The Rule Set Based Access Control (RSBAC) (Ott and Fischer-Hübner, 2001) attempts to bring more advanced access control model to Linux based server systems. RSBAC is an open source security extension for current Linux kernels. The kernel based patch provides high level of security to the Linux kernel and operating environment. All RSBAC framework components are hard-linked into the custom-built Linux kernel. RSBAC supports divergent security policies implemented as modules in a single framework. However, the framework does not have a mature representation format to provide a unified way of modeling and expressing the policies for all the diverse policy modules that the framework claims to support. This limits its wide-spread adaptability. In contrast to RSBAC, our work provides domain-specific expressive policy formulation framework and is implemented in user-space that allows it to be deployed on any Linux server system.

The Grsecurity package (GrSecurity Developers, 2016) is a composition of Linux kernel patches combined with a small set of control programs. The package aims to harden known vulnerabilities in the Linux system while paying special attention to privilege escalation and root exploits. Grsecurity provides a MAC mechanism based on ACL and RBAC capabilities combined with trusted path execution that allows to limit the right of program executions to certain specified file names. The set of patches provides protection mechanisms for file systems, executables and networks. Grsecurity can harden the chroot environment against certain known attacks, prevent unprivileged users from reading kernel information and is able to limit and isolate the operating system’s process view. In short, Grsecurity hardens the Linux operating system and its proprietary mechanisms while restraining system entities like users and processes. It does this by placing additional logic on the Linux kernel and also alters the kernel’s own mechanisms to comply with the desired behavior. Grsecurity does not follow any formal model of security and access control, but emerged as a composition of countermeasures against several known weaknesses, vulnerabilities, or concrete attacks. Therefore, it lacks a general systematic approach or comprehensive formal model. Consequently, analysis of the security properties of the various mechanisms is non-trivial despite recent attempts to develop techniques for its formal analysis (Bugliesi et al., 2012).

The decomposition of complex, legacy, monolithic applications into fine-grained, least-privilege memory compartments is supported by Bittau et al. (2008) that provides programming primitives to allow the creation of compartments with default-deny semantics. Specifically, Bittau et al. offer a flexible memory tagging scheme that grants a compartment memory privileges at a memory-tag granularity for the related memory objects. Contrarily to Bittau et al., in our work we deal with information flow control at the granularity of stand-alone OS processes. Efforts at compartmentalization have also been recently observed in UNIX systems with Capsicum framework (Watson et al., 2010, 2012) that represents a lightweight operating system capability and sandbox framework planned for inclusion in FreeBSD. It supports compartmentalization of monolithic UNIX applications into logical applications. By adding capability primitives to standard UNIX API, it gives application developers a path to satisfying the requirement of least-privilege. Privilege separation also referred to as compartmentalization is addressed by introducing capabilities and capabilities mode. However, these capabilities should not be confused with Linux capabilities, which are coarse-grained privileges that are not associated with objects and cannot be transferred across processes. Capsicum capabilities are an extension of UNIX file descriptors and reflect rights on specific objects such as files and sockets. In contrast to our work that adds a purely user-space management layer on top of the existing kernel level support for capabilities, Capsicum requires application modifications to exploit these new security functionalities. Moreover, extensive kernel changes are needed in Linux and FreeBSD to provide support for Capsicum capabilities. However, our access control framework also permits the management of compartmentalization for a single monolithic application through the notion of policy classes and regulation of control and data flow interaction between such isolated compartments implemented in the form of containerized OS processes.

The application-level access control is emphasized in Decentralized Information Flow Control (DIFC) (Myers and Liskov, 2000). DIFC allows application writers to control how data flow between the pieces of an application and the outside world.
As applied to privacy, DIFC allows untrusted software to compute with private data while trusted security code controls the release of that data. As applied to integrity, DIFC allows trusted code to protect untrusted software from unexpected malicious inputs. In either case, only bugs in the trusted code, which tend to be small and isolated, can lead to security violations. Current DIFC systems that run on commodity hardware can be broadly categorized into two types: language-level and operating system-level DIFC (Krohn et al., 2007; Roy et al., 2009). Language level solutions provide no guarantees against security violations on system resources, like files and sockets. Operating system solutions can mediate accesses to system resources, but are inefficient at monitoring the flow of information through fine-grained program data structures (Roy et al., 2009). DIFC efforts like Flume (Krohn et al., 2007) and Laminar (Roy et al., 2009) generally employ an interposition layer that replaces system calls with Inter-Process Communication to the reference monitor, which enforces data flow policies and performs safe operations on the application’s behalf. Regardless of the level of implementation (Language level, OS level or both as in the case of Laminar) the security and privacy guarantees come at a price – application code has to be modified and performance overheads are incurred on the modified binaries. Moreover, the complexities of rewriting parts of the application code to use the DIFC security guarantees are not trivial and require extensive API and domain knowledge (Roy et al., 2009). These challenges, despite the provided benefits, limit the widespread applicability of this approach. Our solution allows to divide the information flow between service components into data and control planes that are regulated through the user-space reference monitor. Therefore, no modification to OS kernel is required. The rewrite of existing applications for utilization of data flow may not be necessary, since a separate flow requesting application that leverages our TSL can handle such a task and deliver the replica of a data object to unmodified application (Belyaev and Ray, 2016b).

Application-defined decentralized access control (DCAC) for Linux has been recently proposed by Xu et al. (2014) that allows ordinary users to perform administrative operations enabling isolation and privilege separation for applications. In DCAC applications control their privileges with a mechanism implemented and enforced by the operating system, but without centralized policy enforcement and administration. DCAC is configurable on a per-user basis only (Xu et al., 2014). The objective of DCAC is decentralization with facilitation of data sharing between users in a multi-user environment. Our work is designed for a different deployment domain – provision of access control framework for isolated applications where access control has to be managed and enforced by the centralized user-space reference monitor at the granularity of individual applications using expressive high-level policy language without a need to modify OS kernel.

In the realm of enterprise computing applications running on top of Microsoft Windows Server infrastructure the aim is to provide data services (DSs) to its users. Examples of such services are email, workflow management, and calendar management. NIST Policy Machine (PM) (Ferraiolo et al., 2014) was proposed so that a single access control framework can control and manage the individual capabilities of the different DSs. Each DS operates in its own environment which has its unique rules for specifying and analyzing access control. The PM tries to provide an enterprise operating environment for multi-user base in which policies can be specified and enforced in a uniform manner. The PM follows the attribute-based access control model and can express a wide range of policies that arise in enterprise applications and also provides the mechanism for enforcing such policies. Our research efforts are similar to NIST PM (Ferraiolo et al., 2014) since it offers the policy management and mediation of data services through a centralized reference monitor. However, our access control goals are different. We do not attempt to model user-level policies as done by NIST PM. Our framework, on the other hand, provides the mechanism exclusively for controlled inter-application collaboration and coordination of localized service components across Linux-based isolated runtime environments that also regulates access to system resources based on the principle of least privilege. Note that the importance of such a mechanism that is not currently present in NIST PM is acknowledged by its researchers (Ferraiolo et al., 2014).

Details on the design and implementation of our tuple space paradigm for the enforcement of regulated information flow has been recently presented by us in Belyaev and Ray (2017). In our current work we provide further details on the unified framework, that incorporates discussion on the formal aspects of policy classes and details on the implementation of the Policies Store for storage and management of access control records. Furthermore, we provide discussion on actual component-oriented policies that can be expressed using our DSPL language.

In the mobile devices environment Android Intents (Chin et al., 2011) offers message passing infrastructure for sandboxed applications; this is similar in objectives to our tuple space communication paradigm proposed for the enforcement of regulated inter-application communication for isolated service components using our model of communicative policy classes. Under the Android security model, each application runs in its own process with a low-privilege user ID (UID), and applications can only access their own files by default. That is similar to our deployment scheme. Our notion of capabilities policy classes is similar to Android permissions that are also based on the principle of least privilege. Permissions are labels, attached to application to declare which sensitive resources it wants to access. However, Android permissions are granted at the user’s discretion (Armando et al., 2015). Our server-oriented centralized framework deterministically enforces capabilities and information flow accesses between isolated service components without consent of such components based on the concept of policy classes. Despite their default isolation, Android applications can optionally communicate via message passing. However, communication can become an attack vector since the Intent messages can be vulnerable to passive eavesdropping or active denial of service attacks (Chin et al., 2011). We eliminate such a possibility in our proposed communication architecture due to the virtue of tuple space communication that offers connectionless inter-application communication as discussed in Section 3. Malicious applications cannot infer on or intercept the inter-application traffic of other services deployed on the same server instance because communication is performed via isolated tuple spaces on a filesystem. Moreover, message spoofing is also precluded by our architecture since the enforcement of message passing is conducted via the
centralized LPM reference monitor that regulates the delivery of messages according to its policies store.

Our work also bears resemblance to the Law-Governed Interactions (LGI) proposed by Minsky et al. (Minsky and Ungureanu, 1998; Minsky et al., 2000) which allows an open group of distributed active entities to interact with each other under a specified policy called the law of the group. The inter-application communication in our work is proposed in the same manner via the tuple space using the Tuple Space Controller integrated in our centralized LPM reference monitor that has complete control over inter-application interaction (Cremonini et al., 2000; Minsky and Ungureanu, 1998).

The tuple space model as a type of shared memory, originally introduced by Linda (Gelernter, 1985), has been widely adapted for parallel programming tasks (Buravlev et al., 2016; Wells et al., 2004), support for language-level coordination (Vitek et al., 2003), multi-agent systems (Cabri et al., 2000; Cremonini et al., 2000) and distributed systems (Lucchi and Zavattaro, 2004; Minsky and Ungureanu, 1998; Minsky et al., 2000) in general. Several commercial implementations of tuple space paradigm have also been developed in the past, targeting highly parallel and High-Performance Computing (HPC) applications with enhanced support for tuple persistence, distribution across network nodes and matching capabilities (Buravlev et al., 2016). We have adapted the original Linda model to serve the requirements of secure inter-component communication within a single-node OS with dedicated filesystem-level space per component. In comparison to traditional tuple spaces that allow potentially thousands of tuples per single space, our search complexity is minimal since only at most two tuples are allowed to be present in a given tuple space. That is a deliberate restriction imposed by the necessity of providing basic DoS protection and resource preservation when dealing with concurrent transfers of large data objects made possible through our LPM middleware. As covered in Section 3, the original paradigm has a number of resource-oriented limitations (Buravlev et al., 2016) and does not offer security guarantees. For that matter, many researchers (Cabri et al., 2000; Minsky et al., 2000; Vitek et al., 2003; Yu and Buyya, 2004) have conducted adaptation of the original tuple space model to fit the domain-specific requirements. The LighTS tuple space framework (Balzarotti et al., 2007) is somewhat similar to our work in a sense that it also provides localized variant of a tuple space per application with a possibility of persistence. However, it has adapted the original operations on Linda tuple space for use in context-aware applications. LighTS offers support for aggregated content matching of tuple objects and other advanced functionality such as matches on value ranges and support for uncertain matches. Our adaptation allows coordination and collaboration between isolated service components based on precise content matching on a set of tuple fields. Our model allows a mixed mode of information transfer between service components – tuples can contain actual language-level objects or could be used to replicate larger data objects such as large ASCII file objects. Note, that no restriction on types of replicated objects exists in our TSL implementation – aside from ASCII objects, a complete byte-level replication is entirely possible. Therefore, data objects, such as images, could be potentially replicated between service components. We also enable dual planes of inter-component communication – components can communicate using a control plane, data plane or both. To the best of our knowledge, we offer the first persistent tuple space implementation that facilitates the regulated inter-component communication without a need for components to share a common memory address space or requirements for address space mapping mechanisms (Belyaev and Ray, 2016b; Bershad et al., 1991).

12. Conclusion and future work

We have demonstrated how LPM can be developed for the Linux environment that provides access control specification and enforcement for various service components running in isolated environments. We proposed the notion of policy classes to manage policies pertaining to accessing system and application level resources and demonstrated how regulated inter-component communication can take place through tuple spaces. We formalized the access control policies of the LPM. The initial prototype demonstrates the feasibility of our approach. We plan to extend this work for distributed settings (Belyaev and Ray, 2015, 2016c; Singh et al., 2014) where service policies are managed, formulated and updated in a centralized location, and then distributed and enforced at LPM nodes within datacenter settings. We also plan to adapt the developed framework to use with resource-constrained devices such as IoT nodes where a single node may serve multiple applications and it is important to provide controlled interactions across such applications.

Acknowledgments

This work was supported in part by a grant from NIST under Award Nos. 60NANB16D249 and 60NANB16D250, by grants from NSF under award nos. CNS 1619641 and NSF I/UCRC 1650573, and by funding from CableLabs.

REFERENCES


