Securing a Space-Based Service Architecture with Coordination-Driven Access Control*

Stefan Craß[†], Tobias Dönz, Gerson Joskowicz, eva Kühn, and Alexander Marek
Institute of Computer Languages
Vienna University of Technology
Vienna, Austria
{sc, td, gj, ek, amarek}@complang.tuwien.ac.at

Abstract

In distributed applications, multiple autonomous processes need to collaborate in an efficient way. Space-based middleware enables data-driven coordination for these processes via shared tuple spaces that allow a decoupled form of communication. Complex coordination logic may be provided to clients via reusable service components that access such tuple spaces to fulfill their task. To enable the secure collaboration of different participants, a suitable security concept for space-based services is required. In this paper, we present a fine-grained access control model that targets permissions both for invoking specific coordination services and for the data that is accessed by them. Our space-based policy language adopts the middleware's own coordination mechanisms for the specification of simple yet expressive access control policies, thus combining coordination logic and security mechanisms into a single, unified concept. We show how a lightweight service execution framework that enforces these policies can be bootstrapped with the middleware itself, which enables using the same mechanisms for the invocation of services, the access to data and the management of policies. The feasibility of the approach is demonstrated by a use case based on a management system for distributed firewalls.

Keywords: tuple spaces, coordination middleware, access control, service-oriented architectures

1 Introduction

Autonomous processes that communicate and collaborate with each other form a distributed system that requires coordination mechanisms like data sharing, synchronization, and the remote invocation of APIs. In such distributed scenarios, it is crucial to guarantee the confidentiality and integrity of data. Thus, each involved process should act on behalf of a principal with specific privileges that define the authorized interaction capabilities for the process. The management of access control policies that specify these privileges is a complex task, especially when multiple different organizations and individuals are involved that want to protect their own data and their offered services from malicious access. Secure collaboration among multiple organizations can be accomplished with service-oriented architectures (as used for Web services) combined with standardized security mechanisms. However, such structures are complex to establish and collaboration cannot be done in an ad-hoc manner. Therefore, a flexible mechanism is required for secure inter-enterprise collaboration that allows participants with different roles to interact using an ad-hoc, reactive, and multi-party coordination model.

*This paper is an extended version of the work originally presented at the 7th International Conference on Availability, Reliability and Security (ARES'12), Prague, Czech Republic, August 2012 [1].

[†]Corresponding author: Vienna University of Technology, Institute of Computer Languages, Argentinierstr. 8, 1040 Wien, Austria, Tel: +43-1-58801-18517, Email: sc@complang.tuwien.ac.at

Middleware encapsulates coordination and security mechanisms in distributed systems via proper abstractions. Thus, developers can invoke reliable middleware methods for their coordination logic and security policies instead of having to implement these functionalities themselves, which would lead to more error-prone and less maintainable applications. In this paper, we present a middleware architecture that uses space-based services as a suitable abstraction for secure collaboration in distributed systems.

Space-based services adhere to the space-based computing paradigm, which supports indirect process communication via queryable shared data spaces. This architectural style is based on the Linda tuple space model [2], which specifies how data tuples can be written to and retrieved from a shared space. For the consuming and non-consuming retrieval operations, a query mechanism called template matching is used, which checks if any tuple in the space matches a given template that specifies the expected content for specific fields of the searched tuple. If no matching tuple is available, the operation may optionally block until a proper tuple is written to the space, which allows for an easy way to synchronize processes. Using these simple concepts, tuple spaces enable the coordination of processes that are decoupled in space and time [3]. They do not need to know each other because they only communicate via shared tuples, and they do not need to be available at the same time as the space stores the tuples until they are consumed. Due to the reduced dependencies between collaborating processes, an agile software development style can be applied in a natural way [4]. Processes may join and leave a space dynamically without missing relevant data because the state of a distributed application can be completely modeled via tuples in the space. Therefore, space-based middleware is well suited for the collaboration of autonomous agents [5]. One problem of Linda, however, is that template matching offers only limited query capabilities as only exactly matching fields are regarded. A simple sorting or checks if some value is greater or less than a given threshold are therefore not possible.

The original Linda model also does not handle security, as every process that is able to connect to the space may write or remove arbitrary tuples. For practical usage in open networks like the Internet, secure tuple space variants need to be applied that allow the definition of appropriate access rights. A suitable access control model must enable the flexible management of permissions for highly transient coordination data. To address these problems, several extended tuple spaces have been designed that add new coordination and security features [6], but the natural integration of flexible coordination mechanisms with fine-grained access control policies into a single middleware model is still an open research topic. To achieve this goal, a tight connection of coordination and security features must be established [7]. This means that similar mechanisms should be used to select tuples from a space and to define permissions on them, providing a single paradigm for coordination and security.

Tuple spaces already provide a good mechanism for flexible collaboration of distributed processes, but they require a rather low-level data-driven programming style for application developers, where every interaction step has to be explicitly modeled via space operations. Reusable coordination patterns can help to simplify the software development process and reduce code complexity. These patterns can be expressed via coordination services that fulfill specific tasks by invoking one or more space operations. Developers may either use the high-level API of these services or directly interact with the space.

In traditional service-oriented architectures, the invocation of services via SOAP or REST-based protocols provides the basic coordination between participants. Since the outcome of individual service requests depends on resources like databases, security in these architectures differs between authorization for functions and for data access. Thus, effective access control policies must be established both for the server hosting the service and for the associated resources, which usually use different mechanisms for managing permissions. To provide a proper decoupling between services and resources, the service should not be responsible for managing the user's permissions on the accessed resources. A combined access control approach for services and data would make the semantics of access control policies clearer and easier to manage. Space-based middleware offers more advanced coordination features than common mechanisms for enterprise application integration like message-oriented middleware and publish/subscribe systems. Especially in scenarios that require many-way collaboration among multiple participants, this architectural style has been shown to be more effective [8]. Therefore, it is possible to use tuple spaces both as resources for coordination services and for a lightweight service framework alternative to common service-oriented architectures. When combined with suitable security mechanisms for space-based middleware, a holistic approach for secure space-based services can be realized that offers a joint authorization approach for coordination services and their resources.

This paper presents a flexible coordination-based access control model for the space-based middleware XVSM [9, 10] and a secure service architecture that is realized on top of it. In XVSM, processes collaborate via space containers that support multiple coordination mechanisms besides template matching, like FIFO queues or SQL-like queries. We show how the advantages of space-based and service-oriented architectures can be combined using a space-based service model, where services do not only operate on the space but are also invoked by writing requests into containers. The main challenge was to specify suitable security mechanisms for this model that combine privileges for service and data access in a natural way using comprehensible yet expressive access control policies that can be easily managed. We use an access control model that allows fine-grained rules based on XVSM's extensible query mechanism and the well-known XACML standard [11]. Rules can be specified both to grant permissions on invoking certain services and on accessing data containers, either directly via space operations or indirectly via the service. Finally, we show how a Secure Service Space architecture can be built that enforces these policies based on authenticated security attributes and a container-based policy management.

The XVSM access control model was originally introduced in [12] and extended in [1], which also describes a bootstrapped authorization architecture for enforcing this model. In this paper, we introduce a novel service framework on top of XVSM that provides a more flexible programming model than using the space on its own. To enable a secure service environment, we generalize the Secure Space architecture from [1], which allows to constrain permitted requests to the XVSM runtime, into a Secure Service Space model that supports access control rules on requests for arbitrary coordination services that operate on an XVSM space. The main contribution of this paper is therefore the introduction of a secure high-level service framework for space-based middleware. We provide an analysis of a practically relevant use case from the firewall management domain to show the feasibility of this extended model.

The paper is organized as follows: Section 2 describes the functionality of the XVSM middleware and introduces a simple model for space-based services that are built on top of it. Section 3 analyzes the requirements for space-based authorization and outlines the basic idea of the used access control model. In Section 4, the syntax and semantics of the access control policy language for this model are described, while Section 5 presents the Secure Service Space, which is based on the space-based service architecture and the XVSM access control model. Section 6 shows how the Secure Service Space can be applied for the management of distributed firewalls. Section 7 evaluates the feasibility of the described approach and compares it to related work. Finally, we conclude and present future work in Section 8.

2 Service-Based Space Model

For designing a suitable service-based space model for the flexible collaboration of distributed components, we follow the basic tuple space paradigm by allowing blocking and non-blocking queries on a shared space that provides a simple API for writing and retrieving data. However, we want to support additional coordination laws beside Linda template matching, which has limited expressiveness when checks on equality of fields are not sufficient. Thus, we have chosen an extensible coordination approach that includes query features from relational databases, message queues, and key-value stores. Section 2.1 explains this approach by means of our space-based middleware architecture XVSM (eXtensible Virtual Shared Memory), which has been previously described in [9] and [10]. In Section 2.2, we introduce a

Table 1: Some predefined coordinators with required coordination data and selection parameters

Coordinator	Coordination data	Selection parameters	Coordination law
any	-	count	arbitrary selection
key	key	key	selection via unique key
label	label(s)	label, count	selection via non-unique label
fifo	-	count	selection in FIFO order
type	-	type, count	selection via data type of entry
linda	-	template, count	Linda-like template matching
query	-	query	SQL-like query on entry fields with comparisons, sortings, etc. [10]

space service model on top of XVSM, which defines an architectural style for implementing XVSM services and their clients. Clients can directly manipulate data using XVSM operations but also indirectly via reusable coordination services that access the space. The Secure Service Space, which is described later in this paper, provides an architecture that adds security mechanisms to this service model.

2.1 XVSM Middleware

XVSM is a space-based middleware that provides methods for writing and retrieving *entries* to and from structured sub-spaces called *containers*. The main operations on containers are `write`, `read`, and `take`, the latter of which reads and deletes the selected entries in a single step. Each container is managed by one or more *coordinators* that determine the coordination law of the container and define how entries can be written and queried. A coordinator has an internal view of the container, which may correspond to a certain ordering or a map. This view is updated whenever an entry is added to or removed from the container, and it is queried when entries are selected using the coordinator. Therefore, coordinators have to provide `register` and `unregister` methods as well as a `select` function.

Whenever an entry is written to a container, the `register` method is invoked for all associated coordinators, which add the entry to their views. Beside the actual entry, some coordinators require additional *coordination data* as extra parameters, like a key that can be used to select the entry later on. The registration may fail if some internal coordinator constraint would be violated (e.g. duplicate keys) by writing the entry. After an entry is removed using the `take` operation, the `unregister` method is called for all of the container's coordinators to remove the entry from their views.

For `read` and `take` operations on a container, an *XVSM query* is given, which is represented as a chain of one or more *selectors*. Each selector is associated with a specific coordinator and requires zero or more selection parameters. The query is evaluated by calling the `select` functions of each addressed coordinator, which take a list of input entries and the given selection parameters to return a list of output entries. For the first selector in the XVSM query the input corresponds to the whole container, whereas subsequent selectors use the output of the previous stage as input. The result of the query are the output entries of the last selector. The invoked `select` functions may filter or reorder entries, but it is not possible to add or modify them. An XVSM query may, however, fail if not enough matching entries are currently available for one of the involved `select` functions.

Table 1 shows some predefined coordinators with their basic coordination laws, their coordination data parameters for writing entries, and their selector arguments for querying entries. This coordination mechanism is extensible because custom coordinators can be added easily as long as they implement the basic coordinator methods. Some coordinators support an optional `count` parameter for selecting entries, which can either be a concrete number of entries that must be available, or the default value `ALL`, which indicates that all matching entries should be returned, even if the result is empty. As an example, the selector chain "`type(`*`Event`*`) | query(priority > 5) | fifo(3)`" consists of three selectors that are separated by the "`|`" symbol. This query chooses all entries of type `Event`, then selects those events with a priority higher than five, and finally returns the first three of these entries according to FIFO order. The `fifo` selector causes the query to fail if less than three entries are available at this point. The container is accessed via an API that can be invoked synchronously or asynchronously via callbacks:

```
write(container, entriesWithCoData, timeout, transaction, context)
read(container, query, timeout, transaction, context)
take(container, query, timeout, transaction, context)
```

Beside the target container, which is identified via a unique URI, and entries with coordination data or a selector chain, respectively, three additional parameters are supported. The *timeout* specifies how long an operation should be retried if it does not succeed immediately, thus supporting blocking and non-blocking operations. With an optional *transaction*, several operations can be combined into a single atomic step. Transactional behavior is guaranteed via pessimistic locking on entries. If an entry is locked by a concurrent transaction, coordinators may either decide to return a different entry or indicate that no matching entry is currently accessible, which may cause the operation to block. The *context* is a set of key-value pairs that includes further information about the operation, like the identity of the issuer.

Additional methods provided by XVSM include the creation, deletion, and lookup of containers as well as the management of transactions. The semantics of XVSM can be extended via *aspects*, which are dynamically registered to run before or after specific operations. Aspects may adapt operation parameters or return values, and invoke other XVSM operations or external methods. As an aspect may also cause the associated operation to fail, a basic security mechanism could be implemented that explicitly checks credentials and grants access only for selected subjects. However, a declarative approach as presented in this paper provides a more flexible authorization mechanism that is easier to maintain than explicitly modifying program code for every changed security requirement.

2.2 Space Services

As shown by several space-based service models [13, 14, 15], tuple spaces can be used for connecting clients and services in an ad-hoc way, which enables complex message exchange patterns. As with message-oriented middleware, reliable message delivery that is decoupled in time and space can be achieved. If the services themselves are coordination services that realize their functionality via space operations, a space-based service framework appears as a logical choice to minimize system complexity.

Basically, client applications could coordinate themselves by accessing tuple spaces directly. However, by interposing space-based coordination services, we gain several advantages. Firstly, security is improved, as clients are not directly exposed to the internal data structures of a given system. Secondly, writing own coordination code is more error-prone than invoking reusable code provided by tested coordination services. Thirdly, the maintainability of the system is improved, because having the coordination logic for related containers on a single space site prevents inconsistencies. This logic can be encapsulated within several services that provide well-defined functionality for the clients.

To realize space-based services using XVSM, only two additional containers are required. The *request container* stores service requests from clients, whereas the *response container* stores the corresponding service results. Figure 1 shows the service model for an example with two services `X` and `Y`. A service is invoked by writing a suitable request entry for a specific service (like `XReq` for service `X`) into the request container. To fetch requests, services issue a blocking `take` operation for matching request entries using a `type` query on their corresponding request type. After the service has fulfilled its task, it

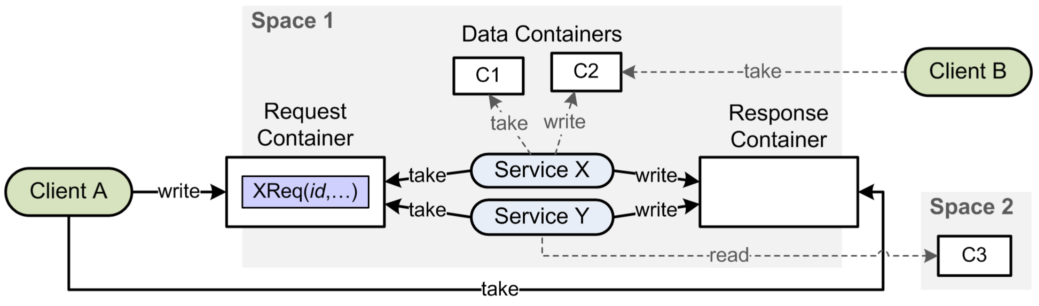


Figure 1: Space services with XVSM.

writes the result into the response container with `key` coordination using a unique request ID that has to be included with each request. Afterwards, the service continues with fetching another request if available, while the client can take the result using the request ID. This `take` operation can either be invoked directly after writing the request and return when the result is ready, or the client may choose to query the response container later on. Depending on the active security policies, clients may be forced to use the space services or they could be able to bypass this mechanism and directly access the containers if this fits the application requirements better.

According to the request's parameters and the service logic, the service may access additional containers during its execution. In the example, service `X` takes entries from container `C1` and then writes these entries (or some form of modified or aggregated data) into `C2`. The entries are used to synchronize the requestor with client `B`, which waits for input into the container using a blocking `take` operation. Alternatively, client `B` could invoke a service that executes this `take` operation and returns its result in the response container, but for a single space operation a service invocation does usually not pay off due to the low complexity of the operation and the extra overhead of the service architecture. For client `A`, however, the service actually reduces complexity because the client does not need to understand the low-level synchronization protocol with the other client. Service `Y` demonstrates that a service may also access containers on remote spaces. As the involved processes are fully decoupled via XVSM containers, the location of the involved containers can be changed in a flexible way. If container `C1` is moved to a remote site, only the container URI used in the implementation of service `X` has to be adapted, while the clients are basically unaffected by the change. In fact, also the request and response containers could be located on different machines than the services, which increases the possible concurrency. As parallel container access is possible, multiple service instances of the same type are able to concurrently fetch client requests from a common request container. While one instance is occupied with executing its service logic, the others can process subsequent requests. Thus, load balancing is implicitly supported by this architecture as idle service instances actively take free requests.

3 Space-Based Access Control

Space-based middleware like XVSM can be applied to connect loosely coupled business processes. Different organizations and individuals may be involved in the distributed application that is composed of these collaborating processes. It is self-evident that each involved party needs full control over access rights on its own data and services. The expressiveness of the access control model determines the possible flexibility of the solution regarding changing security constraints. As space-based computing aims at supporting a flexible software architecture, the middleware's security mechanisms should support dynamic permission changes without modifications to the business logic.

Simple approaches for secure spaces only support setting permissions on the whole space or on fixed partitions. Access control lists (ACLs) can be set to define which users should be able to access which entity using which space operation. In some cases this might be sufficient, but it would necessitate that all entries within a container share the same permissions, which imposes a rather strict dependency between application developers, who decide which data fits best into which container, and security experts, who define which privileges should be granted to the participants. If some new user type should be able to access only a restricted set of entries within a container, these entries must be relocated to a different container with separate access rights, which requires modifications in the coordination logic of all processes that use these data. To allow the definition of more fine-grained permissions, mechanisms have been invented to define permissions for individual tuples [16, 17]. However, the problem with this approach is that spaces are mostly used for transient coordination data, which are produced by one process and consumed shortly after by others. Therefore, it is not feasible for administrators to update ACLs for every new entry before it can be accessed. Instead, the process that writes an entry has to specify the corresponding permissions, which forces the application developers to cope with authorization issues. A better and more natural way to handle space-based authorization is to define permissions based on properties of the accessed entries. Linda templates may be used within access control policies to permit or deny access to all matching tuples [18], which provides a way to dynamically change permissions without having to rewrite any coordination logic. As the expressiveness of template matching is limited, however, expressive access control models as defined for LGL [19] and TuCSoN [20] combine templates with additional mechanisms like Prolog rules. These models enable the specification of fine-grained access control rules based on the authenticated user, the content of the accessed tuples, the used space operation and its parameters, as well as additional context information.

For the XVSM access control model, we extend this approach and use the middleware's extensible coordination mechanism for specifying the set of entries that can be accessed by a certain subject. Due to the high expressiveness of XVSM queries, additional mechanisms like Prolog rules are not required to define fine-grained permissions on entries with specific content or coordination properties. As the same concepts are used for selecting and protecting entries, the policies are easy to comprehend and complement the coordination logic of the distributed application in a natural way. Our access control model is applicable both for regulating access to regular data containers as well as to the meta containers of the space service model. Thus, service requests can be secured with the same mechanism as regular entries. This approach enables multiple variants for defining permissions for a distributed application:

- **Only data access authorization:** Rules define access rights on data containers but not for service containers. Clients can invoke any service but it will fail if it needs to access entries in a way that is not permitted for the corresponding subject. This approach offers fine-grained access control policies based on the types of information that subjects are allowed to read, take, and write.

- **Only service access authorization:** Access control policies only affect the request container of the service model. Rules define whether a subject is allowed to write a particular request to the request container. To prevent users from bypassing the service model, all data containers have to be inaccessible from external sources. Only indirect access via services is allowed.

- **Combined authorization:** Both data access and service access authorization apply. If the services as well as the data containers that are accessed by them are protected individually, a second layer of protection is created. Simple coarse-grained permissions can be achieved via rules on the request container while generally allowing internal container access by services. If more fine-grained rules are required (e.g. because only specific data may be changed by a service for a particular subject), additional rules that restrict access to data containers can be added.

For large enterprise applications, it is usually not feasible to define special permissions for individuals. Instead, a role-based access control model (RBAC) [21] can be applied that assigns access rights according to organizational abstractions. Each user has one or more roles that depict his or her responsibilities within the organization. Access control rules apply for specific roles instead of single users, thus simplifying the management of permissions when positions change within an organization. Attribute-based access control (ABAC) [22] is a more general version of RBAC, where users do not only possess role attributes but arbitrary security attributes that can be targeted within an access control rule to define permissions for a set of users with common properties. For maximum flexibility in scenarios with users from multiple domains, XVSM supports both RBAC and ABAC policies.

4 Policy Language

The policy language for the XVSM access control model combines features from the well-established XACML (eXtensible Access Control Markup Language) standard [11] with the query mechanisms of XVSM to provide a simple yet expressive way for specifying permissions on spaces. XACML defines a declarative, XML-based policy language for expressing fine-grained access control policies in a uniform way. XACML *policies* consist of *rules* that contain a *target*, an *effect*, and an optional *condition*. The target defines for which requests a rule must be evaluated, while arbitrary Boolean functions on request and context attributes can be used within a condition to refine the rule. The effect of a rule is either "Permit" or "Deny". For computing the final authorization decision for a request, the effect of all applying rules must be combined using a specific *combination algorithm*. Rule targets include values for specific security attributes of subjects for which the rule should apply. Thus, XACML supports ABAC and as specialization also RBAC by using roles as security attributes [23]. For the XVSM access control policy language, we adapt XACML to the requirements of space-based authorization. We do not provide a compatible extension to XACML as the XML-based syntax and the hardly comprehensible set of features would not allow simple, human-readable policies. Instead, we define our own policy language that enriches basic XACML concepts by integrating XVSM queries, which enable the refinement of rules in a way that is inherent to space-based computing.

Similar to XACML, an XVSM access control policy is composed of uniquely named rules whose results are combined using a specific combination algorithm. The target consists of three fields: *subjects*, *resources*, and *actions*. The subjects field contains a set of subjects for which the rule applies. Each subject is identified via a set of one or more security attributes, like a role or an affiliation. The issuer of a space operation only matches the subject if it is able to provide matching values for each specified security attribute. Thus, a principal with the authenticated attributes "`[role: admin, userId: eva, affiliation: ViennaUT]`" matches the subject "`[role: admin, affiliation: ViennaUT]`". The resources field corresponds to a set of containers that are targeted by the rule using the operations (`write`, `read`, and/or `take`) defined in the actions field.

Using only rule targets and effects of either `PERMIT` or `DENY`, it is already possible to define simple ACLs on container level. To provide more fine-grained mechanisms for container access, we include a condition that adds constraints on the environmental context, which can be depicted by the content of specific containers, and a *scope* field that restricts a rule to a subset of entries within a container. A rule only applies if the target matches, the condition holds, and the entries within the scope overlap with the set of entries accessed by the operation. All rule fields except for the effect are optional, which enables the definition of general rules that may, for instance, apply to all containers for any subject.

A condition consists of one or more *condition predicates* that are combined using disjunction, conjunction, and negation. A condition predicate specifies a container and an XVSM query that has to be evaluated on it. The predicate evaluates to `true` if the execution of the query on the specified container yields at least one entry as result. If the query result is empty, the predicate evaluates to `false`. The combined Boolean values of the predicates determine the result of the condition. This mechanism can be used to define rules based on the environmental context, which is represented by entries within arbitrary containers. A conditional rule may be convenient to restrict access based on the current state of a complex interaction, which could be stored in a container that cannot be accessed by ordinary users due to different rules. A condition predicate may check if an entry for a specific state is present to activate the rule. Thus, by writing and taking entries that are accessed by conditional rules, administrators have the possibility to affect authorization decisions without modifying the actual policy. Multi-step conditions, where query parameters are dynamically set based on results of another query, are not supported to keep the complexity of the model low. If necessary, such complex conditions can be modeled via aspects that perform multiple `read` operations with dynamic parameters and write their result into a container that is targeted by a simple condition predicate.

The scope of a rule is given via one or more XVSM queries that are combinable via union, intersection, and complement operators. These queries are evaluated on the target container of the operation and their results are combined using the given set operators. Thus, a dynamically computed subset of entries is specified for which the rule applies. Using this mechanism, data-driven rules can be defined that permit or deny operations based on the content and the coordination properties of the accessed data. For instance, a rule could be specified that allows access to entries of a specific type. For `read` and `take` operations, the rule and its scope is evaluated in advance to allow coordinators to choose alternative entries if access to some matching entries is denied. For `write` operations, however, a tentative container state that already includes the written entries is examined. Therefore, the rule applies if the entries would be included in the scope after they had been written. This approach allows that fine-grained content-based rules may not only be specified for selecting but also for writing entries.

Thus, the abstract syntax of a rule can be defined using EBNF-like notation:

```
'RULE' ruleId
'SUBJECTS:' ('*' | (subject {',' subject}))
'RESOURCES:' ('*' | (containerId {',' containerId}))
'ACTIONS:' ('*' | (('write'|'read'|'take') {',' ('write'|'read'|'take')}))
'CONDITION:' ('-' | condExpr)
'SCOPE:' ('*' | scopeExpr)
'EFFECT:' ('PERMIT' | 'DENY')
```

```
with subject = '[' attrName ':' attrValue {',' attrName ':' attrValue} ']'
   condExpr = (containerId '|' xvsmQuery) | (condExpr '∧' condExpr)
            | (condExpr '∨' condExpr) | ('¬' condExpr) | ('(' condExpr ')')
  scopeExpr = xvsmQuery | scopeExpr'ᶜ' | (scopeExpr '∪' scopeExpr)
            | (scopeExpr '∩' scopeExpr) | ('(' scopeExpr ')')
  xvsmQuery = selector {'|' selector}
```

An example for an XVSM access control rule and its evaluation is shown in Figure 2. The container `eventC` holds multiple event types that may be written by some kind of monitoring system. A specific user is responsible for removing unnecessary warnings from this event container, but the system does not trust this subject enough to grant full container access. Therefore, the presented rule permits the removal of warnings with a priority of less than three, but only if a special token is set in a status container `statusC`, which allows privileged users to temporarily deactivate the rule by removing the token. To evaluate the rule, the system compares the examined operation with the target of the rule. As the provided user role, the container, and the operation type match the corresponding target fields, the rule evaluation continues with checking the condition. The condition contains a single predicate that looks for an entry in the status container with a specific key. If this entry was not present, rule evaluation would stop at this point. In the next step, the scope query is computed, which selects all events of type `Warning` that have

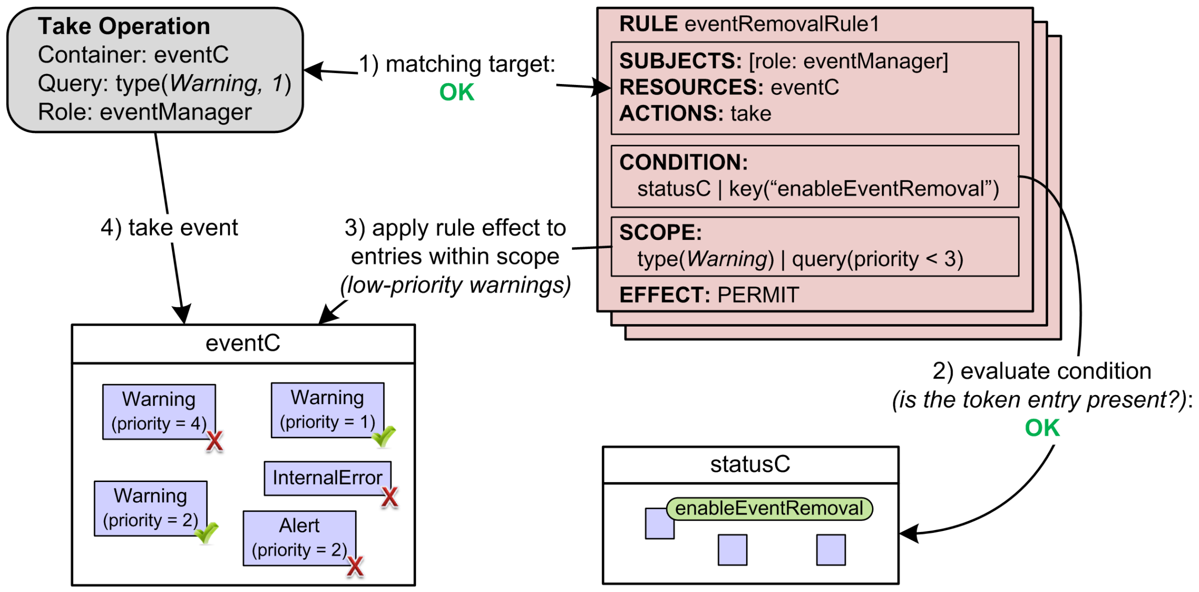


Figure 2: Rule evaluation semantics for XVSM access control model.

a `priority` property with a value of less than three. Assuming that no other rule applies, the subject is allowed to take those two entries that match the scope, while other entries are not accessible. When the examined operation is finally executed, a single `Warning` entry is taken from the container according to the given `type` selector. As the associated coordinator supports non-deterministic coordination, any of the two permitted entries is chosen, while the third entry of the same type is ignored. If the client tried to access one of the denied entries, the query would simply fail and possibly block until a given timeout is reached. For the client, there is no difference if the entry is not present, if access is denied, or if it is locked by a concurrent transaction. Thus, a transparent access control mechanism is achieved where a policy can be easily changed without affecting the business logic of involved processes.

Using the presented mechanisms, it is possible to define fine-grained rules based on the dynamic evaluation of the content and coordination properties of entries as well as the environmental context. What is still missing is a way to specify rules that depend on the context of the user. The context that is included with each space invocation contains authenticated security attributes like roles and a user id as well as additional properties set by the client or the space runtime. Context-aware queries can be supported by allowing a dynamic parameter type for XVSM queries within scope and condition fields. Such parameters start with a "$" sign followed by a distinct name to indicate that they are dynamically replaced by the corresponding value from the operation's context parameter with the same name. For instance, a rule with the scope "`label($userId)`" permits users to access only their own data instead of all entries from users with a specific role, as the dynamic label parameter is replaced with the actual user identifier at run-time. Thus, written entries can be tagged for private use with the `label` coordinator.

If several rules apply for an examined space invocation, a predefined or custom combination algorithm resolves possible conflicts and decides the final authorization result. Similar to XACML, this specifiable algorithm determines the order in which rules are evaluated and how their results are combined. These algorithms may, for instance, deny access if at least one rule denies it (`DENY-OVERRIDES`), permit access if at least one rule permits it (`PERMIT-OVERRIDES`), or chose the effect of the first applicable rule (`FIRST-APPLICABLE`). As XVSM operations may access multiple entries in a single operation and some rules may apply for only a subset of these entries, the authorization decision must be computed for each entry individually, rather than for the whole operation as in XACML. Otherwise, one rule

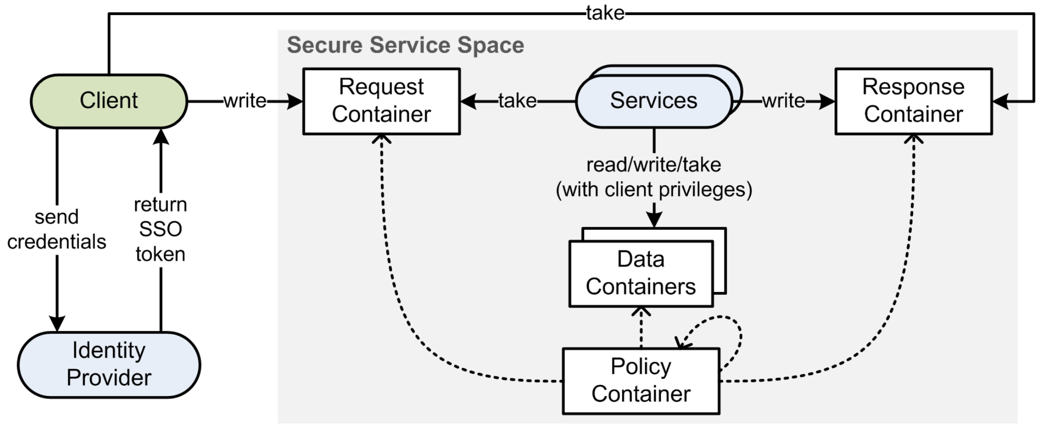


Figure 3: Secure Service Space architecture.

may permit access to one subset of entries accessed by a certain selection operation, while another rule allows queries on the remaining subset. Neither of these rules would be able to permit the operation as a whole, but if their individual entry results are aggregated, the access can be permitted. Thus, an operation is allowed if for each of its written or queried entries the defined combination algorithm yields an authorization result of `PERMIT`.

Due to the scope and condition fields, which enable content-based and context-aware access control rules via XVSM's own coordination mechanism, XVSM policies offer a higher expressiveness than simple ACLs. Compared to XACML, a simplified access control model is applied that still enables the specification of fine-grained policies on space containers based on data-driven and context-aware rules.

5 Secure Service Space Architecture

Based on the XVSM access control model and the space service model, a secured architecture for a simple space-based service execution framework can be built. This is done by adding mechanisms for authenticating subjects and for enforcing access control rules on data and service containers via an authorization architecture that is integrated into the XVSM runtime. Figure 3 shows the basic structure of this Secure Service Space. Before accessing the space, clients must authenticate their principals by sending their credentials to a trusted identity provider. We follow a brokered authentication approach where an external identity provider authenticates principals and supplies the corresponding clients with a cryptographically signed single sign-on (SSO) token that proves their identity. As a minimum requirement, authentication must at least provide a unique user identifier for each subject. Other security attributes are optional, but it is highly recommended to at least include a role attribute to enable RBAC. The SSO token is valid for a limited amount of time, thus enabling the client to reuse the token for subsequent operations on the same or on other spaces. The Secure Service Space only has to verify the SSO token and extract the authenticated security attributes like user identifier, role, and affiliation, which are necessary to enforce authorization on the space according to the currently active access control policy. Using this approach, the space itself does not need to cope with user and role management.

The authenticated client writes requests to the request container, which are then executed by the corresponding services as described in the basic space service model from Section 2.2. In order to be able to fulfill a request, the system must ensure that the client is allowed to write the request to the request container, to invoke all space operations that the service executes on behalf of it, and to take the corresponding response from the response container. XVSM access control rules that affect permissions to

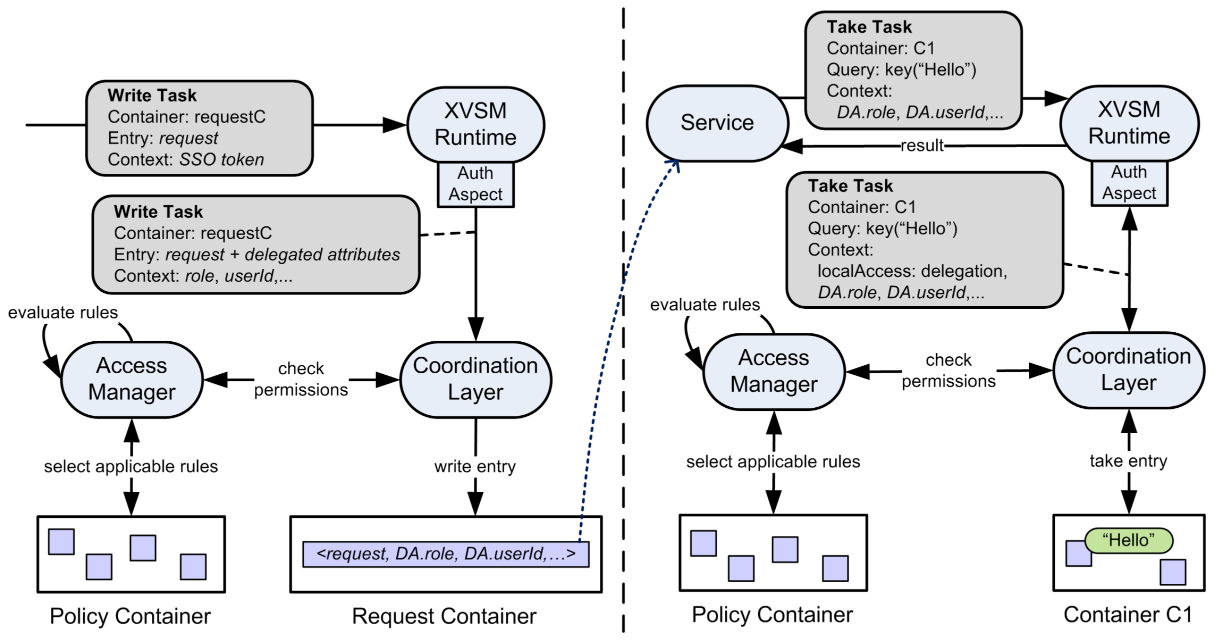


Figure 4: Authorization process for issuing a service request (left) and for indirect (local) container access by the invoked service (right).

all of these containers are stored as entries in a dedicated policy container that can be accessed using regular space operations. Service access rules affect the request container, while data access rules regulate direct and indirect access on data containers. Rules that target the response container can be used to prevent clients from accessing responses that are not directed at them. Access to the active policy can be restricted by including rules that target the policy container itself. This bootstrapped approach enables the management of administrator privileges with the same mechanism as the modification of regular permissions on services and data. Default rules ensure that a local administrator may access the policy container. To allow others to remotely manage a space, an administrator can simply insert a new rule that permits full or restricted access to the policy container for the chosen subjects.

For indirect authorization, services include the user's security attributes when invoking operations on data containers, which requires a mechanism for secure delegation. Therefore we distinguish between *authenticated* and *delegated* attributes. While authenticated attributes are verified by an identity provider and protected through an SSO token, delegated attributes are created by trusted spaces in order to access downstream services or spaces on behalf of the initiating user. We assume that access within a local space is automatically trusted, i.e. the defense against rogue services placed into a space is not covered in this paper. As administrators of the Secure Service Space have full control over the locally installed services, we argue that it is sufficient to establish trust between space instances, which can be expressed via rules that permit access for trusted spaces. Spaces may authenticate each other via dynamically obtained SSO tokens (like regular clients), although alternatively certificates issued by a trusted authority could be used. To ensure confidentiality and integrity of data, we assume that all network communication is encrypted using state-of-the-art mechanisms that protect against man-in-the-middle and replay attacks. Implementation details on authentication and encryption issues are, however, out of scope of this paper.

Figure 4 shows how the Secure Service Space model from Figure 3 and the underlying access control model is realized with XVSM. On the left-hand side, the client's service request is written to the request container, while on the right-hand side, the invoked service takes an entry from a local data container as part of its coordination logic that is executed on behalf of the client. The client includes the SSO token from the identity provider as context parameter for the write operation. Whenever an XVSM operation is invoked, internally a corresponding task object is sent to the runtime of the target space. Thus, the task from the client arrives at the runtime of the XVSM space that hosts the Secure Service Space. A special authentication aspect, which is invoked by the runtime for each request, verifies the SSO token by checking the signature of the identity provider and extracts the authenticated security attributes from the token. As all authentication logic on the space is encapsulated in this aspect, the external identity providers are easily exchangeable. The aspect modifies the write task by replacing the SSO token in the context with the security attributes and also adding them as delegated attributes to the request entry that should be written to the request container. Delegated attributes are created by adding the prefix "`DA.`" to the authenticated user attributes. The attributes in the context are used to authorize the active `write` operation, while the delegated attributes are required in the request entry for later usage by services.

The coordination layer of XVSM executes the modified task and writes the entry into the request container if the operation is permitted by the access manager. This component selects the applicable rules from the policy container using a selector function that returns all rules with a target that is compatible with the task, whereas the task's subject corresponds to the set of security attributes from its context. Then, the access manager evaluates the scope and condition fields of these matching rules and combines their results via the active combination algorithm to create an authorization result for the `write` operation. In the terminology of the XACML authorization model, the access manager acts as Policy Decision Point (PDP), which decides whether access should be permitted or denied. To retrieve the active policy rules, the PDP accesses the Policy Administration Point (PAP), which is represented by the policy container. The coordination layer corresponds to the Policy Enforcement Point (PEP), which invokes the PDP for every operation and enforces its decision. Finally, the Policy Information Point (PIP), which provides additional information for the PDP to form its decision, is composed of all containers that are included in scope and condition queries of XVSM access control rules. The PAP and PIP containers are read by the access manager using the same query mechanism as regular `read` operations by clients. As an internal component of the space runtime, the access manager is implicitly authorized to access all local containers. Additional details on how the implementation of the coordination layer and the access manager module enforce the XVSM access control model can be found in [1].

If a service accesses the local space, it is implicitly authenticated via the `localAccess` attribute, which is set by the authentication aspect of the runtime. This special authenticated attribute either has the value "`delegation`" if the context contains delegated attributes that depict the user identity on whose behalf the task should be executed, or "`system`" if the service itself needs to access a container. Depending on the used attributes in the subject, several types of rules can be distinguished that are all enforced using the same mechanisms based on authenticated and delegated attributes. *Direct access* rules (e.g. "`[role:` *`admin`*`]`") target the authenticated attributes of the task invoker, which can be a user or a remote space. *Indirect local access* rules (e.g. "`[localAccess:` *`delegation`*`, DA.role:` *`admin`*`]`") aim at controlling indirect user access from a local service, while *indirect remote access* rules (e.g. "`[userId:` *`Space1`*`, DA.role:` *`admin`*`]`") target both the delegated attributes of the original user and the authenticated attributes of the space from where the task was issued. Indirect access rules must always include at least one authenticated attribute of the invoking space in the subject, because delegated attributes are not verified by the target space. They are just treated as additional information provided by a trusted invoker. By adding such a rule, an administrator explicitly establishes a trust relation to the subjects that are identified by the given authenticated attributes. Finally, *internal access* within the own space is always permitted via an implicit default rule on the subject "`[localAccess:` *`system`*`]`", which also enables the configuration of initial access control rules when starting up the space. This rule cannot be removed to prevent that local administrators are locked out of the policy container.

A written request can be fetched using a `type` selector for a single entry of a specific type that can be processed by the service. Although not shown in Figure 4, this `take` operation is also executed via the space runtime and the coordination layer, which permits the operation due to the default rule on internal access. In the example, the service's job is to take an entry with a specific key from a data container. To ensure that the indirect access of the subject to this container is authorized, the service adds the delegated attributes from the request entry to the task object that is passed to the runtime. As the service accesses a local data container and contains delegated attributes, it is implicitly authenticated by the authentication aspect using the indirect local access mode. Similar to the `write` operation for the service request, the coordination layer invokes the access manager to check if the operation is permitted for the associated subject. If the service accessed a remote container instead, the local space would authenticate to the remote space just like a regular client. Beside the verifiable SSO token, the delegated attributes of the user are included in the context to support indirect remote access rules at the target.

When a service has finished its execution, the response is written to the response container using the unique request identifier as key for a `key` coordinator and the user identifier from the security attributes as label for a `label` coordinator. As for fetching the request, this internal operation is permitted by default. The `key` coordinator enables clients to retrieve a specific response to any of their (possibly concurrent) requests, while the `label` coordinator allows access control rules that prevent clients from accessing responses for other clients. Such a rule may permit `take` operations on the response container using the context-aware scope query "`label($userId)`". For the final `take` operation by the client, the authentication aspect and the access manager are again involved as for writing the request. As the `userId` context variable is an authenticated attribute, the rule ensures that a subject may only take a response if the corresponding request was written by the same subject.

Using the presented architecture, data access authorization is possible via direct and indirect access rules, while service access authorization is realized via direct access rules on request containers. Only certain trusted subjects, which may, for instance, represent software agents from the same organization as the space server, may be allowed to access other containers directly. To avoid bypassing the service framework, container creation and deletion as well as aspect operations are disabled for external sources. Our approach exploits the flexible coordination features of XVSM: The access control model is bootstrapped using XVSM's own query mechanism, whereas the authorization architecture that enforces this model is bootstrapped using operations on XVSM containers. Finally, the Secure Service Space, which allows a combination of data- and service-oriented security, is built upon this secured space.

6 Firewall Management Use Case

To demonstrate the feasibility of the Secure Service Space, we describe an architecture for the management of distributed firewalls. Administrators configure and monitor firewalls via a common *Security Management Center* (SMC) instead of directly accessing the firewalls' local administration interfaces. This centralized approach enables the SMC to update several firewalls at once in a unified way while checking for inconsistencies among the configurations. Even if the firewalls belong to different organizations with separate administrators, a common management center may provide an added value via aggregated information from all managed firewalls, like intrusion alerts for a distributed intrusion detection system. For the example, we restrict the SMC functionality to three use cases:

- **Monitoring:** Firewalls push different types of events (e.g. internal errors or intrusion alerts) to the SMC, which provides administrators with relevant information in a possibly aggregated form.
- **Configuration:** Administrators read and modify configurations for one or more firewalls. The SMC checks the consistency of these configurations and prepares them for deployment.
- **Deployment:** The modified configurations are deployed to the associated firewalls at once, which accept their new configurations and accordingly update their internal settings.

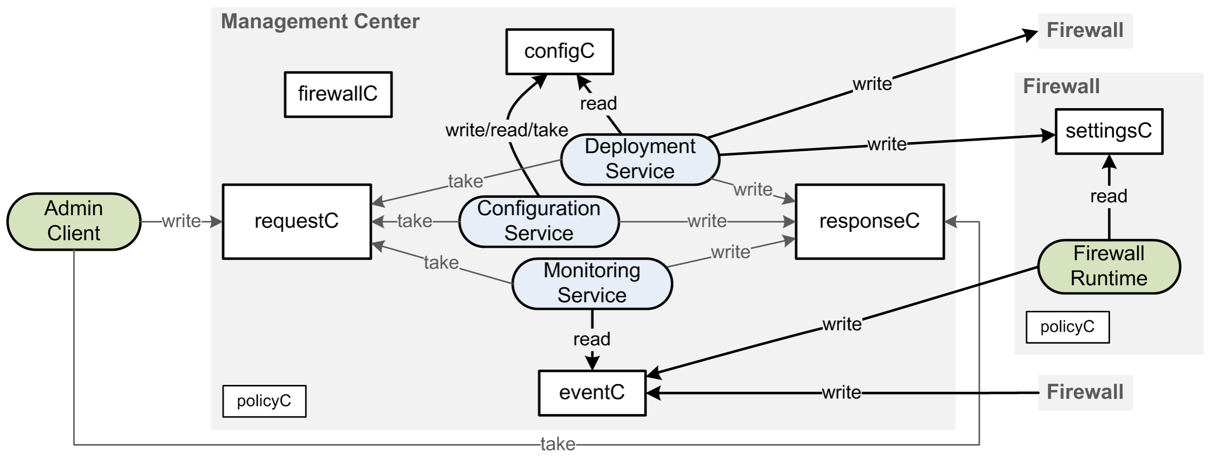


Figure 5: Distributed firewall management scenario realized with Secure Service Space

When controlling security-relevant infrastructure like firewalls, a high level of security must be guaranteed for the access to management functions and data. Therefore, administrators should not be granted full access to the SMC but only to the functionality they require according to their authenticated security attributes. They should be able to modify configurations only for firewalls of their own organization or even only for a single device, while triggering the deployment process may be restricted to experienced administrators with a special role. The internal data of the SMC must be protected from unauthorized access. Thus, administrators may only access these data indirectly via trusted service components. Aditionally, the SMC and the firewalls must be mutually authenticated to preserve the integrity of the system. Only firewalls that are registered to be managed by the SMC should be able to access it. To prevent infected firewalls from affecting the management of other devices, firewalls only have restricted permissions on the SMC so that they may only report their events but not manipulate events or configurations of other devices. All these security requirements can be fulfilled by applying a Secure Service Space architecture with appropriate access control rules on the involved space containers.

Figure 5 shows a possible solution for the firewall management scenario using a Secure Service Space architecture. Administrators may manage firewalls by using an administration client that invokes the SMC via its request and response containers (`requestC` and `responseC`). The use cases are realized via three SMC services that access internal containers as well as external containers on the firewalls. The configuration service supports reading and modifying configurations for specific firewalls by writing and selecting respective entries to and from the internal configuration container (`configC`). When the deployment service is invoked, it reads from this container to retrieve the modified configurations and pushes them to the remote firewall containers (`settingsC`) of all affected firewalls. The firewall runtime processes change their behavior according to the modified settings in their internal containers. If a firewall needs to report some kind of event, the runtime process writes the data into the event container (`eventC`) of the SMC. On request, the monitoring service returns available events to the client. An additional container (`firewallC`) stores entries for each firewall that is managed by the SMC.

For defining the access control policies for this scenario, we apply very simple rule combination semantics by using the `PERMIT-OVERRIDES` combination algorithm together with a default rule that denies access to all containers for all subjects. Every access is denied unless explicitly allowed by another rule. Thus, we are able to use sets of only `PERMIT` rules, which are stored in the respective policy containers (`policyC`) of the SMC and the firewalls, to specify the permissions on the containers. For users of the administration client, we specify two roles: a regular `admin` role and a `seniorAdmin` role with additional privileges. Assuming that senior administrators are able to adopt both roles, service access rules for the SMC based on the request type may look as follows:

RULE monitoringRule
SUBJECTS: [role: admin]
RESOURCES: requestC
ACTIONS: write
CONDITION: -
SCOPE: type(monitorReq)
EFFECT: PERMIT

RULE configurationRule
SUBJECTS: [role: admin]
RESOURCES: requestC
ACTIONS: write
CONDITION: -
SCOPE: type(configureReq)
EFFECT: PERMIT

RULE deploymentRule
SUBJECTS: [role: seniorAdmin]
RESOURCES: requestC
ACTIONS: write
CONDITION: -
SCOPE: type(deployReq)
EFFECT: PERMIT

Regular administrators may only write requests for the monitoring and the configuration service, whereas senior administrators can also invoke the deployment service. For more fine-grained rules that depend on the data that is accessed by the invoked services, data access rules are needed. In contrast to administrators, firewalls may access SMC containers directly, but their role only allows restricted access to push their events. The indirect local access of administrators to the configuration container and the direct access of firewalls to the event container can be regulated via the following rules:

RULE configAccessRuleX
SUBJECTS: [localAccess: delegation,
DA.role: admin, DA.affiliation: X]
RESOURCES: configC
ACTIONS: write, read, take
CONDITION: -
SCOPE: key(FW24) $\cup$ key(FW42)
EFFECT: PERMIT

RULE eventPushRule
SUBJECTS: [role: firewall]
RESOURCES: eventC
ACTIONS: write
CONDITION: firewallC | label($userId)
SCOPE: query(source = $userId)
EFFECT: PERMIT

The first rule specifies that any configuration entry that is indirectly accessed by the configuration service on behalf of an authenticated administrator with a particular affiliation `X` is restricted to two entries with specific keys, which correspond to the identifiers of the firewalls that are managed by this organization. Entries can neither be written nor queried using any other key. Similar rules can be established for all involved organizations. The second rule restricts access for firewalls on the event container. Firewalls may only write events with an internal `source` field that corresponds to the authenticated firewall's user identifier. This prevents that infected devices may tamper events from other firewalls. An additional condition on the SMC's firewall container checks if the authenticated firewall is even registered at the management center, which can be done by a local SMC administrator with appropriate permissions on `firewallC` by writing a registration entry for the firewall using its identifier as label. Administrators must also be permitted to indirectly read the event container, which can be achieved using indirect local rules on `read` operations with a scope like "`query(source = FW11)`" for a specific affiliation.

When pushing new configurations to remote firewalls, also the SMC must prove its identity to the firewall. Thus, the deployment service authenticates with the identity of the own space when accessing the settings containers. Access control rules at the firewalls may permit full access to the SMC role. However, if only configurations from specific SMC users should be accepted, an indirect remote access rule with a subject like "`[role: smc, DA.userId: Admin24]`" can be used at the firewall.

Although the presented example policies do not cover all possible security constraints of a distributed firewall management scenario, we have shown how the Secure Service Space can be utilized to define flexible access control rules for realistic use cases. Service access rules enable coarse-grained permissions based on the invoked operations, while data access rules allow to refine privileges based on the content of accessed entries and contextual information. Rules for the SMC depend both on the user context (e.g. a user's affiliation) and on the environmental context (e.g. a required registration entry for firewalls). To extend this model, additional rules and roles may be added. As shown in Figure 2, constraints can be specified based on how sensitive the accessible information is. As security violations can be transparently masked by the coordination layer of XVSM, services like the monitoring service can simply invoke a query "`any(ALL)`" to retrieve all permitted entries for a specific subject. Thus, neither the client nor the service needs to be adapted if permissions change. As all internal and external communication is decoupled via space containers, new services can be easily added that are able to collaborate with existing components in an ad-hoc way, for example to remove unnecessary events from the event container. To grant access for such a service, only two simple rules must be written to the policy container. One rule allows writing the new request type to the request container, while the other one regulates the indirect data access to the event container.

By using coordination services instead of directly integrating the space operations into the clients, we provide an additional abstraction layer that hides the implementation of the coordination logic from the clients. Thus, changed requirements only affect a single service instead of several different client implementations. The space-based service architecture also provides distribution transparency for the involved clients. The SMC could be located on a single site or on an internal network of several machines, which could enhance system performance. Implicit load balancing and enhanced reliability can be achieved by using several services of the same type. As access control rules only target entries within containers, the distribution of services does not require policy changes if we assume that the machines on the internal network share a common authenticated attribute (e.g. "trustedNode") that is also added by the authentication aspect for local access. This attribute can then be included in rule subjects instead of concrete invoker identities. If container locations are moved or even replicated for performance and availability reasons, the corresponding rules can simply be copied to the policy container of the target space. Thus, a flexible and maintainable SMC architecture can be created using the Secure Service Space.

An alternative SMC implementation using a traditional service-oriented architecture would provide its functionality via Web services that synchronize their data via a database and trigger additional services at firewalls to deploy configurations. Application servers have the capability to configure the authorization of services. However, if authorization depends on the types of data a user may access, application logic needs to be added. Service programmers should furthermore not be responsible for permissions on a database that may be managed by different staff. A better way would be to delegate the authorization decision to the database, which however employs a different mechanism for specifying permissions on its tables. In contrast, the Secure Service Space enables the specification of permissions both on service and data access using the same mechanism. A simpler space-based authorization approach using only ACLs on containers would also not support the complex requirements of the SMC use cases, as shown by the example rules that depend on dynamically evaluated scopes and conditions. Therefore, this example shows the benefit of expressive XVSM access control rules in a realistic scenario.

7 Discussion and Related Work

XVSM and its access control model enable developers and administrators to coordinate processes and to specify fine-grained access control policies using similar concepts. The model supports *coordination-aware authorization* as rules are based on an extensible query language that is also used for selecting entries, which simplifies the synchronization of business logic and security policies. Additionally, *authorization-aware coordination* is achieved because coordinators may ignore denied entries and return accessible alternatives if available. Thus, changed permissions remain transparent for the clients. The Secure Service Space adds a high-level programming model based on reusable service components that are decoupled via the space. Application developers need not cope with low-level coordination protocols because they can invoke services via simple request objects instead. When these internal protocols are modified due to requirement changes, or data and services are distributed to optimize the system, the clients are not affected. Thus, a highly scalable and maintainable system architecture is achieved. Independent protection of services and resources with the same mechanisms enables maximum flexibility for defining access control policies.

Our bootstrapped authorization architecture provides high scalability, whereas the computational overhead for the dynamic rule evaluation depends on the complexity of the policy, as tested with a Java implementation of XVSM[1] described in [1]. Thus, the tradeoff between complex security constraints and performance requirements has to be analyzed when designing a distributed application.

In related work, different mechanisms have been used to add security to space-based middleware. *KLAIM* [24] defines typed access rights that are statically enforced before execution, which is not sufficient if dynamically changing policies should be supported. *SecOS* [16] and *SecSpaces* [17] use a capability-based approach where tuples are locked with keys that must be known by clients that need to access them. WSSecSpaces [15] offers SecSpaces as a Web service that enables the coordination of other services in a decoupled way. The Secure Service Space also provides coordination services but on a higher abstraction layer than simple tuple space operations. In *Lindacap* [18], so-called multicapabilities are distributed, which allow access to all tuples of a specific space partition that match a certain Linda template. However, capabilities offer only limited control over the active permissions as they are difficult to revoke and may be shared among clients. If administrators want to effectively manage access rights on a space, a central management point as provided by the XVSM policy container seems more appropriate.

MARS [25] extends the Linda model by adding so-called reactions to the tuple space, which include instructions that are executed when certain operations are performed on specific tuple types by particular agents. Similar to XVSM aspects, reactions may influence the outcome of the space operation. Security is supported via ACLs on tuple spaces and tuples, where roles are mapped to permitted space operations on these entities. In XVSM, more complex permissions based on data content and context are supported.

LGL [19] has introduced a content-based authorization approach for Linda tuple spaces. Prolog rules with included Linda templates form a global law that defines permissions on the space. These rules are triggered either by invocation events that correspond to a space operation where the included entry or template matches the rule's template, or by selection events that occur when entries that match the rule's template are read or taken. The Secure Service Space provides a similar approach by allowing rules based on data and service requests. By managing a control state for each process, LGL also supports context-aware rules. LGI [26] provides a decentralized authorization architecture for LGL via trusted controllers, whereas our enforcement mechanism is directly integrated in the middleware. TuCSoN [7] defines distributed spaces via nodes that are connected in a tree-like topology. Gateway nodes control the visibility of their child nodes and are able to enforce authorization for them. Such a delegated authorization approach may also be interesting to apply for the Secure Service Space. Similar to XVSM, a bootstrapped architecture is chosen to enforce authorization in TuCSoN via reactions that are triggered by certain space operations. Prolog-based RBAC policies are supported for defining fine-grained permissions on allowed interaction patterns [20]. For XVSM and the Secure Service Space, Prolog is not necessary as glue code for the specification of expressive rules as the extensible query mechanism provides enough flexibility to enable both complex coordination queries and fine-grained permissions.

The SMEPP middleware [14] provides high-level services on top of the SecureLime [27] tuple space middleware to ease the development of P2P applications. For invoking services, a space-based approach via invocation and reply tuples is chosen similar to the Secure Service Space model. SMEPP is secured via pre-shared symmetric keys that protect specific groups of services, while our approach allows dynamically changeable and more fine-grained permissions on services and data.

In summary, related approaches for secure spaces either provide lower expressiveness or require dif-

[1]`http://www.mozartspaces.org`

ferent concepts for authorization and coordination. While some space-based service architectures already exist, they only offer basic authorization mechanisms that target permissions on service invocations. The Secure Service Space allows the definition of fine-grained access control policies on services and on the accessed data while using the same concepts for selecting and protecting entries within a space.

8 Conclusion

In this paper, we have presented a secure space-based framework for collaborating services. The communication between clients and services is decoupled via a space-based middleware that enables complex coordination mechanisms based on extensible queries. Moreover, the services may access space containers to model state changes and communicate with other services and external components. As the service framework and service resources are modeled using the same concepts, a unified access control mechanism can be applied for regulating both service invocations and data access. The middleware's own query language is used to express fine-grained access control rules that target the used service request, the properties of accessed data entries, as well as the context of the user and the environment. The authorization architecture is bootstrapped with operations on space containers, which facilitates concise framework semantics. Also the management of administrator privileges is realized via a policy container that can be modified and secured like regular data. Thus, we argue that the introduced Secure Service Space architecture provides a lightweight alternative to common service-oriented architectures for coordination services that connect decoupled components in an ad-hoc way.

In future work, we plan to extend our service model towards a full-featured service execution framework. We also want to adapt the security model to support distributed policies where rules can be located on spaces different from the one that enforces them. Finally, we will investigate the feasibility of our approach for the definition of secure and dependable coordination patterns that can be used to implement distributed applications with high reliability, for business software as well as for embedded systems.

9 Acknowledgments

The work is funded by the Austrian Government under the programs FFG FIT-IT, project 825750 Secure Space, and FFG Bridge, project 834162 LOPONODE Middleware.

References

[1] S. Craß, T. Dönz, G. Joskowicz, and e. Kühn, "A coordination-driven authorization framework for space containers," in *Proc. of the 7th International Conference on Availability, Reliability and Security (ARES'12), Prague, Czech Republic*. IEEE, August 2012, pp. 133–142.

[2] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, January 1985.

[3] L. Aldred, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede, "On the notion of coupling in communication middleware," in *Proc. of On the Move to Meaningful Internet Systems 2005: CoopIS, COA, and ODBASE (OTM'05) - Part II, Agia Napa, Cyprus, LNCS*, vol. 3761. Springer-Verlag, October-November 2005, pp. 1015–1033.

[4] R. Mordinyi, e. Kühn, and A. Schatten, "Space-Based Architectures as Abstraction Layer for Distributed Business Applications," in *Proc. of the 4th International Conference on Complex, Intelligent and Software Intensive Systems (CISIS'10), Krakow, Poland.* IEEE, February 2010, pp. 47–53.

[5] e. Kühn, R. Mordinyi, L. Keszthelyi, and C. Schreiber, "Introducing the concept of customizable structured spaces for agent coordination in the production automation domain," in *Proc. of the 8th International*

Conference on Autonomous Agents and Multiagent Systems (AAMAS'09) - Volume 1, Budapest, Hungary. IFAAMAS, May 2009, pp. 625–632.

[6] C. Bryce and M. Cremonini, "Coordination and Security on the Internet," in *Coordination of Internet Agents: Models, Technologies, and Applications*, A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, Eds. Springer-Verlag, 2001, pp. 274–298.

[7] M. Cremonini, A. Omicini, and F. Zambonelli, "Coordination and Access Control in Open Distributed Agent Systems: The TuCSoN Approach," in *Proc. of the 4th International Conference on Coordination Languages and Models (COORDINATION'00), Limassol, Cyprus, LNCS*, vol. 1906. Springer-Verlag, September 2000, pp. 99–114.

[8] D. Martin, D. Wutke, T. Scheibler, and F. Leymann, "An EAI Pattern-Based Comparison of Spaces and Messaging," in *Proc. of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC'07), Annapolis, Maryland, USA*. IEEE, October 2007, pp. 511–518.

[9] e. Kühn, J. Riemer, R. Mordinyi, and L. Lechner, "Integration of XVSM Spaces with the Web to Meet the Challenging Interaction Demands in Pervasive Scenarios," *Ubiquitous Computing And Communication Journal, special issue on Coordination in Pervasive Environments*, vol. 3, March 2008.

[10] S. Craß, e. Kühn, and G. Salzer, "Algebraic foundation of a data model for an extensible space-based collaboration protocol," in *Proc. of the 13th International Database Engineering & Applications Symposium (IDEAS'09), Cetraro, Calabria, Italy*. ACM, September 2009, pp. 301–306.

[11] OASIS, "eXtensible Access Control Markup Language (XACML) v2.0," OASIS Standard, February 2005, http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf.

[12] S. Craß and e. Kühn, "A coordination-based access control model for space-based computing," in *Proc. of the 27th Annual ACM Symposium on Applied Computing (SAC'12), Riva, Trento, Italy*. ACM, March 2012, pp. 1560–1562.

[13] D. Wutke, D. Martin, and F. Leymann, "Facilitating Complex Web Service Interactions through a Tuplespace Binding," in *Proc. of the 8th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS'08), Oslo, Norway, LNCS*, vol. 5053. Springer-Verlag, June 2008, pp. 275–280.

[14] F. Benigni, A. Brogi, J.-L. Buchholz, J.-M. Jacquet, J. Lange, and R. Popescu, "Secure P2P programming on top of tuple spaces," in *Proc. of the 17th IEEE International Workshops on Enabling Technologies (WETICE'08), Rome, Italy*. IEEE, June 2008, pp. 54–59.

[15] R. Lucchi and G. Zavattaro, "WSSecSpaces: a secure data-driven coordination service for Web Services applications," in *Proc. of the 19th Annual ACM Symposium on Applied Computing (SAC'04), Nicosia, Cyprus*. ACM, March 2004, pp. 487–491.

[16] J. Vitek, C. Bryce, and M. Oriol, "Coordinating processes with secure spaces," *Science of Computer Programming*, vol. 46, no. 1-2, pp. 163–193, January-February 2003.

[17] R. Gorrieri, R. Lucchi, and G. Zavattaro, "Supporting Secure Coordination in SecSpaces," *Fundamenta Informaticae*, vol. 73, no. 4, pp. 479–506, October 2006.

[18] N. I. Udzir, A. M. Wood, and J. L. Jacob, "Coordination with multicapabilities," *Science of Computer Programming*, vol. 64, no. 2, pp. 205–222, January 2007.

[19] N. H. Minsky and J. Leichter, "Law-Governed Linda as a Coordination Model," in *Proc. Selected papers of the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Bologna, Italy, LNCS*, vol. 924. Springer-Verlag, July 1995, pp. 125–146.

[20] A. Omicini, A. Ricci, and M. Viroli, "RBAC for Organisation and Security in an Agent Coordination Infrastructure," *Electronic Notes in Theoretical Computer Science*, vol. 128, no. 5, pp. 65–85, May 2005.

[21] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *Computer*, vol. 29, no. 2, pp. 38–47, February 1996.

[22] E. Yuan and J. Tong, "Attributed Based Access Control (ABAC) for Web Services," in *Proc. of the 2005 IEEE International Conference on Web Services (ICWS '05), Orlando, Florida, USA*. IEEE, July 2005, pp. 561–569.

[23] OASIS, "Core and hierarchical role based access control (RBAC) profile of XACML v2.0," OASIS Standard, February 2005, http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-rbac-profile1-spec-os.pdf.

[24] R. de Nicola, G. L. Ferrari, and R. Pugliese, "KLAIM: A Kernel Language for Agents Interaction and Mo-

bility," *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 315–330, May 1998.

[25] G. Cabri, L. Leonardi, and F. Zambonelli, "MARS: A Programmable Coordination Architecture for Mobile Agents," *IEEE Internet Computing*, vol. 4, no. 4, pp. 26–35, July-August 2000.

[26] N. H. Minsky, Y. M. Minsky, and V. Ungureanu, "Making tuple spaces safe for heterogeneous distributed systems," in *Proc. of the 2000 ACM Symposium on Applied Computing (SAC'00) - Volume 1, Como, Italy*. ACM, March 2000, pp. 218–226.

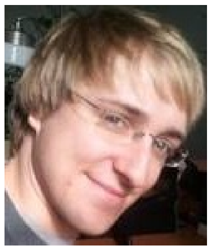
[27] R. Handorean and G.-C. Roman, "Secure sharing of tuple spaces in ad hoc settings," *Electronic Notes in Theoretical Computer Science*, vol. 85, no. 3, pp. 122–141, August 2003.

Stefan Craß is a research assistant at the Institute of Computer Languages of the Vienna University of Technology. He is currently working on his PhD thesis on secure and dependable coordination spaces. Further research interests include modular middleware architectures, space-based collaboration protocols, access control models and wireless sensor networks. He holds a master's degree in Software Engineering and Internet Computing from Vienna University of Technology.

Tobias Dönz is a research assistant at the Institute of Computer Languages of the Vienna University of Technology. He is the main developer of the XVSM implementation MozartSpaces and his interests include other space-based middleware, distributed systems architecture and concurrent programming. He holds a master's degree in Software Engineering and Internet Computing from Vienna University of Technology.

Gerson Joskowicz is a senior researcher at the Institute of Computer Languages of the Vienna University of Technology. He specializes in enterprise integration and security of space-based systems. He is a graduated engineer in electrical engineering (Dipl.-Ing.) of the Vienna University of Technology and a Ph.D. (Dr. techn.) in biomedical engineering of the Graz University of Technology.

eva Kühn is the leading scientist of the Space-Based Computing Group at Vienna University of Technology's Institute of Computer Languages and works as project coordinator of nationally and internationally funded research projects. International publications comprise the areas of multi-database integration, heterogeneous transaction processing, parallel and distributed programming, coordination languages, space-based computing, self-organization and software architectures. She is a graduated engineer of computer sciences (Dipl.-Ing.), Ph.D. (Dr. techn.) and Venia Docendi (Univ. Doz.) from the Vienna University of Technology. For her Ph.D. thesis on "Multi Database Systems", she has received the Heinz Zemanek research award. She holds international patents and has industrial experience as CTO of two TU Vienna spin-off companies. She is also a member of the Senate of the Christian Doppler Research Association, and a member of the Science and Research Council of the Federal State of Salzburg (Austria).



Alexander Marek is a PhD student at the Institute of Computer Languages of the Vienna University of Technology. His current projects include space-based middleware for wireless sensor networks and the development of a modeling environment for space-based middleware interactions. He holds a master's degree in Software Engineering and Internet Computing from Vienna University of Technology.