SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Registration number: FEI-5382-92313

IMPLEMENTATION OF ADDITIONAL FUNCTIONALITY FOR AUTHORIZATION SERVER 'MYSTABLE'

BACHELOR'S THESIS

2021 **Alica Ondreáková**

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Registration number: FEI-5382-92313

IMPLEMENTATION OF ADDITIONAL FUNCTIONALITY FOR AUTHORIZATION SERVER 'MYSTABLE'

BACHELOR'S THESIS

Study Programme:	Applied Informatics
Study Field:	Computer Science
Training Workplace:	Institute of Computer Science and Mathematics
Supervisor:	Mgr. Ing. Matúš Jókay, PhD.

Bratislava 2021 **Alica Ondreáková**

Slovak University of Technology in Bratislava
Institute of Computer Science and Mathematics

Faculty of Electrical Engineering and Information Technology
Academic year: 2020/2021
Reg. No.: FEI-5382-92313

BACHELOR THESIS TOPIC

Student:	**Alica Ondreáková**
Student's ID:	92313
Study programme:	Applied Informatics
Study field:	Computer Science
Thesis supervisor:	Mgr. Ing. Matúš Jókay, PhD.
Workplace:	Institute of Computer Science and Mathematics

Topic: **Implementation of Additional Functionality for Authorization Server 'mYstable'**

Language of thesis: English

Specification of Assignment:

The main goal of the thesis is to extend the functionality of the mYstable authorization server for Medusa project. Medusa implements decision-making in the kernel for authorizing actions of processes. Implementation of the security model depends on an authorization server. So far, there is only one functional implementation of the authorization server (in C language), which is too complicated to extend (lack of documentation, own configuration language). For this reason, a new authorization server called mYstable was created. It's written in Python programming language to allow rapid prototyping of new features.

Tasks:
1) Study of Medusa communication protocol.
2) Study of the actual implementation of mYstable server.
3) Propose an implementation of additional functionality for the mYstable authorization server.
4) Implement proposed ideas.
5) Evaluate the benefits of the project.

Selected bibliography:

1. Káčer, J. – Jókay, M. *Medúza DS9.* Diplomová práca. Bratislava : FEI STU, 2014. 35 p.
2. Kirka, J. – Jókay, M. *Autorizačný server mYstable pre projekt Medusa.* Diplomová práca. 2018. 36 p.

Assignment procedure from:	15. 02. 2021
Date of thesis submission:	04. 06. 2021

Alica Ondreáková
Student

Dr. rer. nat. Martin Drozda	**Dr. rer. nat. Martin Drozda**
Head of department	Study programme supervisor

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Alica Ondreáková
Bakalárska práca:	Rozšírenie funkcionality autorizačného serveru mYstable
Vedúci záverečnej práce:	Mgr. Ing. Matúš Jókay, PhD.
Miesto a rok predloženia práce:	Bratislava 2021

Medúza je bezpečnostý modul pre Linux, ktorý slúži ako rozšírenie základných bezpečnostných politík. Skladá sa z dvoch hlavných komponentov: monitora virtuálnych svetov a autorizačného servera v užívateľskom priestore. Originálna implementácia autorizačného servera je projekt Constable napísaný v programovacom jazyku C. Táto práca sa zaoberá implementáciou jednotného priestoru mien (podľa vzoru Constabla) do novej implementácie autorizačného servera v jazyku Python s názvom mYstable. Cieľom práce je pripraviť modul, ktorý by bolo možné v blízkej budúcnosti jednoducho integrovať do projektu mYstable.

Kľúčové slová: Medúza, Autorizačný server, Bezpečnostná politika Linuxu, mYstable, Constable, Python

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Author:	Alica Ondreáková
Bachelor's thesis:	Implementation of Additional Functionality for Authorization Server 'mYstable'
Supervisor:	Mgr. Ing. Matúš Jókay, PhD.
Place and year of submission:	Bratislava 2021

Medusa is a security module for Linux distributions that serves as an extension of basic Linux security features. It is formed of two main components, the kernel-space monitor and user-space authorization server. The original implementation of the authorization server is called Constable, it's written in the C programming language. This thesis deals with the implementation of a unified namespace (based on Constable) into the new implementation of the authorization server. This implementation is based on Constable's abstract concepts. This new server is called mYstable and it's written in the Python programming language. The main goal of this thesis is to prepare a module that implements these abstract concepts so they can be later integrated into the mYstable server.

Keywords: Medusa, Authorization server, Linux security, mYstable, Constable, Python

Acknowledgments

I would like to express a gratitude to my thesis supervisor.

Contents

List of Figures and Tables

List of Abbreviations

AS	Authorization Server
DAG	Directed Acyclic Graph
DFS	Depth-First Search
LSM	Linux Security Module
NLTK	Natural Language Toolkit
OS	Operating System
PID	Process Identifier
RBAC	Role-Based Access Control
UTF-8	Unicode Transformation Format (8-bit)
UUID	Universal Unique IDentifier according to RFC 4122

Introduction

Medusa is a security module for Linux distributions that serves as an extension of basic Linux security features. Standard security features of Linux include user and group separation, file security, and root access. However, these basic security features have proven to be insufficient. Medusa provides a sufficient level of audition, through a set of rules that need to be upheld in order to execute operations. Medusa consists of two basic parts: the Authorization server and the Monitor of virtual spaces. The Monitor of virtual worlds resides in the kernel and is tasked with monitoring operations. An operation is invoked by a subject and carried out on an object. For example, when deleting a file through a file manager, the file manager acts as the subject and the file as the object. We can describe the virtual world as an access matrix composed of objects and subjects [1]. The decision of whether to permit or deny an operation is carried out by the Authorization server. The server lies in the user-space. A huge advantage of Medusa is that just a small portion of it resides in the kernel [1]. Thanks to this, Medusa is easy to maintain and port while preserving kernel compatibility. This compatibility is achieved via a communication protocol which sends information about the kinds of objects and subjects supported by the kernel after each start of the system.

The original authorization server of Medusa was Constable, written in C. But due to a large code-base and a lack of documentation in addition to many more issues, it was harder to prototype new functionality. Instead, a new authorization server was proposed, called **mYstable**. Currently, mYstable lacks much of Constable's original features. The main goal of this thesis is to implement a domain model functionality, available in the original Constable server, into mYstable. The first chapter introduces project Medusa and Constable in more detail. The second chapter describes our implementation of the domain model into mYstable. The third and final chapter compares the implemented mYstable domain model implementation against the original implementation in Constable, and gives guidelines for future development of mYstable.

1 Project Medusa

Medusa was developed by Marek Zelem [1] and Milan Pikula [2]. It aimed to provide a security patch for the kernel. Back in the day, many security modules (patches) were in development including SELinux, Rsbac and Grsec/Pax. Each module provided a unique approach towards security and kernel modification. During the 2001 Linux Kernel summit, it was proposed that SELinux should be the official security module for Linux. However, this idea was rejected by Linus Torvalds who observed that many promising modules are still in development.[1] Instead of making SELinux official security module, it was proposed that a unique interface should be created so any module can be part of the kernel. Thus the Linux security module was born, which provided a sufficient mechanism to implement mandatory access control by different security modules.

1.1 Monitor of virtual spaces

This chapter is based on [1]. The monitor of virtual spaces resides in the kernel and can be described as an access matrix composing of objects and subjects.

- **Objects**—One object can reside inside one or more virtual spaces. There is also a possibility that the object doesn't belong anywhere, but this scenario is not very useful. Each object has defined a bit set, representing membership (or lack thereof) in virtual spaces. This is called the domain of an object. We will represent this domain as $\boldsymbol{OVS(o)}$.
- **Subjects**—Subjects have virtual world membership defined by a separate bit set for each transaction type. For example, the file manager can read, write and see a file. We will represent this set as $\boldsymbol{SVS_a(s)}$, where a is the transaction type.

Transactions that can be carried out on object are as follows:

1. Read—subject can read content of the object,
2. Write—subject can modify content of the object,
3. See—subject can obtain information about the existence of the object.

When a subject attempts to perform a transaction on an object, the monitor evaluates the operation as follows. If the object and subject don't share a virtual world intersection for

[1] `https://en.wikipedia.org/wiki/Linux_Security_Modules#History`

the transaction type, the request is denied. If they share at least one intersection of virtual spaces, the kernel can send (depending on the configuration) a query to the authorization server. The server then sends a response back to the kernel. So the necessary condition to execute the operation is as follows:

$$\boldsymbol{SVS_a(s) \cap OVS(o) \neq \emptyset}$$

1.2 Communication protocol

Following chapter is based on [2] and [3]. The kernel communicates with the authorization server via a special device—`/dev/medusa`. The communication scheme is a simple request–answer scheme. When an action is invoked, a transaction request is sent from the kernel to the authorization server. The server can then process the request accordingly. The request carries all the necessary information about the subject and object requesting the transaction, such as rights, domain or set of domains, name, and the unique id of the event. This attribute description is called the `cinfo`.

The efficiency of communication protocol lies in its abstraction. Thanks to the high level of abstraction, Medusa can communicate with any kernel that supports the communication protocol. This abstraction is defined in terms of k-objects and events.

- **k-objects**—Short for kernel objects, these represent system functions and structures that are supported by the kernel. A simple example can be an i-node that describes file/directory properties. Structure description can vary from kernel to kernel. During the fresh start/initialization of the authorization server, communication protocol translates this structure into a universal k-object representation. Every k-object should implement at least one of the methods `fetch` and `update`. Both of them are requests made by authorization server. When invoked by authorization server `fetch` delivers **k-object** information from kernel to authorization server. `Update` is invoked when server wants to update information about **k-object** in kernel.

- **events**—Events represent system calls into the kernel, for example `open`, `read`, `exec` and `kill`. The authorization server uses these to monitor system behaviour, subject to policy. However, while most events have an associated subject, there are special cases of system calls that are responsible for the initialization of new k-objects that the authorization server has not seen before, and do not have associated subjects. These are `getfile`, `getprocess`, `getipc` and `getsocket`, and they let the authorization server know that new objects were created.

1.3 Authorization server

The authorization server resides in the user-space and decides whether a transaction will be permitted or denied if the monitor can't decide. Also, the server is responsible for initialization and modification of objects and abilities of subjects. The transactions of the authorization server are not being monitored to avoid deadlock. Deadlock occurs when multiple processes are running simultaneously and holding each other's resources, hence effectively blocking one another. This risk can occur when running the server locally. When an operation is waiting for a decision, the operation is put on hold, and is blocked. This means that the authorization server process would need permission from itself to continue but is being blocked by itself. The benefits of being outside of the kernel are uncomplicated allocation and file accessibility [1]. The first implementation of the authorization server was "Constable", written in C. But due to many issues like lack of documentation and slow prototyping, a new server was proposed. This server is "mYstable" and it is written in Python.

The authorization server is responsible for initialization, decision-making, and modification when needed. It handles the following events:

1. **AS initialization**—System using the Medusa communication protocol sends data to the authorization server about k-objects and events. Thus, we ensure that data inside the authorization server are up to date with those in the kernel.

2. **K-objects initialization**—When a new k-object is created inside the kernel, the communication protocol sends it to the authorization server. The authorization server can then place this object into one or more virtual space.

3. **Transaction approval**—When a transaction needs approval, the kernel sends a transaction request.

The authorization server answers transaction requests in the following manner:

1. **Deny**—Denial of operation.

2. **OK**—Decision will be made according to Unix security policy.

It's worth mentioning that before the LSM (Linux Security Module) framework was implemented into the Linux kernel, two more server answers were available, Allow and Skip. **Skip** gave users fake information about the success of the transaction. The authorization server denied the operation however the user would think of it as a successful one. The **Allow** response forced the transaction even though it was against basic Unix security policy. If we would like to implement Allow and Skip, we would need to get around Linux Security Module architecture. As a direct consequence, our project would be rendered useless as we couldn't integrate it into official Linux kernel stream.

1.3.1 Constable

This chapter is based on [1]. Constable was developed by Marek Zelem in 2001. A big emphasis was aimed at the modularity of the server and abstraction. The Constable's structure is presented in Figure 1.

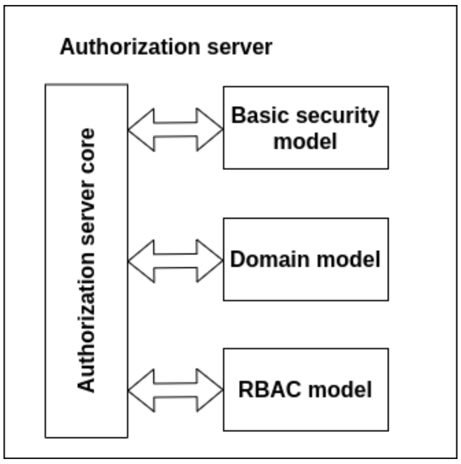

Figure 1: Structure of Constable authorization server [1]

- **Authorization server core**—The core of the authorization server is a mediator between different security modules and the kernel. Each module communicates with the core via an abstract mechanism. This abstract mechanism is uniform for all modules.

    1. **Virtual spaces management**—Each object can belong to one or more virtual spaces. This is represented by a per-object bit set, where each bit represents one world.
    2. **Unified namespace**—Serves as an abstraction for object identification. In this context, an object can be everything from a file to a process. In an ideal scenario, everything would be a file, thus we could use the file system to explicitly identify an object. However, this is not the case. For example, processes are identified by their PID numbers. Constable solves this by creating a new

name hierarchy root, identified by an empty string, where the filesystem branch is user namable (see an example in Figure 2).

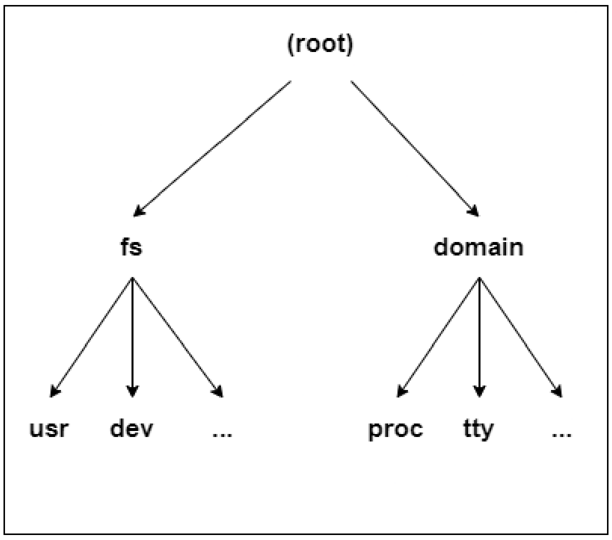

Figure 2: Example of unified namespace structure for domain model

3. **Access abstraction**—Access rights are defined by an access matrix, where rows are subjects and columns objects. Operation represents the rights of a subject, currently used are read, write and see. User can expand on these basic with erase, control, create, enter by configuration.
4. **Target control**—Objects can be logically grouped in virtual spaces, independently of their location in the name hierarchy. These are also called targets in older implementations. In later implementations targets were renamed to spaces.
5. **Object management interface**—This interface provides communication with the kernel. Necessary information about objects and subjects is transferred in this way. The structure of this information depends on the kernel. The structure can be identified via a header.
6. **Subject management**—The authorization server can be composed of multiple security modules, each with its own policy. Each module initializes and places the subject in virtual space according to its policy. By the union of these virtual spaces, we get definitive rights of the subject.
7. **Event distribution**—When a kernel action is invoked (e.g. a process tries to write some data to a file), all modules receive an event. The event carries all the necessary information about the invoked operation. When this event

operation falls into the jurisdiction of a module, its service function is called. Service function is event handler of a concrete module. Modules decision is based on the module's policies. Modules send their answers to the core, which then chooses the most restrictive answer.

8. **Translation and interpretation of configuration file**—An important part of abstraction via which we can define targets and assign rights to operations. A target is an enumeration of all paths in the hierarchy towards an object or a branch. Branches are distinguished from objects by the **recursive** keyword. Subjects that belong to a target inherit their rights.

- **Basic security model**—The name hierarchy contains processes in the `proc` branch. This branch includes all processes of the system. The hierarchy of processes is based on file execution. When a process invokes the `exec` syscall on a file, it is moved to the child node corresponding to the file's target.

- **Domain model**—The name hierarchy contains domains in the `all_domains` branch. Every process belongs to exactly one domain. We can set the rights of each domain and thus set the rights of its processes.

- **RBAC model**—This model incorporates the RBAC[2] module into the authorization server. RBAC is a role-based approach towards security policy. RBAC branch in the unified namespace is based on the RBAC role hierarchy. This model works according to its own configuration file.

1.3.2 mYstable

mYstable is the second implementation of the authorization server for Medusa system implemented in the Python programming language. Constable lacked the readability and documentation, which complicated further development and porting. The main concepts behind the new authorization server are readability, portability, and maintainability. Readability and portability are achieved by switching to a high-level programming language (Python 3.x). A high-level language is easy to read and provides a higher level of abstraction [4]. However, the code is interpreted on every run and so it is slower than compiled code. Thus usage of some improved algorithms is necessary. The code maintainability depends on the adherence to basic standards and maximizing usage of built-in modules.

[2]`https://en.wikipedia.org/wiki/Role-based_access_control`

Python programming language and its abundance of external modules from mathematical ones like NumPy to natural language processing ones like NLTK come in handy for mathematicians and programmers alike. However, when it comes to lifespan, some of the smaller modules fail to live upon the expectations. There are numerous causes, like funding, competition, and migration of useful parts to a more complex module. Because of this, the best practice is to either use modules with active community and documentation or use built-in modules.

Momentarily, there is one external module used in mYstable, the `treelib`. As the name suggests, it implements data types and algorithms for a tree data structure. In this structure, we store all file paths from our system, which mYstable can use. Another new (internal) module is `graphlib`, which we use to manage virtual spaces. Graphlib is a built-in module hence it's stable. The modules will be explained in more detail in chapter 3.1.

2 Spaces management

As mentioned in the section above, main goals of our thesis are the following: implementation of the unified namespace, virtual world management, and target control. Even though they are three different abstract concepts, they are connected. Their implementation is thus tangled and a unified name for these abstract concepts would be in order. The virtual world is a space with a name, where we store an accessible folder called the path in actual implementation or specific groups of folders (virtual spaces) called spaces. But we can also add or subtract new paths and spaces, turning spaces on and off if desired. We are managing the life cycle of the space from initialization to the desired end, so it would be fitting to call these three abstract concepts spaces management.

Primarily we will describe the abstract model that we are trying to implement. Secondly, we will go over abstract concepts used. Opening with tree data structure where we look at practical uses of the tree and its role in our abstract model. Then we will get over one type of linear ordering that is important for us, as we will use this linear ordering in concepts to come. The most important concept we will be backing by this ordering is a *Directed Acyclic Graph* (DAG). In short, DAG is an abstract data structure that we will use to overcome a cycle problem in the abstract model. Finally, we will get over algorithms used for topological sorting that will help us define our ideal graph.

2.1 Space Model

In this section, we will describe the abstract model of spaces management modeling some situations via diagrams that will respond to the model's functionality. Firstly, we will get over a lifecycle of space, we will use an activity diagram for this. Then we will move on to the state diagram and explain it.

Space is a set of objects that share the same rights. Each object should belong to at least one space. However, this is policy dependant. Here, object represents everything from a process, a socket, an inter-process communication object to a file. If the policy doesn't include an object in the system, then the excluded object is being ignored. Spaces can inherit or mask-out (remove) objects from other (sub)spaces. The inheriting and masking relations are transitive, asymmetrical, and mutually exclusive. In our actual implementation masking takes precedence, so when both inheriting and masking the same space, the result will be that the set will be masked.

When processing space declarations of spaces sequentially, the inheritance hierarchy is not expanded greedily. Instead, it is deferred until all space declarations are seen.

This allows inheriting spaces that are declared out of order. Deferral of inheritance is done by building an inheritance graph. When the declaration is finished, we try to build topological order for this graph. Topological sort can succeed only when the graph is *acyclic*. Similarly, a graph and corresponding topological order is built for masking objects. The whole process is shown in Figure 3.

Let's consider the following situation (Figure 4), where we transitively inherit space C but explicitly mask it. In this example we use letters to express spaces and numbers for paths. We can alternatively write this situation down like this (the Constable authorization server uses this notation for space addition and removal):

```
space A = path1, path2
        + space B
        - space C
space B = path3, path4, space C
space C = path4, path5
```

Figure 4 represents evaluated inheritance and masking. This evaluated figure shows the final membership for paths. In the end, space A inherited path3, as we wanted to exclude paths that belonged to space C. Path 4 and 5 were masked, as they belonged to space C. We can also explain this via set theory, but firstly we will define some formal notations down bellow.

- Let Θ be set of all spaces
- Let $\mathcal{O}$ be set of all paths
- Let $\alpha(A)$ denote the direct members of space A, $\alpha\colon \Theta \to 2^{\mathcal{O}}$
- Let $\omega(A)$ denote the effective members of space A after evaluating space inheritance and masking, $\omega\colon \Theta \to 2^{\mathcal{O}}$
- Let $\phi(A)$ denotes to all spaces that are masked by A
- Let $\gamma(A)$ denotes to all spaces that are inherited by A

Putting all of this together we will get the following equation 1.

$$\omega(A) = \Big[\alpha(A) \cup \bigcup_{B \in \gamma(A)} \omega(B)\Big] \setminus \bigcup_{C \in \phi(A)} \omega(C) \tag{1}$$

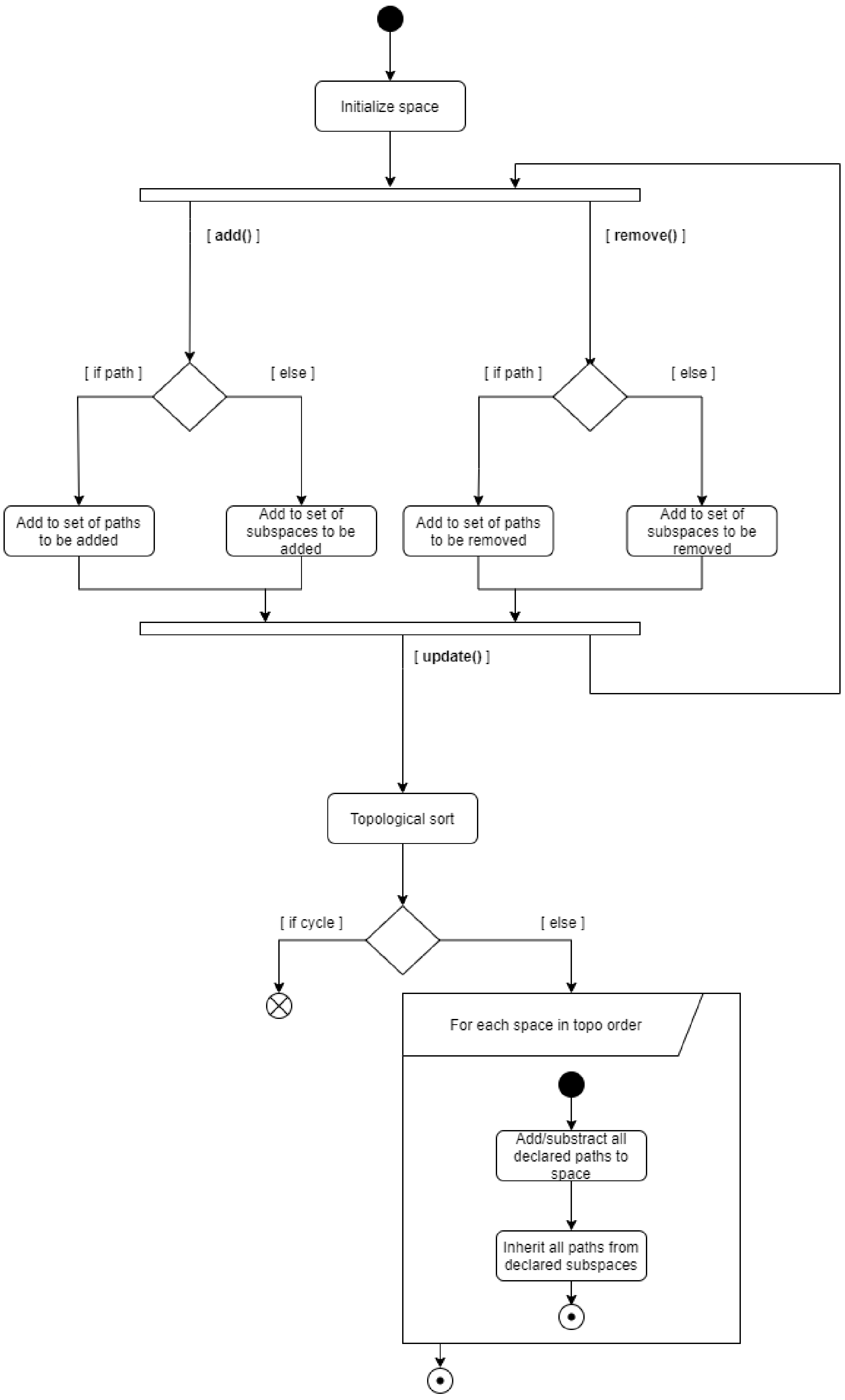
Initialize space
[add()]
[remove()]
[if path]
[else]
[if path]
[else]
Add to set of paths to be added
Add to set of subspaces to be added
Add to set of paths to be removed
Add to set of subspaces to be removed
[update()]
Topological sort
[if cycle]
[else]
For each space in topo order
Add/substract all declared paths to space
Inherit all paths from declared subspaces

Figure 3: Activity diagram for space

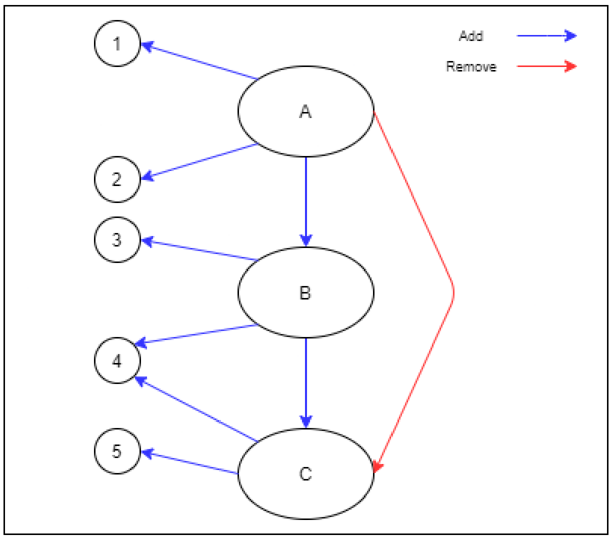

Figure 4: The flow of addition and recursive removal of space before being processed

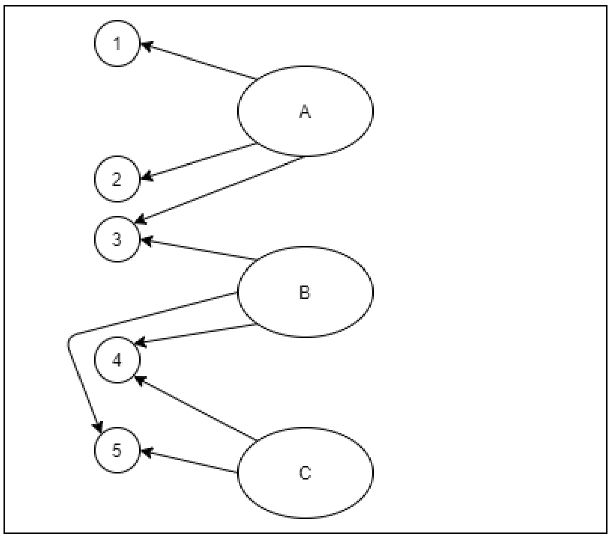

Figure 5: The flow of addition and recursive removal of space after processing

Substituting the graph in Figure 5, we get:

$$\begin{aligned}
&\alpha(A) = \{1, 2\}, && \alpha(B) = \{3, 4\}, && \alpha(C) = \{4, 5\} \\
&\gamma(A) = \{B\}, && \gamma(B) = \{C\}, && \gamma(C) = \{\} \\
&\phi(A) = \{C\}, && \phi(B) = \{\}, && \phi(C) = \{\} \\
&\omega(C) = \alpha(C) = \{4, 5\} \\
&\omega(B) = [\alpha(B) \cup \omega(C)] = \{3, 4, 5\} \\
&\omega(A) = [\alpha(A) \cup \omega(B)] \setminus \omega(C) = \{1, 2, 3\}
\end{aligned}$$

In Figure 6, we can see an example of direct cycle. This is another situation that can occur while working with spaces.

```
space A = path1, path2
        + space B
space B = path3, path4
        + space C
space C = path5, path6
        + space A
```

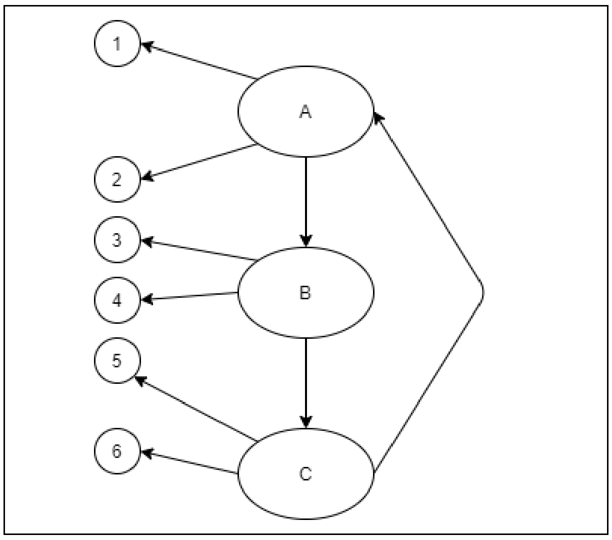

Figure 6: Example of a cycle

2.2 Tree

A tree is an abstract data structure that resembles tree roots due to its hierarchical structure. Tree consists of one node called the root and its descendants. In contrast to other data structures like arrays, stacks, and lists, tree is a nonlinear structure. A nonlinear data structure can't be sequentially ordered because one node can have more than one relationship (1...n). On the other hand, trees are convenient when it comes to expressing the hierarchy of items. If we would like to put our file system in a data structure, we would use a tree data structure.

2.3 Topological ordering

Topological ordering is a linear ordering of vertices, where for every directed edge uv, u comes before v in the ordering. A necessary condition for this sorting is that graph has no direct cycles. Most of the algorithms used for the construction of topological ordering have a linear running time. We will describe two of them more precisely down below in section 2.5.

For a DAG graph, there always exists at least one valid topological ordering. Topological ordering is thus a valid sequence of tasks in which they get executed. Imagine washing machine dial, where basic programs are labeled, each consist of subtasks. We can read the dial from longest to shortest program, or from left to right and vice versa. All orderings from above are correct, as there is more than one valid topological order in this case.

2.4 Directed Acyclic Graph

A graph is a nonlinear data structure that consists of vertices that can be somehow related to each other. We express this relationship with edges that connect these vertices in question. There are two types of edges: directed and undirected. For example, two people hold affection for each other. This can be expressed by an undirected edge as the emotion goes both ways. In the case of unrequited affection, it is a direct one.

The Directed Acyclic Graph (DAG) is a graph with directed edges and no cycles. A necessary condition for any graph to be DAG is that its vertices can be topologically ordered (see section 2.3 about topological ordering).

2.5 Algorithms

Algorithms used for topological sorting also check if there is a cycle present. This is convenient, as the necessary condition for graph's vertices to be topologically ordered is that there is no direct cycle in the graph. Firstly, we will introduce a lesser-known algorithm and then present a variant of a well-known one.

Kahn's algorithm

1. Firstly, we compute the degree of each vertex in the graph. Vertex degree is the number of incoming edges towards a given vertex. Then we check if any zero-degree vertices are present and add them to a queue.

2. We pop a vertex and add it to topological order, then we decrement the degree of adjoining vertices. Then we check again for any zero-degree vertices and add them to the queue. If a vertex is successfully processed, we increment our vertice counter.

After all of the vertices are processed, we either return the topological order or throw cycle error. If the value inside of the vertice counter is greater than the number of vertices of the graph, we return cycle error. So if there is a cycle present, we will process at least one vertex more than once, thus incrementing our vertice counter.

Depth-first search (DFS)

The DFS algorithm starts at any given random vertex and moves down the path until it reaches the end and then backtracks. At first, all nodes are labelled as "unvisited". We will use stack and list data structures. We start at a random vertex and put it in our list, and place all its adjacents vertices in the stack. If we can't continue, then we take value from the top of the stack and pop it, then place it into the "visited" list. We search if this value from the top of the stack has any new adjacent vertices. If so, we put them in the stack. We continue this process until the stack is empty.

3 Implementation

This section is devoted to actual implementation that is based on section 2 and its subsections. Firstly, we will introduce modules that we used for the implementation of abstract structures (more on them in the subsections of 2). Then we will take a look at classes and their functions. We will describe each class and explain technologies used in them.

3.1 Modules

This chapter is based on treelib documentation[3] and actual implementation. Treelib is an external module that implements a tree data structure. It supports common tree operations like traversing, insertion, and deletion. The module is compatible with both Python 2 and 3. This Python module has two main classes: `Node` and `Tree`. We used this module to implement tree hierarchy from paths. The tree structure is explained in more detail in section 2.2. For the sake of coming subsections, we will explain few methods from this module. Firstly, we will explain `create_node` method and its arguments.

```
create_node(tag : string, identifier : bytes, parent : bytes, data : Node)
                                      --> Node
```

- Tag is a readable name for the node. In our implementation, we use components of the path.
- The unique ID of a node is generated via UUID library, if identifier is absent. Our identifier is in this case 8 byte hash created from path string.
- Parent node signifies, to which parent does child belong. Unique identifier of parent, 8 byte hash is used).
- User defined data associated with this node.

```
get_node(nid : bytes) --> Node
```

- This method will fetch us the object (node) if it is present in our tree, otherwise returns `None`. In our case, nid is our 8 byte long hash created from path.

Graplib is a built-in module that provides us with the functionality for topological sorting of graph nodes. The structure and possible usage of this graph is described in

[3] `https://treelib.readthedocs.io/en/latest/index.html`

more detail in section 2.4. Also the topological ordering is explained in section 2.3 and some algorithms are explained that can be used to evaluate the graph nodes in section 2.5. This built-in module consists of two classes. For our project, the most important one is `TopologicalSorter` class and its methods of DAG creation and handling. The second class we are using in our actual implementation is `CycleError`. This class throws an exception when there is a direct cycle detected. More on this situation in section 2.1. This situation is also shown in Figure 6. There can be more than one cycle present in the DAG, but `CycleError` returns just one.

The third important module we use is `hashlib`, a built-in module used for implementing a common interface for different hashes and message digest algorithms[4]. Some well-known hashing algorithms like md5, sha256, sha512, and many others are integrated into this module. Hash is an algorithm that takes an input of any length and converts it to a byte array of fixed size. It is of course impossible to generate unique values if the length of the hash output is possibly smaller than the length of the input, and hash collisions can occur. The length of the output array depends on the used hash algorithm. The output byte array is created from a unique set of characters that are computed by a given hash algorithm. We create hash via md5 constructor which is explained down bellow.

```
hashlib.md5(usedforsecurity : bool = False)
```

- Default value of `usedforsecurity` argument is `True`. `False` value indicates that hash algorithm is used as one-way and non-cryptographic function, this is also our case.

3.2 Namespace

Namespace class does the heaviest lifting when it comes to space management. Spaces are initialized here, checked for cycles, and then masked accordingly. Technologies used in `Namespace` class are explained in subsections of section 2.

Firstly, we need to initialize and store spaces so we can manipulate them later on. This is done by `space()` function. We will use a dictionary data structure to map space names to their indices (`name_index_map`). As a key, we use the name of a given space (string) and as the value index of a given space (integer). Every time a new name is detected, we increment the counter. The function works in two modes, based on the boolean value of the create argument. When a new space name is first seen (`create=True`), we assign to

[4]`https://docs.python.org/3/library/hashlib.html`

it a unique index and initialize its data (represented by class `VirtualSpace`, see section 3.4). Otherwise, we return the unique index of this space (`create=False`).

Secondly, we need to manage those spaces we created. We need to be able to add paths and spaces to a given space or alternatively subtract them. For this purpose, we use `space_add()` and `space_sub()` functions. These functions just add data of paths and spaces that should be subtracted or added to a given space. Masking is not done here, as we still didn't check for cycles and other problems. We use these functions to fill instances of given space that are managed by `VirtualSpace` class. The arguments of these functions are following: `(s_name, *paths)`. Firstly, we find the index of the given space `s_name`, so we can access its properties and modify them. Also, we need to be able to differentiate if the second argument is a path or space. We do this by checking if the argument starts with a backslash ("\"). If this is the case, we insert the argument inside `nodes_add` else to `subspaces_add` list.

Also, we need to build a tree from paths provided in spaces. This is being done by `_node_for_path()` function. We transform each component of the path to a node that can be placed inside of our abstract tree data structure. More on tree data structure in section 2.2. Each node has a unique identifier (more in subsection 3.1). We use this identifier to check if the current path is already represented as a node. If this identifier is absent from our tree, we continue to create this new node. Firstly, we split this path into smaller pieces called components. As a separator character we use "\". After this split, we check if the given subcomponent has a parent. If not, parent is absent and thus we are looking at the root. Otherwise we take the parent node and slowly build up to its nodes that are not explored. Each node gets a unique hash value so we can differentiate between them.

We also want a unique identifier for each node that is part of the tree. This will be achieved by hashing the given path, which is the name of the node. We will use the md5 hash algorithm for our purposes and digest its first 8 bytes. Also, before we start hashing, we must encode our path to UTF-8 encoding. In the (rare) case of hash collision, we repeat the hashing process until an unique hash is produced. Constructor of `hashlib` module is explained in subsection 3.1. Firstly, we invoke the constructor to initialize the hasher. Then we feed our hasher with input, in our case the whole length of encoded path via `update()` method. This method can also take parts of our input string, as we don't need to feed it full length to be able to produce output. In the end, we create hash from our path calling the `digest()` method and take the first 8 bytes (64 bits) from it.

We also need to check for cycles. If a cycle would be present we would get stuck and that's unwanted behavior. We solve this problem by using properties of the directed acyclic graph (DAG) (more in section 2.4). The necessary condition for a graph to be DAG is that it is acyclic and thus at least one topological ordering exists (explained in section 2.3). For this purpose, we utilized a built-in module `graphlib` (more above modules used in subsection 3.1). We do this graph building in `_build_topo()` function. We iterate over each space in its inherited `subspaces_add` and `subspaces_sub` lists. Our starting vertex is first space in `vs` variable. We then iterate over all of its subspaces and connect them with directed edges. When finished, we move to the next space and its subspaces and do the same. If space doesn't exist we return an error that given space doesn't exist. Finally, we try to evaluate this graph. If any cycle is present, we return an error, otherwise we return valid topological order and can move on space masking.

After we finish filling out spaces, we can move to `space_update()` function. Firstly, we try to build topological order by calling `_build_topo()` from all spaces and their respective subspaces. When this operation is successful we take this valid topological order and iterate over it. Topological ordering is just a valid sequence of spaces. Firstly, we take all the paths of this space that are stored in `nodes_add` list and mask them via `_space_add_real()` function. Then we iterate over subspaces of a given space and do the same for them. The same thing is done for the unmasking of spaces. The reason why we have `_space_update()` to do the masking is that we can't be sure if all spaces are initialized when we are inserting new inheritance to them. If this would be the case and we would want to fill a space not yet initialized we would fail. Thus one function was created that runs in the end after the process of initialization and filling of spaces is completed.

3.3 Node

The `Node` class is responsible for setting spaces on and off, based on actions invoked in the `Namespace` class. Each path is represented by a node that has its set of bits. This set of bits represent spaces, each bit is one space. If the bit is one (value 1), it means that a given node belongs to space represented by this bit, otherwise it's set to zero (value 0), thus node is absent from this space. We use functions `spaces_mask_set()` and `spaces_mask_unset()` to set bit in question on/off. Via (`1` $\ll$ `space_id`) we retrieve bitmask, in this mask just bit (`space_id + 1`) is turned on. This bit represents the space we are turning on. Then we make a logical OR operation with the original bitmask of the given node. Of course, this is the case when we set the mask on, however, resetting the mask is similar. This works for any number of spaces as integers in Python.

3.4 VirtualSpace

Virtual space class holds properties of space like the name, its index and so on. We use this information to check if space already exists and return its properties (in `Namespace` class 3.2). Here we also differentiate between paths and spaces and if they are to be masked or not. We have two lists for nodes: `nodes_add` containing nodes that should be masked for space (turned on) and `nodes_sub` containing those that should be unmasked (turned off). Similarly, we have two lists for subspaces of a given space. Later on, we use this information from subspaces to build a topological order from our spaces. Thanks to this, we check for possible cycle error in our implementation.

Conclusion

The first chapter of this thesis was devoted to Medusa and its communication protocol. Firstly, we explained what is Medusa, a module that provides mandatory access control for different security policies. Then we explained in more detail the monitor that is used for these purposes and its transactions. Next, we moved on to the communication protocol and its abstraction. This protocol is based on a simple answer-request scheme, that provides communication between the monitor and authorization server. Also, the authorization server's purpose and its duties in this scheme were explained. In order to better understand the new implementation of the new mYstable server, we dove into concepts and abstraction of the original authorization server—**Constable**. Based on these abstract concepts, we proposed a model of our own which implemented the following functionality: unified namespace, virtual world management and target control. The new authorization server **mYstable** aimed to be modular as possible and abstract as the old Constable server. After we evaluated our space model and unified name space for abstract concepts that we wanted to implement, we moved on to actual implementation.

Implementation maximally utilized abstract structures that best mirrored the needs of the new authorization server and its speed. As the new authorization server is written in Python instead of C, this was necessary to maximize the efficiency of the new server.

Finally, we evaluated the benefits of the new server in contrast to Constable. The new server is easy to use, doesn't lack documentation, and is written in Python. Python language is easier to read and thus easier to maintain. In the future, this space model should be integrated into the mYstable server.

Resumé

Cieľom práce je implementácia novej funkcionality pre autorizačný server *mYstable*. Tento server je druhou implementáciou autorizačného servera pre bezpečnostný modul Medúza. Medúza bola originálne napísaná v roku 2002 dvoma študentami: Marekom Zelemom a Milanom Pikulom. Slúžila ako sada bezpečnostných záplat do jadra. V čase písania Medúzy bolo navrhnutých viacero bezpečnostných systémov v jadre Linuxu (SELinux, Rsbac, Grsec/Pax . . .), ale ani jeden z nich nebol v tom čase súčasťou oficiálnej distribúcie jadra.

Medúza sa skladá z dvoch častí: monitora virtuálnych svetov a autorizačného servera. Monitor virtuálnych svetov sa nachádza v jadre OS a vieme ho opísať ako maticu prístupov pozostávajúcu z objektov a subjektov. Riadky tejto matice sú subjekty a stĺpce tvoria objekty. V prípade, že pod subjektom sa nachádza objekt, znamená to, že subjekt má právo s týmto objektom manipulovať.

V aktuálnej implementácii, ktorá je v súlade s bezpečnostnou politikou Linuxu, rozlišujeme nasledovné oprávnenia: čítanie (*Read*), zápis (*Write*) a „videnie“ (*See*; ide o právo zistenia existencie objektu). Druhou časťou Medúzy je autorizačný server, ktorý rozhoduje, či bude nejaká operácia vykonaná alebo zamietnutá. Komunikáciu medzi týmito dvoma časťami sprostredkúva komunikačný protokol. Jadro komunikuje s autorizačným serverom za pomoci špeciálneho zariadenia `/dev/medusa`. Komunikačná schéma je jednoduchá: požiadavka a odpoveď. Komunikačný protokol týmto pádom slúži ako prostredník medzi monitorom, ktorý sa nachádza v jadre a serverom, ktorý beží v užívateľskom prostredí. V prípade, že je potrebné spracovať požiadavku, komunikačný protokol doručí všetky potrebné infomácie o danej transkacii autorizačnému serveru. Na základe týchto informacií sa potom vie autorizačný server rozhodnúť (povoliť alebo zamietnuť operáciu).

Komunikačný protokol rozlišuje medzi kernel objektami (tzv. *k-objects*) a udalosťami (tzv. *events*). Kernel objekt je abstracia pre objekt nachádazujúci sa v jadre OS. Takže sa jedná o dáta štruktúr, ktoré sú podporované daným jadrom OS. Na zistenie informácií o konkrétnom kernel objekte môže autorizačný server zavolať metódu `fetch`, ktorá doručí z jadra OS všetky informácie o objekte. V prípade, že chce autorizačný server zmeniť údaje v jadre OS, môže tak urobiť pomocou metódy `update`. Udalosti zasa reprezentujú aktivity, ktoré je možné na strane jadra OS monitorovať (napríklad systémové volania ako `open`, `read`, `exec` a `kill`). Niektoré udalosti majú špeciálny význam – ich cieľom je informovať autorizačný server o vzniku nového objektu jadra. Na takéto udalosti by mal

autorizačný server reagovať správnym nastavením bezpečnostnej štruktúry vzniknutého objektu v jadre OS.

Úlohy autorizačného servera sú teda nasledovné: inicializácia bezpečnostnej politiky na strane dátových štruktúr autorizačného servera, inicializácia nových k-objektov a rozhodovanie o povolení či zamietnutí aktivít na strane jadra OS. Inicializácia autorizačného servera spočíva v získaní všetkých potrebných informácií o k-objektoch a udalostiach riadeného jadra OS. Inicializácia nových k-objektov bola spomenutá vyššie. Autorizačný server može odpovedať na nejakú žiadosť o rozhodnutie (transakcia) nasledovne: `Deny` (zamietnutie transakcie) alebo `OK` (povolenie transakcie).

Momentálne existujú dve implementácie autorizačného servera: *Constable* a *mYstable*. *Constable* je originálny server pre Medúzu, napísaný v programovacom jazyku C jedným z pôvodných autorov Medúzy, Marekom Zelemom. Skladá sa z jadra autorizačného servera a príslušných bezpečnostných modulov (viď obr. 1 na str. 5). Jadro autorizačného servera slúži ako mediátor medzi rôznymi bezpečnostnými modulmi a jadrom OS. Táto schopnosť plynie z abstrakcie, na ktorej je jadro autorizačného servera postavené. Rozhranie jadra autorizačného servera je uniformné pre všetky bezpečnostné moduly.

Úlohou tejto bakalárskej práce bolo implementovať časť tejto abstrakcie pre autorizačný server *mYstable*, a to správu virtuálnych svetov, jednotný priestor mien a správu cieľov. Momentálne sa bezpečnostný systém Medúza používa len s autorizačným serverom (ďalej len AS) *Constable*, kedže *mYstable* ešte nie je dokončený. V prvom rade sme pri návrhu implementácie časti tejto funkcionality brali do úvahy problémy, ktoré sú prítomné v AS *Constable*. Hlavným cieľom AS *mYstable* je čitateľnosť, prenositeľnosť a ľahko udržiavateľný (modulárny) kód s dostatočnou dokumentáciou. Na základe týchto cieľov sme najprv navrhli abstraktný model pre jednotný priestor mien. Tento abstraktný model slúži pre správu cieľov a správu virtuálnych svetov.

Kedže Python je interpretovaný jazyk, snažili sme sa zamyslieť nad čo najefektívnejšou implementáciou jednotného priestoru mien. O inicializáciu, napĺňanie dátami a ich overovanie sa stará trieda `Namespace`. Na vytvorenie hierarchie ciest, ktoré patria do jednotného priestoru mien, sme využili stromovú štruktúru. Každý uzol stromu predstavuje jednu cestu. Táto cesta má svoj jedinečný identifikátor – 8 bajtový hash. Dĺžka jedinečného identifikátora je obmedzená komunikačným protokolom Medúzy, v rámci ktorého sa aktuálne využíva priestor o veľkosti 8 bajtov na tieto účely.

Každá inštancia triedy virtuálneho sveta `VirtualSpace` má dva zoznamy, v ktorých uchováva cesty jednotného priestoru mien. Jeden zoznam predstavuje cesty, ktoré majú patriť do virtuálneho sveta, druhý zoznam reprezentuje cesty, ktoré do inštancie virtuálneho sveta explicitným spôsobom nepatria. Virtuálny svet môže vo svojej definícii obsahovať okrem ciest aj iné virtuálne svety. Preto musíme kontrolovať, či už daný svet existuje. Ak nie, pridáme ho do globálneho zoznamu inštancii virtuálnych svetov a pridelíme mu poradové číslo. Podobne ako v prípade ciest má inštancia triedy virtuálneho sveta dva zoznamy aj pre iné inštancie virtuálnych svetov: svety, ktoré majú byť súčasťou definície virtuálneho sveta sa nachádzajú v jednom zozname a svety, ktoré z definície sveta majú byť vylúčené sú v druhom zozname.

Pri tomto prístupe (definícia sveta môže obsahovať iný svet) je potrebná kontrola zacyklenia; cyklus nie je povolený. Preto sa pred finálnym vyhodnotením, ktoré cesty jednotného priestoru mien spadajú pod každý z definovaných virtuálnych svetov, robí kontrola. Tá je založená na zostrojení smerového acyklického grafu všetkých virtuálnych svetov. Ak definície virtuálnych svetov neobsahujú cyklus, podarí sa graf zostrojiť a výstupom kontroly je validná topologická sekvencia, na základe ktorej sa postupne vyhodnocuje naplnenie jednotlivých virtuálnych svetov. Funkcionalitu zostrojenia acyklického grafu (a topologickej postupnosti) zabezpečuje vstavaná knižnica jazyka Python `graphlib`.

Trieda `VirtualSpace` teda obsahuje nasledovné atribúty: štyri zoznamy (dva pre cesty, dva pre vnorené virtuálne svety), meno virtuálneho sveta a identifikačné číslo virtuálneho sveta. V rámci komunikačného protokolu Medúzy sa používa na identifikáciu virtuálnych svetov reťazec bitov. Preto na jednoznačnú identifikáciu virtuálneho sveta slúži jeho identifikátor (index do bitového reťazca).

Napokon sme implementovali aj triedu `Node`, ktorá udržiava informácie pre jednotlivé uzly stromu jednotného priestoru mien ohľadom ich príslušnosti do virtuálnych svetov. Na tento účel slúži atribút `spaces_mask`. Ide o reťazec bitov reprezentujúci príslušnosť uzla (cesty v jednotnom priestore mien) do jednotlivých virtuálnych svetov. Ak má nejaký bit v reťazci nastavenú hodnotu 1, znamená to, že uzol patrí do daného virtuálneho sveta.

Na záver konštatujeme, že sme naplnili ciele zadania: navrhli sme a implementovali modul pre správu jednotného priestoru mien, správu virtuálnych svetov a cieľov podľa vzoru AS *Constable*. Modul je pripravený na integráciu s novým AS *mYstable*. Zdrojové kódy sú dostupné na `https://github.com/elis4265/bp1`.

Bibliography

1. ZELEM, Bc. Marek. *Integrácia rôznych bezpečnostných politík OS Linux: Diplomová práca 2001*. Slovenská Technická univerzita v Bratislave, Fakulta Elektrotechniky a Informatiky, 2001.
2. PIKULA, Bc. Milan. *Distribuovaný systém na zvýšenie bezpečnosti heterogénnej počítaťovej siete: Diplomová práca 2002*. Slovenská Technická univerzita v Bratislave, Fakulta Elektrotechniky a Informatiky, 2002.
3. KÁČER, Bc. Ján. *Medúza DS9: Diplomová práca 2014*. Slovenská Technická univerzita v Bratislave, Fakulta Elektrotechniky a Informatiky, 2014.
4. KIRKA, Bc. Juraj. *Autorizačný server mYstable pre projekt Medusa: Diplomová práca 2018*. Slovenská Technická univerzita v Bratislave, Fakulta Elektrotechniky a Informatiky, 2018.

Appendix

A Project Documentation on Remote Repository

GitHub repository for this project can be found here: `https://github.com/elis4265/bp1` and has the following structure:

```
bp1
 |__README.md
 |__test_tree.py
 |__tree.py
 |__tree_structure.py
```

Documentation can be found here `https://github.com/elis4265/bp1/blob/master/README.md`