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

Registration number: FEI-5384-86207

SECURITY POLICY MINING

MASTER'S THESIS

2022 **Bc. Peter Košarník**

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

Registration number: FEI-5384-86207

SECURITY POLICY MINING

MASTER'S THESIS

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

Bratislava 2022 **Bc. Peter Košarník**

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

Faculty of Electrical Engineering and Information Technology
Academic year: 2021/2022
Reg. No.: FEI-5384-86207

MASTER THESIS TOPIC

Student:	**Bc. Peter Košarník**
Student's ID:	86207
Study programme:	Applied Informatics
Study field:	Computer Science
Thesis supervisor:	Mgr. Ing. Matúš Jókay, PhD.
Head of department:	Dr. rer. nat. Martin Drozda
Consultant:	Ing. Roderik Ploszek
Workplace:	Institute of Computer Science and Mathematics

Topic: **Security Policy Mining**

Language of thesis: English

Specification of Assignment:

The candidate's task is to examine an algorithm that creates security policy for an operating system from observed events in the system. Data gathering and security policy testing will take place using the Medusa security module.

Tasks:
1. Analyze current state of the art.
2. Choose and study the algorithm for developing a security policy from the observed events in the OS.
3. Implement your chosen algorithm.
4. Based on the monitored events in the OS, create a policy using the implemented algorithm.
5. Evaluate your findings.

Selected bibliography:

1. XU, Zhongyuan; STOLLER, Scott D. Mining attribute-based access control policies from logs. In: IFIP Annual Conference on Data and Applications Security and Privacy. Springer, Berlin, Heidelberg, 2014. p. 276-291.

Deadline for submission of Master thesis:	13. 05. 2022
Approval of assignment of Master thesis:	02. 05. 2022
Assignment of Master thesis approved by:	prof. RNDr. Gabriel Juhás, PhD. – Study programme supervisor

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Bc. Peter Košarník
Diplomová práca:	Ťažba bezpečnostných politík
Vedúci záverečnej práce:	Mgr. Ing. Matúš Jókay, PhD.
Konzultant:	Ing. Roderik Ploszek
Miesto a rok predloženia práce:	Bratislava 2022

Výskum v oblasti ťažby bezpečnostnej politiky je vykonávaný najmä pre modely riadenia prístupu, ako je ABAC, ReBAC alebo RBAC. V našej práci sa venujeme ťažbe bezpečnostnej politiky pre bezpečnostný modul Medúza, ktorého vývoj je v súčasnosti zameraný na zabezpečenie operačného systému GNU/Linux. Medúza implementuje MAC model riadenia prístupu. Na ťažbu bezpečnostnej politiky sme navrhli vlastný algoritmus, ktorý dokáže rozšíriť už existujúcu bezpečnostnú politiku o nové pravidlá alebo dokáže vytvoriť úplne novú bezpečnostnú politiku od základov. Vstupom do algoritmu je záznamový súbor obsahujúci záznamy žiadosti o prístup ku súborom a odpoveď jadra Linuxu na túto požiadavku. Žiadosti sú vytvárané procesmi, bežiacimi v operačnom systéme. Navrhnutý algoritmus sme použili pri ťažbe bezpečnostnej politiky pre MariaDB mysql server a bezpečnostnú politiku sme vyhodnotili manuálne, a s pomocou *WSC* metriky. Z vyťaženej bezpečnostnej politiky bol na záver, s použitím bezkontextovej gramatiky, vygenerovaný konfiguračný súbor pre autorizačný server Constable.

Kľúčové slová: Linux, ťažba bezpečnostnej politiky, LSM, Medusa, bezpečnosť operačných systémov

ABSTRACT

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

Study Programme:	Applied Informatics
Author:	Bc. Peter Košarník
Master's thesis:	Security Policy Mining
Supervisor:	Mgr. Ing. Matúš Jókay, PhD.
Consultant:	Ing. Roderik Ploszek
Place and year of submission:	Bratislava 2022

The research of security policy mining was primarily carried out on access control models such as ABAC, ReBAC or RBAC. In our thesis, we research the security policy mining for Medusa security module, which is concerned with security in operating system GNU/Linux. Medusa implements the MAC access control model. We developed our own algorithm for mining of security policies, which can either extend the existing policy with new rules or build a new security policy. The input to the algorithm is a log file, whose entries contain information about the issued access requests in the operating system and the decision of the Linux kernel on that access request. The access requests are issued by processes, that run in the operating system. We used the algorithm to mine the security policy for the MariaDB mysql server. We evaluated the mined security policy using the *WSC* quality metric. As the last step, we generated the Constable configuration file from the mined security policy using context-free grammar.

Keywords: Linux, security policy mining, LSM, Medusa, operating system security

Acknowledgments

I would like to express a gratitude to my thesis supervisor Mgr. Ing. Matúš Jókay PhD. and consultant Ing. Roderik Ploszek for disciplined approach towards my thesis and constructive feedback on design ideas. I would like to also express the gratitude for giving me encouragement and motivation when needed, know-how transfer for better understanding of project Medusa domain and technical know-how for better approach of thesis development. Last but not least, the patience during review of my thesis.

Bc. Peter Košarník

Contents

List of Figures and Tables

List of Abbreviations

ABAC	Attribute-based access control
DAC	Discretionary access control
DTO	Data transfer object
JSON	Javascript Object Notation
LSM	Linux Security Modules
MAC	Mandatory access control
RBAC	Role-based access control
ReBAC	Relationship-based access control
VS	virtual spaces

List of listings

Introduction

The security policies in large organizations are hard to manage. The requirements on security are changing, and the security policies need to be kept up to date manually by the system administrators. Manual interaction with security policies can lead to a couple of problems. We can either have a policy that is overly permissive, i.e. it allows access to a resources in a system that the subject should not be able to, or the security policy denies the access to the resources to authorized subjects.

There is also a problem to even create a new security policy from scratch. This requires knowledge about the system and about the processes that are being run in the system.

One of the solutions to the above problems can be to observe the actions that are being performed in the system. Then we can try to find algorithms, whose goal is to produce a new security policy or update the existing security policy with the observed data. The process of creating new security policies from observed actions is also called a policy mining.

Algorithms that are used for policy mining are referred to as policy mining algorithms. In this thesis, we are concerned with exploring the usage of policy mining to generate a security policy, that runs on the operating system GNU/Linux. We chose Medusa security module as the security solution, for which we will try to mine a security policy.

Since Medusa cannot operate without the process in user space, also called the authorization server, we will need to represent the mined security policy in a way that the authorization server can understand. For this thesis, we chose Constable as our authorization server. Therefore, the desired output of our effort should be the configuration file for the authorization server Constable, which represents the mined policy.

The first chapter of this thesis should be an introduction to the Medusa domain. We define the VS model and describe how Medusa enforces the security policy. The chapter follows with the description of the Constable authorization server, the decision process, and the syntax of the configuration.

The second chapter presents the information about the current state of the policy mining, requirements for policy mining algorithms, the evaluation metrics for the mined policies, and the definition of policy mining problem for MAC access control.

The third chapter is the definition of the policy mining problem specifically for Medusa. The last chapter is the practical part, presenting how we approached the policy mining and what results we achieved.

1 Introduction to LSM Medusa

This chapter is an overview of the Medusa security project and the authorization server Constable. We will provide more details about Medusa and Constable in later chapters.

1.1 Mandatory access control

Mandatory access control (MAC) is a type of access control, where decision is made by a central authority. MAC primarily works with two entities — subjects S and objects O. Subjects are entities performing some action on an object. Object is a resource being accessed by a subject. MAC is considered to be non-discretionary access control, meaning that the access is not managed by the object owner as it is with DAC. As an example of usage, MAC access control model is used e.g. in military systems, operating system Linux, etc.

MAC is uniformly used across the whole system and is covering all subjects and objects. MAC applies following restrictions on subjects.

- Subjects cannot pass privilege to other subjects.
- Subjects cannot change attributes of other entities under mandatory access control i.e. other subjects, objects and system components.
- Subjects cannot choose security attributes to be assigned for newly created objects or subjects.
- Subjects cannot change the access control rules.
- Subjects cannot pass information to unauthorized subjects or objects.

MAC can be simultaneously implemented with the DAC. The combination of MAC and DAC adds an additional layer of protection to protect system from elevation of privileges and unauthorized execution of code on behalf of user. This prevents compromising the entire system by compromising a single user or a process [1, 2, 3].

Definition 1 (Access request) *Let subject $s \in S$ be an entity requesting an access to the object $o \in O$ performing an action a. Then $\beta = [s, o, a, m]$ is an access request, where m is a metadata information about the request.*

The metadata m are not explicitly defined because they are context sensitive. However, example of metadata can be timestamp of request, assigned ID of request, failure code, etc.

The access request can be denied or allowed by an access control mechanism according to the security context. A security context is any information assigned to subject, object, or action that is used to determine whether the access request should be allowed or denied.

1.2 Linux Security Modules

Linux Security Modules (LSM) is a lightweight access control framework for GNU/Linux introduced in kernel v2.6. LSM provides an interface for security modules that implement MAC and allows various security operations to be hooked and possibly enforce a special access control behavior.

1.2.1 Motivation

Basic GNU/Linux implements the DAC scheme. The problem with the implementation of DAC in GNU/Linux is that it provides insufficient granularity, since there are only three privileges — read, write, or execute. The problem was solved by introducing capabilities and also other security solutions to enhance security. As a result, many security projects have been developed as patches to enhance security in the Linux kernel [4, 5, 6].

The goal of LSM framework is to unify the functional needs of as many security projects as possible while minimizing the impact on the Linux kernel. This is achieved by allowing to mediate access through hooks, also known as LSM hooks, just ahead of access of the kernel object (Figure 1) [7].

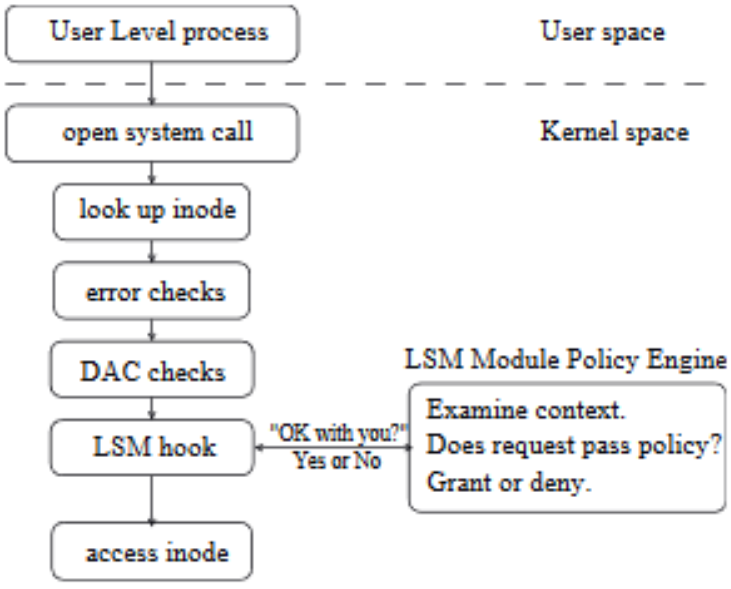

Figure 1: LSM hook architecture (source [7])

1.3 Medusa

Medusa is a security module created by Marek Zelem and Milan Pikula in 1997. Medusa was originally developed as a security patch for the Linux operating system. The development stopped in 2001 and was resumed by Ján Káčer in 2014 when Medusa was ported to 64-bit computer architecture and started to implement the LSM framework [8, 9].

The uniqueness of Medusa comes from dependency on a user-space program called authorization server, which implements the main logic of the security policy. Medusa is the interface on the kernel level to enforce decisions made by the authorization server.

1.3.1 K-objects and events

K-object is an internal Medusa data structure that encapsulates a subset of information of kernel structures. The k-objects vary in information they store and can be split into two main categories:

1. `file_kobject` stores information about the kernel structures representing a file. The information can be a device number, an inode number, etc.

2. `process_kobject` stores information about the kernel structures representing process. The information can be pid, ppid, etc.

Each k-object typically defines two operations — *fetch* and *update*. Operation fetch sets the k-object data according to a kernel structure. Operation *update* applies the changes to k-object to kernel structures. Layer L2 is responsible for mapping of kernel structures to k-objects.

1.3.2 Events

The event is a request from Medusa. The Medysa request can typically demand approval or initialization of security information [10]. Events that demand the approval of an access request are generated each time Medusa cannot provide a decision. The initialization of k-objects includes two special events — `getfile` and `getprocess`. As the name suggests, the `getfile` event is used for initialization of file k-objects while `getprocess` is used to initialize process k-objects [9].

On startup of the authorization server, Medusa does not have any information about k-objects. Therefore, each access request is allowed because Medusa does not have any security context, based on which the decision can be made. Events `getfile` and `getprocess` are called when the given k-object is not initialized. After the events are called, the k-object is processed by the authorization server. The called event depends on the type of processed k-object [10, 9, 11].

1.3.3 Architecture

The architecture of Medusa currently consists of 4 layers [8, 12].

Layer L1 is responsible for registering to system calls of the operating system. Layer L1 on operating system Linux registers the LSM hooks.

Layer L2 is responsible for defining objects that represent abstractions in a given operating system. The defined objects are k-objects, events, access types, etc.

Layer L3 is responsible for registrations of k-classes, used authorization server, and event types

Layer L4, also known as communication layer, is responsible for communication with authorization server.

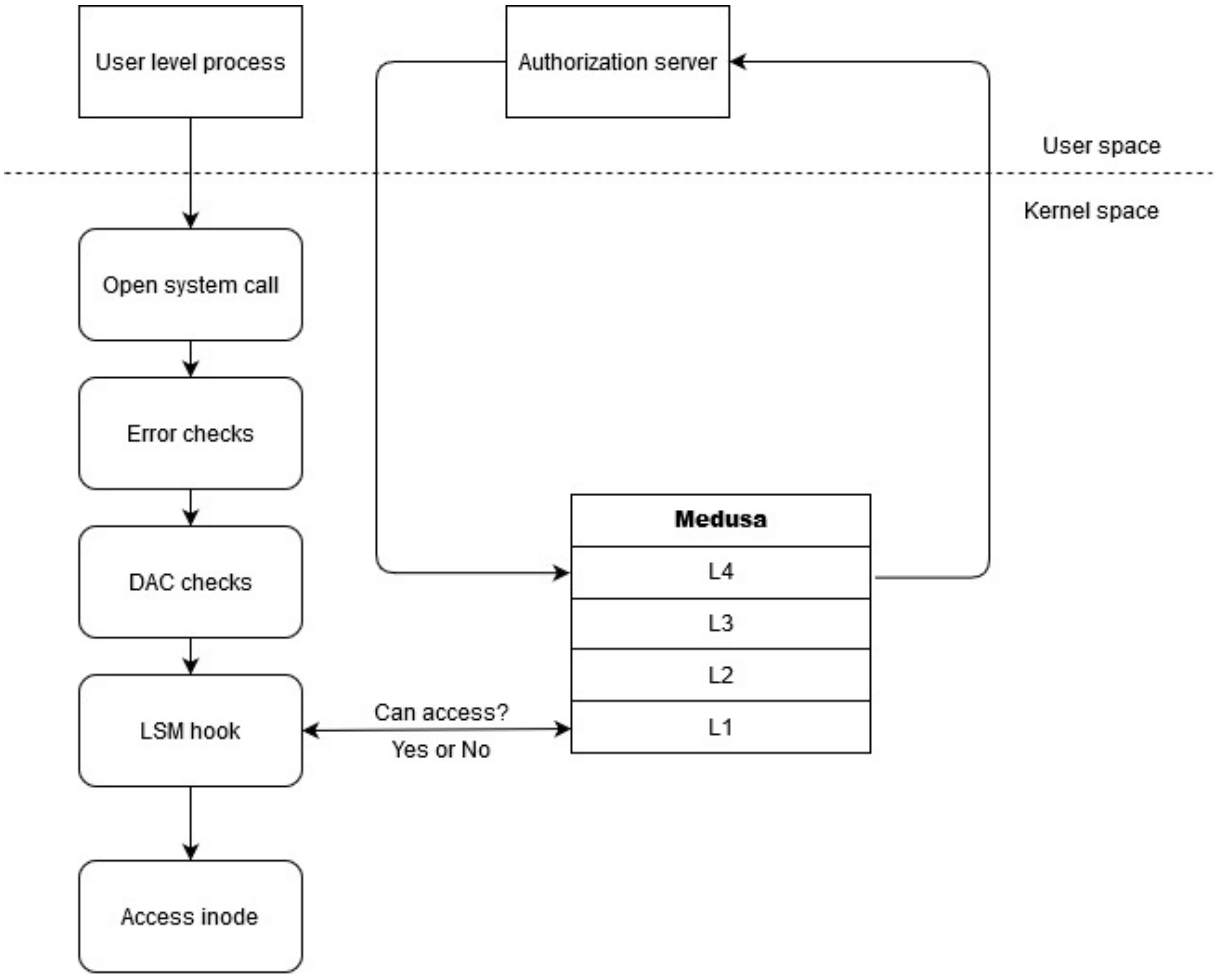

Figure 2: Medusa request processing (source [12])

1.3.4 Access control

The access control of Medusa is based on the Lampson, and Graham and Denning models. The base abstraction model, defines set of subjects $s \in S$, set of objects $o \in O$ and access matrix with access types $a \in A = \{READ, WRITE, SEE\}$. The base abstraction model is described by original authors in [4]. The base abstraction model with access matrix is replaced with VS model in Medusa. The VS model splits subjects and objects into domain groups called virtual spaces (VSs). The Medusa's access control mechanism uses VSs to determine whether access should be granted or denied. Each object and subject need to have a security context assigned. A set of all VS in a system is denoted as VS_{ALL} [4, 13].

Definition 2 (Object security context [4]) *Each object $o \in O$ requires to have assigned a zero, one or multiple VS that it belongs to. A function $OVS(o)$ computes a set $VS^* \subseteq VS_{ALL}$, to which the object o belongs.*

$$OVS(o) \rightarrow VS^* \tag{1}$$

Definition 3 (Subject security context [4]) *Each subject $s \in S$ requires information about the set of VS that it can access. The accessible set of VS for subject, also known as abilities, is computed for each $a \in A$. A function $SVS_a(s)$ computes a set $VS^* \subseteq VS_{ALL}$ (abilities), which represents accessible VSs by subject s, when the access type a is requested.*

$$SVS_a \rightarrow VS^* \tag{2}$$

The definitions 2 and 3 define only the security context. We have not defined how the access of subject to an object can be granted yet.

Definition 4 (Allowed access type [4]) *Let $VS_o = OVS(o)$ be a set of VS, to which object $o \in O$ belongs. Let $VS_{s|a} = SVS_a(s)$ be a set of VS, which subject $s \in S$ can access using the access type $a \in A$. The access type $a \in A$ on object $o \in O$ requested by $s \in S$ is allowed, when VS_o and $VS_{s|a}$ share a common $vs \in VS_{ALL}$.*

$$VS_{s|a} \cap VS_o \neq \emptyset \tag{3}$$

As pointed out by Lorenc, Medusa internally represents VS as bitmaps, and therefore the advantage is that any access request can be evaluated quickly [13].

Access types only define how the subject can access the object. However, the access types are used only as requirements in operations.

Definition 5 (Operation) *Let $op \in OP$ be an operation that is being performed on object o by subject s. Each operation is assigned a set $A' \subseteq A$ of required access type to perform the operation. A function $OPA(op)$ computes a set A' of required access types for operation op.*

$$OPA(op) \rightarrow A' \tag{4}$$

The operation in practice is any action that is accessing the object in some way i.e. creation of a new directory, change of a file name, read of a file name, etc. The access type represents how the object is accessed, i.e. *READ* permits to access the information about the object, *WRITE* allows to alter the object and SEE allows to see the object.

1.3.5 Decision process

The access request is created in the operating system and is first sent to layer L1. Layer L1 registers hooks and passes the access request to layer L2 to start a decision process. It is worth mentioning that decision process consists of two parts:

1. Medusa tries to make a decision based on available information about k-objects. If Medusa cannot make a final decision, the access request is passed between layers and then sent from layer L4 to the authorization server.

2. The authorization server processes the request from Medusa and returns a response, which is then returned to the kernel. The decision process is described in chapter 1.4.2.

Medusa can make a final decision if any of the following applies.

- Medusa does not have the latest information from the authorization server about the requested object and the authorization server could not be reached. The access is allowed.

- Medusa evaluated the operation in the access request. If some access type $a \in A'$ of operation *op* is not allowed according to definition 4, then the access is denied.

- Medusa evaluated the operation in the access request according to definition 4 and the access was allowed. If the authorization server could not be reached, the access is allowed.

The only scenario, where the access request is passed to the authorization server is where the access request's operation is allowed according to the definition 4. The reason

behind this, is that the authorization server has its own decision process. The authorization server can possibly alter Medusa's decision.

Medusa defines five various responses. However, in practice only 3 are applicable due to limitations of the LSM framework [12].

`MED_FORCE_ALLOW` response allows the operation regardless of other security policy that is running in the system. This is currently not applicable due to limitations of LSM framework.

`MED_ALLOW` response allows the access request.

`MED_DENY` response denies the access request.

`MED_ERR` response represents that an error have occurred during the decision process. This error can occur on layer L4 and is converted to `MED_ALLOW` response in layer L3.

`MED_FAKE_ALLOW` response should represent a faked allowed access request, meaning that the operating system receives information that the access request was allowed, but no action is performed behind the scenes. This is currently not applicable due to limitations of the LSM framework.

1.4 Constable

The authorization server Constable was developed by original authors of Medusa, Pikula and Zelem [10]. The authorization server is a process running in user space. It is responsible for authorization of access requests issued by kernel and sent through Medusa.

1.4.1 Motivation

The uniqueness of the Medusa solution is that the configuration of authorization and authorization itself is mainly carried out by a process in user space. As a result, only small authorization logic is implemented in the kernel and it is possible to implement any security model [10, 13].

1.4.2 Decision process

There are only three access types in Medusa, meaning that there exist only seven unique combinations of access types. The decision process in Constable allows one to enforce access control with finer granularity. The finer granularity is achieved by using events and handlers (described in chapter 1.3.2). Handler is a function which reacts on an event sent from the kernel and can possibly alter a decision made by Medusa.

The authorization server Constable is currently capable of 4 possible decisions. As can be seen in Table 2, the response codes are different from the Medusa equivalents. This

is due to changes in our previous research [12], in which we renamed the responses in Medusa, but left the responses in the authorization server untouched. Responses `SKIP` and `ALLOW` can be sent to Medusa, but those responses are converted to `MED_DENY`. If an authorization server sends an unsupported answer, it is considered to be suspicious.

Table 2: Constable responses in comparison with Medusa responses

Constable response	Medusa response equivalent
`SKIP`	`MED_FAKE_ALLOW`
`OK`	`MED_ALLOW`
`ALLOW`	`MED_FORCE_ALLOW`
`DENY`	`MED_DENY`

The decision process can be broken down into the following steps [13]:

1. The authorization server receives a request for authorization from Medusa.
2. If any handlers were found, the authorization server sends the information to its modules and waits for their response.
3. The authorization server tries to find registered handlers that match the subject, object, and operation from the request.
4. The authorization server collects the result from its modules.
5. The final decision is evaluated from responses of modules.
6. The authorization server sends the information about the final decision to its modules. The modules do not return any response.
7. The authorization server sends the access request decision to Medusa.

We would like to point out that, in the current implementation, step 1 is reached only if Medusa allowed the access request. If the access request was denied by Medusa, the information is not being sent to the authorization server.

1.4.3 Configuration

This chapter covers configuration of Constable. We cover only the core parts of configuration and this chapter does not serve as a complete reference sheet for Constable configuration.

Introduction Constable represents objects in the configuration in a tree hierarchy. The main idea of original authors is that the whole configuration will be constructed as one tree. The limitations of Linux system is that not everything is accessible in the filesystem hierarchy, i.e. users. The authors solved this problem by constructing their own virtual tree that does not have to reflect the real filesystem hierarchy. The virtual tree always starts with the virtual root node, which does not represent any real object in the filesystem. However, it is a node that will connect all of the subtrees in the hierarchy.

Used syntax We will use a couple of notations.

- Symbol `<arg>` represents the mandatory argument. Its value is replaced with a user-defined input.
- A command consists of a set of four types of symbols — mandatory argument, optional argument, literal, and repetition symbols.
- The symbol `[value]` is an optional argument. If the optional argument contains a separator "`|`" such as `[<arg1> | <arg2>]`, it splits the arguments into a list of choices. The optional argument then can be one of two values — either `<arg1>` or `<arg2>`.
- Literal symbol is any string in a command that is not a mandatory argument or repetition symbol, i.e. `tree`. We would like to note that even the body of an optional argument can contain literals.
- The repetition symbol $*$ means repetition 0, 1 or more times, i.e. `[<arg>]`* means that the optional argument `[<arg>]` can be repeated 0, 1 or more times.
- Repetition symbol `+` means repetition 1 or more times, it is used in a same way as $*$.
- If we want to indicate repetition of multiple symbols, we can enclose them in parentheses, i.e. `(<arg1> <arg2>)`*.

Tree definition [13] The command creates a new tree under the root node. The created tree can contain only nodes of type `<type>`. The optional parameter `clone` defines how the tree is constructed. If the value of `clone` is omitted, then each newly added node to a tree is placed as a child of the root node. If the `clone` argument is specified, then the structure of created tree will be hierarchical. The optional argument `[by <op> <expr>]` defines an operation that defines how the nodes should be inserted into the tree. The

commonly used operations are `getfile` (for trees of type file) and `getprocess` (for trees of type process).

Tree definition command

Syntax	`tree "<name>" [clone] of <type> [by <op> <expr>];`
Example	`tree "fs" clone of file by getfile getfile.filename;`

Primary tree The concept of primary trees was created just to simplify the definitions of VSs. Each node in the tree needs to be uniquely identified and we need to specify the name of the tree, to which the node belongs. If the name of the tree of type file is "fs", then we would need to prefix each object belonging to this tree with string "fs". To avoid such tedious work, we can mark a tree as primary. As a result, all objects whose name is absolute path will automatically belong to the primary tree.

Primary tree definition command

Syntax	`primary tree "<name>";`
Example	`primary tree "fs";`

Virtual space definitions [13] The VS was defined in chapter 1.3.4. We will refer to the VS belonging to a tree of type `file` as *filesystem VS*. We will refer to the VS belonging to a tree of type `process` as *process VS*. VS tagged as `primary` is used to prevent naming conflicts, which can occur if subject belongs to multiple VSs. The keyword `[recursive]` is used to mark that node at `<path>`.

Virtual space definition command

Syntax	`[primary] space <name> = [recursive] <path> [(+|-) [recursive] <path>]*;`
Example	`space all_files = recursive "fs/";`

Ability assignment Abilities of subject (defined in Definition 3) can be declared in the definition of *process VS*, but also later in the configuration separately. We present the latter option.

Ability assignment command

Syntax	`<subject> (<accessType> <object> [, <object>]*)+;`
Example	`all_domains READ all_files, all_domains;`

Events Constable can react to events that are registered. Parameter `<subject>` and `<object>` can be identified with the name of VS or we can use the $*$ symbol to represent "any VS" condition. The object is not always required argument, since not all operations require an object. As an example we can provide `fork`, which does not require any object since it is duplicating the process. The optional parameter `[handler_body]` defines the code that should be executed when the handler is triggered. The parameter `<flag>` has default value `VS_ALLOW`, but can be substituted for any value from the following list [13]:

`VS_ALLOW` flag is present if the event is informing about the issued access request which was allowed by Medusa based on VS.

`VS_DENY` flag is present if the event is informing about the issued access request which was denied by Medusa based on VS.

`NOTIFY_ALLOW` flag is present if the event is informing the modules that the issued access request was allowed.

`NOTIFY_DENY` flag is present if the event is informing the modules that the issued access request was denied.

Event handler definition

Syntax	`<subject> <event>[:<flags>] [<object>] { [handler_body] }`
Example	`* getprocess { }`

Functions Functions can be used as helper-subroutines to encapsulate repetitive procedures. There are a few special functions, such as `_init` that have a special meaning.

Function definition

Syntax	`function <name> { [function_body] }`
Example	`function _init { }`

2 Introduction to Policy Mining

In current articles, the policy mining can be described in various ways. Xu and Stoller ([14]) describe its use cases to mine access control for large organizations because manually, it would be tedious work. Cotrini et al. do not provide a definition for policy mining but provide very similar description that systems become very complex and it is difficult to maintain policies manually. In 2018, there was increased interest in Attribute-based access control, also known as ABAC. According to Cotrini et al., the use case of an algorithm should be to effectively transform policy from one security model to ABAC (e.g. RBAC security model to ABAC security model). Stoller et al. have not defined the policy mining, but defined the policy mining problem for ReBAC [14, 15].

2.1 Terminology definition

From the above use cases, we can try to provide a formal definition of policy mining. First of all, *policy mining* is top-level abstraction of what Xu and Stoller and Cotrini et al. were describing [14, 15]. We adapted the definition of policy mining problem defined for ReBAC to MAC policy mining problem. [16]

Definition 6 (Equal policies) *Let* Π *an* Π' *be security policies for some system. We say that* Π *and* $\Pi^{'}$ *are equal policies if every access request allowed or denied according to* Π *would be allowed or denied also by* $\Pi^{'}$*.*

Definition 7 (Policy mining problem) *Let* $\Pi = \{S_C, O_C, A_C, R\}$ *be a MAC security policy. The policy mining problem is to create a security policy* Π *with maximum policy quality metric* Q_{pol} *when given context information* C *about policy* Π_0 *as input such that the security policies are equal.*

To elaborate on the Definition 7, we would like to explain each part, from which the security policy Π consist of. We provided only generic definition and the details about each set are left for concrete instance of a problem.

Subject context S_C The subject context contains the security context that is assignable only to the subjects.

Object context O_C The object context contains the security context assignable only to the objects.

Action context A_C The action context contains the security context assignable only to the actions.

Rules R defines the relationship between S_C, O_C and A_C. A single rule is represented with triplet $[S'_C, O'_C, A'_C]$ where $S'_C \subseteq S_C, O'_C \subseteq O_C, A'_C \subseteq A_C$. An access request is allowed by a rule if subject s with subject context S'_C is requesting action a having action context A'_C on object o having object context O'_C.

Subject context, object context and action context are authorization elements which are used in process of making a decision by security policy Π.

The definition of policy mining problem includes some context information C, which hasn't been described yet. In practice, it could be a log file collected from the system where the policy was enforced or implemented, it could be currently running security policy itself, or combination of both. Informally, the log file can contain issued access requests and policy-enforced decisions. The log entry can have following structure:

Subject information. Information about a subject within the system. The information does not have to be just the identifier, but also the security context.

Object information. Information may depend on what the object is representing. In case the object is a file in Linux filesystem, the information can be path to the file, owner of the file, mode, DAC permissions, etc.

Action. The action that was being executed on an object by a subject.

Decision. The decision made by the access control mechanism.

Metadata information. The information about a log entry, e.g. timestamp, entry id etc.

The Q_{pol} policy quality metric aims to optimize a selected property of the security policy, for which the metric was selected. We provided a list of commonly used quality metrics in the chapter 2.5.

The primary goal of security policy is to enforce behavior in the system.

Definition 8 (Authorization tuple) *An authorization tuple is denoted as* $t_\Pi = [\beta, d]$ *such that* β *is the access request and* $d \in \{ALLOW, DENY\}$ *is a response of security policy* Π *on access request* β*. We denote set of authorization tuples where* $d = ALLOW$ *with* T^+_Π*. Similarly, the set of authorization tuples with* $d = DENY$ *is denoted as* T^-_Π*. We denote the set of all possible authorization tuples as* $T_\Pi = T^+_\Pi \cup T^-_\Pi$*.*

Definition 9 (Authorization decision function) *The authorization decision function* $\Gamma_\Pi(\beta) = d$ *is a function which maps access request* β *to a decision* d *when given context of security policy* Π*.*

The *authorization decision function* is a formal way of representing that, when given an access request β and security policy Π, an access control mechanism, which enforces a decision according to the security policy makes a decision d.

Definition 10 (Allowed access request) *Let β be an access request. The access request is allowed by some rule $r \in R$ from security policy Π if all of the following conditions are met:*

- *The subject s in β has required subject context specified in rule r.*
- *The subject is accessing object $o \in \beta$ having object context specified in r.*
- *The subject s is requesting action a which has action context defined in rule r.*

The definition 10 implicitly infers that our defined metamodel for MAC security policy allows only whitelisting. The limitation of our definition of rules is that we cannot specify rule which forbids the action.

Definition 11 (Allowed authorization tuples) *Let T_Π be a set of authorization tuples of a security policy Π. Let's select some rule $r \in R$ from a security policy Π. We denote a set of allowed authorization tuples by rule r as $[[r]]_{T_\Pi}$. The security policy Π is implicit argument to the notation $[[r]]_{T_\Pi}$.*

2.2 Desired properties of security policy

A primary goal of policy mining is, by definition, to produce a security policy. However, when we design an algorithm, we want to strive for certain properties if we want to fulfill the primary goal. The next list contains a few properties that the security policy can strive for:

Generalization The policy should be able to generalize and should generalize well.

Completeness The policy should be able to provide a decision for any issued access request [17].

Correctness The policy should make the correct decision for any access request [18].

Improvement The policy should supersede the original policy in the target metrics. As an example of improvement, the new policy can be less complex, more fine-grained, etc. An example where improvement was achieved is by Li et al., the authors of the ASPGen framework [18].

Conciseness Sometimes, the intended use of policy mining is to migrate one security policy to a new security model, while keeping it editable by system administrators afterwards.

The generalization property is essential for any security policy, because if the security policy does not meet this property, decisions made will be very likely too restrictive, which means that it will allow only access requests which appear in the input context information. Such a policy has very limited usage, since most of the systems are dynamic, and as a consequence, they change very often.

The completeness property goes hand in hand with the correctness. If a policy misses completeness, that means that there is a part of the system which is not covered by the policy. That can lead in some cases to undesired behavior because then uncovered decisions are purely implementation-dependent. This is especially a problem if the system uses blacklisting, i.e. what is not forbidden is allowed.

The correctness is another essential property. The definition of correct in this context is that the new policy should make the same decision as the input policy would. The original policy can have its flaws, where it makes an incorrect decision according to the organization's intent due to bad implementation. However, if the new policy still has these flaws, it is still considered to be correct.

The improvement property is considered to be more desirable than required. The algorithm cannot have this property if it does not have some of the other properties in the list.

Conciseness's definition sounds a little bit vague. To elaborate more on what it actually means, administrators should be able to edit it. If the policy mining algorithm mines the policy for the ABAC security model, then our goal is to have the rules as fine-grained as possible. [15] If the algorithm mines policy for RBAC security model, we would prefer to have a few roles, which are as least overpermissive as possible, so it is easily manageable by the system administrators.

2.3 Usages of policy mining

Each algorithm is designed with some goal in mind. The list below contains a few usages of policy mining collected from already implemented algorithms.

Simplification of existing policy The policy mining algorithm can receive logs generated from existing policies as Xu and Stoller [14] did. It can also generate a new policy which can cover everything that is included in the logs, while the policy can

be much smaller and more effective. Or we may want a new policy which covers only certain fraction from the existing policy.

Generation of new policies An example of this use case is the policy mining algorithm by Molloy et al., who tried different approach from Xu and Stoller and generated a new RBAC security policy from real logs. [19]

Migration of security policy to a different model The logs are just data about user permissions, but those data describe the behaviour the policy enforces. As a result, it is possible to simply work with the logs and some context, and get an equal policy implemented by another security model. [15]

The properties which the policy should have depends on what our goal is. If we are designing the algorithm for simplification of existing policy, it means that we have some existing policy, and we strive for improvement property. If we are generating a new policy, then it may be desirable to generate it only for some part of the system and then we don't really need completeness for the whole system, but rather for the part of the system that we are interested in. If we are migrating a security policy to a new security model, then it may be desirable to have generalization, completeness, correctness and possibly conciseness.

There are multiple algorithms already implemented to mine ABAC policies from logs, for which we can provide examples of selection of their goals. Xu and Stoller strived primarily for generalization, correctness and completeness. The authors came up with a deterministic algorithm and were among the first researchers who tried to mine ABAC policy from logs. [14] Jabal et al.'s algorithm Polisma focused mainly on completeness and correctness, which the authors tried to achieve with combination of Apriori algorithm [20] and machine learning. [17] Cotrini et al.'s Rhapsody tried to improve the Apriori algorithm and have had conciseness as the primary goal together with other essential properties, namely completeness, generalization and correctness. [15] Joshi et al. used the k-means machine learning algorithm to mine the ABAC policies from logs and strived for generalization, conciseness, correctness and completeness. According to their results, their algorithm could achieve improvement in the existing policies.

2.4 Prerequisites for policy mining

The context information is the main input to the algorithm and it is worth exploring different types of context information that we can work with. The quality of mined policy

is not dependent just on the algorithm but also on the quality of context information. The algorithm can achieve different goals and results with various input.

2.4.1 Log files

A very common context information used in policy mining algorithms are log files. We can observe multiple types of log files used throughout research of policy mining. The suitable type of logs depends on the situation.

Real logs The most complicated logs are real logs for a couple of reasons. Real logs are usually incomplete. As a consequence, it is difficult to achieve the completeness of a policy because not all access decisions could be triggered and, therefore, some rules in the log could be missing. Next, the logs can contain a lot of noise, especially if they span a longer period of time. The noise are overassignments or underassignments created due to e.g. improper configuration. For example, the log can contain information that one subject could try to access a resource several times while in some cases the access was allowed and in others was denied. This could happen for variety of reasons like bug in the implementation, changes in the policy requirements, etc. The context behind the change of decisions is usually difficult to determine using the algorithm alone. The real logs are in a lot of cases sparse, because in real life the users usually trigger only certain rules frequently, while some rules may not even appear in the logs. The real logs can be used in situations when there is insufficient information about the system.

Synthetic logs Synthetic logs are crafted logs. The advantage of using synthetic logs is that there is more control over the data that are included in the logs. If the log is custom-crafted, it is possible to eliminate the problems with conflicts and missing rules which the real logs have. The problem with synthetic logs is that they are less authentic. Often, we can generate synthetic logs by triggering each rule in the security policy by automation tools. [14] [18]

Logs are one of the elements that come as an input to the policy mining algorithm. Molloy et al. have shown that it is possible to generate new policies from the real logs, while they have certainly had satisfiable success. [19] Joshi et al. used real sparse logs for their machine learning algorithm for mining ABAC policies. In their experiments, Xu and Stoller used synthetic logs generated from policies created in previous research. An interesting example of crafting of the synthetic logs was carried out by the authors of the ASPGen framework, who intentionally selected open-source projects as their system under test. As a consequence, they could use white-box testing tools to create and execute test

cases and trigger a lot of different scenarios of the application, including edge cases. The size of the logs may vary and depends on how large the existing policy is or how large the new policy should be. Smaller logs will very likely generate smaller policies because the smaller number of entries in the logs may implicate fewer rules in the new policy. [14] [18]

2.4.2 Security policy context

Usually the algorithm does not explicitly require, but it is nice to have some additional context information about the policy being mined. Algorithms for mining ABAC policies usually require context information about the attributes of users and resources. If the source policy is implemented as an RBAC model, the context information could be roles of the users. If the source security policy is implemented as a MAC model, the context information can be information which could be hardly inferred by the algorithm. While in some algorithms ([14, 21, 15]) the context information was used only to create and assign new attributes in the new policy, an interesting approach was introduced by authors of the Polisma framework. The authors actually used just the context alone to generate some rules that did not appear in the log. [17]

2.5 Policy evaluation

Once the algorithm is implemented and the policy is mined, there are some required techniques, which help the developers evaluate the mined policy. A very obvious way how to do it is to review the mined policy manually. However, in practice, it is very impractical considering the fact that the policies can be very large and during development of an algorithm, there will be a lot of generated policies not worth consideration. Another difficult problem is to review the policy correctly. As an example, it is difficult to proclaim that the new policy is not overpermissive when reviewed manually because humans are prone to errors and we may not realize it just by looking at rules of a policy alone. As a solution to this problem, developers of policy mining algorithms developed a variety of different quality metrics that are being used to evaluate the security policy. These quality metrics help to compare different policies almost instantly, and then pick the policy with the highest quality according to selected metrics.

2.5.1 Weighted structural complexity

According to a study by Beckerle and Martucci, the access control rule sets are easier to read and manage when they are concise. [22] A *weighted structural complexity* (or acronym WSC) is a quality metric which is considered to be of higher quality when the final policy has as few elements as possible. It is important to keep in mind that the policy needs to be equal with input policy. If we omit the latter condition, the policy with the best WSC

is empty policy which does not contain any rules because it is as concise as possible.

The metric was originally proposed for mining policies for RBAC model [23]. However, it was then adapted also by researchers of other policy mining algorithms for ABAC. [14, 21]

To our current knowledge, there is no definition of *WSC* for MAC. Therefore we present our adaption of *WSC* definition for MAC in equation 5. Let $W = \{\forall w_i \in R^+, w_i \geq 0 \wedge w_i \leq 1, \forall i \in N^+\}$ be a set of weights. The set of weights W in the definition is defined manually, and it represents how much value is given to a specific set.

$$\begin{aligned} WSC(\Pi) &= WSC(S_C) + WSC(O_C) + WSC(A_C) + WSC(R) = \\ &= w_0 \sum_{i=0}^{m-1} ||s_i|| + w_1 \sum_{i=0}^{n-1} ||o_i|| + w_2 \sum_{i=0}^{p-1} ||a_i|| + w_3 \sum_{i=0}^{q-1} ||r_i|| \end{aligned} \tag{5}$$

Let's represent each item in sets S_C, O_C, A_C and R as items with a set of attributes i.e. name, role, type, etc. Each item then can be described as a set of values combined together as conjunctions. The syntax $||x||$ represents cardinality of x, and in the equation 5, it is the number of required values to describe an item.

As an example let's define the item, which is described with conjunction $selinux\_role = admin \wedge selinux\_type = mysqld$. The *WSC* of this item is 2 because it has 2 values.

Variables m, n, p and q are number of elements in the corresponding sets. We used in equation 5 a different symbol as upper limit for each infinite series because the number of elements in each set may vary.

2.5.2 F1-score

F1-score is statistical method. In a context of policy mining, it can be used to measure accuracy of a policy. The F1-score is primarily used as machine-learning metric but it can also be used as a statistical method to compare performance of different policies. F1-score F consists of two parameters — precision (P) and recall (R).

$$F = \frac{2PR}{P+R} = \frac{2}{\frac{1}{P} + \frac{1}{R}} \tag{6}$$

For the purposes of the following definitions, let's assume that we have a log file $L \subseteq T_{\Pi_0}$ representing partial set of authorization tuples collected for some security policy Π_0.

Definition 12 (True positive rate) *True positive rate denoted as $TP_{\Pi|L}$ is the portion of access requests that are allowed by security policy Π and have been allowed according to log file L_Π.*

$$TP_{\Pi|L} = \frac{|\{[\beta, d] \in L^+ \,|\, \Gamma_\Pi(\beta) = ALLOW\}|}{||L^+||} \tag{7}$$

Definition 13 (False positive rate) *False positive rate denoted as $FP_{\Pi|L}$ is the portion of access requests that are allowed by security policy Π but have been denied according to log file L.*

$$FP_{\Pi|L} = \frac{||\{[\beta, d] \in L^- \,|\, \Gamma_\Pi(\beta) = ALLOW\}||}{||L^-||} \tag{8}$$

Definition 14 (True negative rate) *True negative rate True negative rate denoted as $TN_{\Pi|L}$ is the portion of access requests that are denied by security policy Π and have been denied according to log file L.*

$$TN_{\Pi|L} = \frac{||\{[\beta, d] \in L^- \,|\, \Gamma_\Pi(\beta) = DENY\}||}{||L^-||} \tag{9}$$

Definition 15 (False negative rate) *False negative rate False negative rate denoted as $FN_{\Pi|L}$ is the portion of access requests that are denied by security policy Π but have been allowed according to log file L.*

$$FN_{\Pi|L} = \frac{||\{[\beta, d] \in L^+ \,|\, \Gamma_\Pi(\beta) = DENY\}||}{||L^+||} \tag{10}$$

Precision and recall are then defined as follows [24].

$$P = \frac{TP_{\Pi|L}}{TP_{\Pi|L} + FP_{\Pi|L}} \tag{11}$$

$$R = \frac{TP_{\Pi|L}}{TP_{\Pi|L} + FN_{\Pi|L}} \tag{12}$$

The challenge in using F1-score is to find a balance between precision and recall. Van Rijsbergen proposed the effectiveness of combining precision and recall in the following formula. Parameter α is selected manually and is used to choose whether to prioritize recall or precision [24].

$$E = 1 - \frac{1}{\frac{\alpha}{P} - \frac{1-\alpha}{R}} \tag{13}$$

2.5.3 Confidence

Confidence focuses on similarity of requests. The usage of *Confidence* is different from usage of *WSC* and *F1-score*. The confidence is used for selection of appropriate rules to the policy, rather than for comparison of policies. [20] Cotrini et al. provided a definition of confidence for ABAC security model in their work about the mining algorithm Rhapsody. The authors also developed new metric called *Reliability*, which is based on Confidence. Reliability should eliminate the overpermissiveness in the mined policy. Overly permissive security policy allows access requests, about which the security policy did not have enough evidence in the log file $L \subseteq T_{\Pi_0}$, where Π_0 is some security policy, for which the authorization tuples were collected [15]. There are also several other works, which created a new metric based on the Confidence.

$$Conf(r) = \frac{|[[r]]_{L^+}|}{|[[r]]_L|} \tag{14}$$

3 Medusa problem description

The goal of this chapter is to present the policy mining problem for security module Medusa.

3.1 Security policy

We would like to start by defining the structure of the security policy in the context of Medusa. Since we want to refer to the concrete instance of the MAC security policy described in chapter 2.1, we define the security policy $\pi = [S_C, O_C, OP_C, R]$.

3.1.1 Subject context

The subject context are subject authorization elements used in the process of responding to an access request. The subject context in Medusa is represented with the process VS. Formally, subjects are assigned to a process VSs. However, this contradicts with the terminology that we specified in chapter 2.1, because we stated that the subject context is assigned to the subject. However, we can look at the definition from reverse perspective and view it as that we are binding the subject to a one or more VSs. Each instance in subject context have the following attributes.

$$process\_vs = [name, execs]$$

Name is the name of a process VS and it can contain only single value. *Execs* is a set of values that represents paths to binary executables used by the process.

3.1.2 Object context

The object context are object authorization elements used in the process of responding to an access request. The object context in Medusa is represented with filesystem VS. We are encountering the same problem with our definition as with subjects 3.1.1, but we can apply the same principle. The filesystem VS consists of the following attributes.

$$filesystem\_vs = [name, path\_entries]$$

`Name` is the name of filesystem VS. We haven't defined the path entry yet. `path_entries` is a set of path entries. The path entry can be defined as follows.

$$path\_entry = [absolute\_path, is\_recursive, is\_addition]$$

`absolute_path` represents the path to the file in absolute format. `is_recursive` is a boolean flag representing whether the entry is recursive. `is_addition` is a boolean flag

representing whether the path entry is addition or exclusion to the VS.

3.1.3 Operation context

Operation in the Medusa context is synonym for action. We want to avoid conflicts with the set of access types A. Therefore, from this point on, we will refer to the action as operation and to the action context A_C as operation context OP_C. The operation is defined as a tuple of `name` and its required access types.

$$operation = [name, acc\_types]$$

name is the name of operation. *acc_types* are required access types that need to be allowed according to definition 4.

3.1.4 Rules

Rules define the allowed operations between subject and object. In the context of Medusa, we can define it as follows. A rule defines the required security context from the subject context, the object context and the required access types.

$$rule = [S'_C, O'_C, A'] \tag{15}$$

The example rule in equation 15 allows each access request, in which all of the following conditions are met.

- The subject belongs to any of the process VSs from S'_C .
- The subject is requesting an operation whose $acc\_types \subseteq A'$ meaning that the operation requires the access types defined in the rule or possibly its subset.
- The requested object by subject belongs to any filesystem VSs from O'_C.

3.2 Problem definition

Definition 16 (Medusa policy mining problem) *Let $\pi = \{S_C, O_C, OP_C, R\}$ be a MAC security policy implemented using the Medusa security module. The policy mining problem is to create a security policy π with the maximum policy quality metric $WSC(\pi)$ when given partial authorization tuples L as the input.*

We aligned the problem definition with Definition 7. Partial authorization tuples L are context information about the initial policy π_0. Our goal is to create a security policy for the application running in the user space, which is not enforced by any security policy.

The last statement seems to contradict with Definition 7 because if no policy was loaded, then the set of authorization tuples L does not describe any security policy. However, it is worth to point out that if no security policy is enforced, then all access requests are allowed. This behavior can be described with a non-restrictive security policy and that will fit into the Definition 7.

3.2.1 Choice of policy quality metric

We chose the $WSC(\pi)$ as our quality metric because our goal is to create the security policy, which will have as few rules as possible and will be able to generalize. *WSC* seems to be the only applicable quality metric from the policy quality metrics listed in chapter 2.5.

F1-score is not applicable in our case. As we mentioned in chapter 3.2, the mining is performed on authorization tuples for non-restrictive policy. Therefore, $L = L^+$, meaning that it will not contain any access requests that should be denied. There is no reason to aim for accuracy as our algorithm will work only with allowed entries.

We could use *Confidence* as our metric for optimization of rules, since it aims for similarity of requests. However, we chose *WSC* as it can be also used to compare security policies.

3.2.2 WSC

The *WSC* definition from chapter 2.5.1 had to be adapted for security policy implemented with the Medusa security module.

$$\begin{aligned} WSC(\pi) &= WSC(S_C) + WSC(O_C) + WSC(OP_C) + WSC(R) = \\ &= w_0 \sum_{i=0}^{m-1} (1 + ||s_i.execs||) + w_1 \sum_{i=0}^{n-1} (1 + ||o_i.path\_entries||) + \\ &+ w_2 \sum_{i=0}^{p-1} 1 + w_3 \sum_{i=0}^{q-1} (||r_i.S'_C|| + ||r_i.O'_C|| + 1) \end{aligned} \tag{16}$$

The expression $WSC(S_C)$ consists of two parts. We add +1 as penalization just for the pure existence of each process VS in S_C. Then we add to the complexity the size of the *execs* set. As a result, security policies, which have smaller quantity of VSs and larger *execs* set will be considered smaller. Let's consider an example, in which the security policy contains two process VSs with the same *UID*, but different *execs* sets. Since these two VSs represent the same process, they can be merged together. If we compare this security policy to the security policy that have these two VSs merged, we will get smaller *WSC* by 1.

The expression $WSC(O_C)$ follows a similar pattern as that of $WSC(S_C)$. We consider a security policy to be smaller if we have fewer instances of filesystem VSs, which contain more path entries.

The expression $WSC(OP_C)$ was simplified to just to a sum of ones. The reason behind this is that the operations do not contain any attribute that would contribute to the complexity. More required access types are just a requirement for the operation and do not contribute to the complexity of the security policy.

The expression $WSC(R)$ is expanded to the sum of cardinalities of individual sets. The only exceptions are access types A', where we always add +1. The reason why we do not want to compute cardinality for set A' in a rule is because more access types in A' do not add to the complexity of the rule. The access types are just the relationship between the S'_C and O'_C. We do not want to introduce the +1 penalty for the rules, as we did with other sets. A fewer rules lead to more complicated rules.

4 Implementation

This chapter presents the process of implementation of a policy mining algorithm. The algorithm and all of its related modules are implemented using functional programming language F#.

The solution is split into different modules. We will provide brief overview of each module in Figure 3 and then in the following chapters we will provide more detailed description.

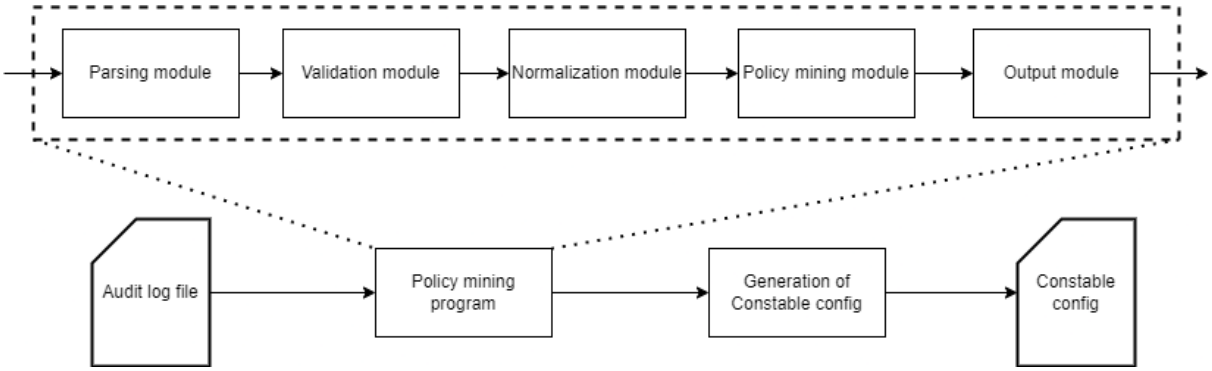

Figure 3: Solution structure diagram

Parsing module. The module is responsible for processing the input entries in the log file and parse them into fields.

Validation module. The module is responsible for validation of entry as a whole, meaning whether all required fields are available.

Normalization module. The module normalizes the value of fields and groups them into required form for the policy mining algorithm.

Policy mining module. The module is processing the normalized entries and outputs the mined rules.

Output module. The module processes the mined rules and transforms it into presentable form.

Generation of Constable configuration. Program which takes mined rules and outputs valid Constable configuration file.

Before we will present detailed description of each part, we will describe the process of obtaining the input log file.

4.1 Goals of implementation

In order to fulfill the goals of thesis, we split the objectives into individual parts. All parts of the following objectives are covered in the implementation chapter. However, we won't cover the implementation details. The implementation itself can be found in our Github repository[1].

- Obtain input data used for policy mining.
- Design an algorithm for policy mining of policy for Medusa.
- Implement the algorithm and mine a security policy.
- Generate Constable configuration from the output of the algorithm.
- Evaluate results.

4.2 Log obtainment

The first step towards policy mining was to get some data to begin with. We needed data to run the algorithm, but also to know what information we have available for the algorithm. We chose to monitor mysql-server MariaDB because it is available for any Linux distribution and we expected a lot of tools that could be helpful along the way.

4.2.1 Testing environment setup

For testing environment, we prepared our own server, where the testing environment was running. The relevant configuration information of server with installed programs and their versions is captured in Table 3.

Table 3: Configuration of testing environment

Property name	Value
Operating system	Fedora 35 Server
Architecture	x86_64
Kernel version	5.16
Running security module	SELinux
SELinux mode	Permissive[2]
Mysql server	MariaDB

[1] https://github.com/Nestastnikos/MiningMedusaPolicy

[2] SELinux in permissive mode allows all access requests, but creates log entry for each access request that should be denied according to loaded security policy.

MariaDB version	10.5.13
Audit system	Auditd

4.2.2 Audit setup

In order to collect some data for our policy mining algorithm, we experimented with different ways of audit setup.

Monitoring certain paths. Our first attempt was to try to monitor certain directories that we thought MariaDB could be using. We setup the audit rules for following files.

- /var/log/mariadb (and its children) — directory for log files
- /var/lib/mysql (and its children) — directory for mysqld data
- /usr/sbin/mysqld — executed binary by client

This method is more suitable when we want to track changes in the directory, but in our case we need to track the actions of a mysql process. The main issue with this approach is that it requires enumeration of paths that are being used. In real example. this can lead to not auditing all actions of a process, because it can use files or directories that were not taken into consideration and therefore are not tracked by the audit system. Even in our case, we enumerated only the most obvious paths, but we did not cover all the directories that the mysql process is using. However, this method can be still beneficial if we need to collect only partial information about the process's activities to obtain some test data, on which we want to test our algorithm.

Monitoring by security context. If the system is monitored by path security module like AppArmor, the security context is defined mostly with paths. If the configuration file is available, it is possible to setup an audit rule for each rule in the configuration. However, in our case, our testing environment machine had installed SELinux security module, which is inode-based. The advantage here is that auditd has builtin support for monitoring actions by SELinux context. In order to collect all data by mysql process, we monitored actions in the whole filesystem by process with SELinux label *mysqld_t*, which is the type assigned to mysql process.

Monitoring by process executable file. When monitoring process actions, in most cases it should be sufficient to monitor all actions within system and filter it by executable binary name (as absolute path), from which the process was being started.

This can be a workaround if no security module is installed on the system. The rules created on one system may not be compatible on other system if the binary has different name, but it is nice workaround if no security module is installed in the system and there is no security context which can serve as filterable property (as in the method above). In our case, it yielded the same results as the previous method.

We used logs collected by all three methods. The logs from monitoring of paths were used just for testing purposes. The logs from the other two methods could be used for policy mining, since they contain all actions made by the mysql process.

4.2.3 Triggering control flow paths of a mysql process

Another step towards policy mining is to collect a log file with audited actions of a process. The challenge with this step is that usually the daemon is doing a lot of repetitive regular tasks, and there are control flow paths that are triggered only in rare cases. Luckily, mysql server is a large program with a lot of users and testing programs. We tried three approaches to trigger different scenarios.

Mysql test suite.[3] We were inspired by the approach of authors of ASPGen (which we already discussed in chapter 2.4.1) and searched for tools that could be used for triggering different scenarios (even edge cases). We found the *mysql test suite* and ran all of the tests in the suite. After the tests finished, we realized that the suite is running inside it's own testing environment, by a process which is not labeled with SELinux context and therefore we did not obtain any audit entries.

Mysqlslap.[4] The program is a diagnostic tool used to simulate load on a mysql server. From the point of view of the server, it simulates scenario where multiple clients are trying to perform queries at once. The advantage of this approach is that it is not running in its own environment but it requires the running program. This means that all actions by the server appear in the audit system. We found multiple similar client programs[5] but these programs were accessing the same files as *mysqlslap* and we didn't get any new entries. We simplified the process of collection and relied only on *mysqlslap*.

Manual execution. We tried to run *mysql*[6] program and trigger some paths manually. However all the audit entries that appeared in the audit system were already equivalent to the ones obtained with mysqlslap program.

[5]https://dev.mysql.com/doc/refman/5.7/en/programs-client.html
[6]https://dev.mysql.com/doc/refman/5.7/en/mysql.html

4.3 Parsing module

The first interaction of a log file with the project is in the parsing module. The parsing module is responsible for processing the log file as a whole.

The outline structure of a single entry is shown in Listing 1. As shown in the example, each entry has fields split into 4 categories.

Proctitle represents an executed command. In our case it is the full command that triggered the creation of audit entry.

Path is input to a system call. If the system call requires multiple inputs, multiple paths are specified for one entry.

Cwd is the current working directory.

Syscall represents information about the executed system call.

It is not necessary to go through all the fields that the entry consists of because the list is very long and we did not use all the information that is available. We parsed all fields from an entry using a regular expression and performed a selection of only needed fields.

```
----
type=PROCTITLE msg=audit(03/10/22 18:48:20.282:283) : proctitle=...
type=PATH msg=audit(03/10/22 18:48:20.282:283) : item=0 name=...
type=CWD msg=audit(03/10/22 18:48:20.282:283) : cwd=/
type=SYSCALL msg=audit(03/10/22 18:48:20.282:283) : arch=x86_64 ...
----
```

Listing 1: Example entry in a log file

However, we highlighted fields that we used in Table 4. We described how the information was used in the algorithm in chapter 4.5.2.

4.4 Validation & normalization module

This chapter presents validation and normalization module together. Validation module is responsible for validation of entries. Validation and normalization are important because we need to ensure that all entries have corresponding data required for the algorithm and that the algorithm can rely that these data are in a certain format. We applied the following group of validations for each entry:

- Each field in Table 4 is present in each entry.

Table 4: Overview of selected fields

Category	Field name	Field description
PROCTITLE	proctitle	Executed command.
PROCTITLE	msg	Timestamp and unique identifier of an entry.
PATH	name	Value of item (it can be either relative, canonical or absolute path).
PATH	nametype	Meaning of an item, it defines context in which the name argument is used.
PATH	mode	Directory or file DAC permissions. If the mode is directory, it may contain information whether the directory has sticky bit.
PATH	uid	Identifier of a user who executed the command.
CWD	cwd	Current working directory as absolute path.
SYSCALL	syscall	Name of executed syscall.

- Each field consists of expected set of characters (in most cases we consider alphanumeric characters).
- Each field has expected format and contains value from predefined set of known values.
- If any of these validations fails, we produce a corresponding error message and the whole pipeline is terminated.

Validation step is required mainly because even though the input log file is under our control, the creation of log file itself is carried out by external system (namely *auditd*). It is possible to view the format of fields in the log file manually, but in our case the log file has a few thousand entries and it is very tedious work to ensure the correctness manually.

Normalization step unifies the data into a single format. Paths are very good example, where the *name* field in *PATH* category can have variety of possible formats — relative path, canonical path or absolute path ending with or without "/". The path in the item has format depending on how it was passed by the program to the system call. To be able to unify the path to absolute format, we have selected the *cwd* field from each entry. If the path is in relative or canonical form, we can transform it to the absolute form using the *cwd* field.

4.5 Policy mining module

The Policy mining module contains procedures used for policy mining. The input to the module are normalized log entries L_N and syscall context C. A syscall context C is enumerated set of tuples representing mappings of syscall to access types required by Medusa. To obtain a syscall context C, we went through implemented access types in security module Medusa and looked manually at currently required VSs for each access type, then we manually wrote JSON file with the desired configuration.

$$C = \{[syscall\_name, r|w|s]^*\} \tag{17}$$

The structure of presented algorithm is inspired by algorithm of Xu and Stoller in their work about ABAC policy mining [14]. Since we used a functional approach, we tried to avoid loops as much as possible. This allowed us to break the algorithm into separate pieces (functions) and combine it with function composition[7] into final result. The outline of algorithm steps is visualized in Figure 4.

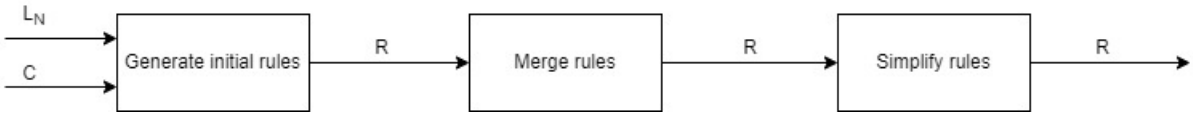

Figure 4: Outline of algorithm

Now we can formally show the outline pseudocode of the algorithm. As can be seen, there are no loops. We will go through the details of smaller functions in the following chapters.

Algorithm 1 An outline of the algorithm

Require: L_N, C

Ensure: $L_N.syscall \subseteq C.syscall$

$R_{initial} \leftarrow generate\_init\_rules(L_N, C)$

$R_{merged} \leftarrow merge\_rules(R_{initial})$

$R_{simpl} \leftarrow simplify\_rules(R_{merged})$

return R_{simpl}

All the inner functions are implemented using recursion, this is the reason why we do not append the snippets together and do not present them as one large algorithm. We mentioned earlier that we wanted to avoid loops because we used the functional language

[7] https://en.wikipedia.org/wiki/Function_composition

F#. At first glance, recursion algorithms do not seem to be preferable, because the number of entries can be relatively large and in the real world we are limited by the resources. Therefore recursion can lead very easily to stack overflow due to too many entries loaded in the memory.

However, we would like to point out that all our recursive functions are implemented as tail recursion. F# has support for tail-call optimization[8], which should eliminate the problem with stack overflow, if implemented correctly.

The function *generate_init_rules*() is described in chapter 4.5.4. The function *merge_rules*() is described in chapter 4.5.5. The function *simplify_rules*() is described in chapter 4.5.6.

4.5.1 Assumptions for the policy mining algorithm

To simplify the problem that we are trying to solve, we made a following assumptions about the audited system.

- The system is a Unix-like system having a root directory "/".
- The system is not enforcing any security policy with any security module. We stated in the chapter 4.2.1 that the system is running SELinux, but we used it in permissive mode only for better collection of log files, to enable possibility to search by SELinux context. As a consequence, everything appearing in the log file should be allowed.
- The audited program is not malicious, therefore we assume that access requests that were denied due to failure are valid access requests that should be allowed in the policy. The reason behind is that programs are usually designed for multiple Linux distributions. The filesystem structure for Linux distribution can vary and it is only natural assumption that the program tries to find a file (e.g. a configuration file), whose locations can vary depending on the Linux distribution we are running or the config can be stored on different locations (e.g. there can be a configuration related to a user, group or to the whole system). This assumption helps to create a policy which can work on multiple Linux distributions.
- The audited program can contain defects, but that defect behavior is treated by us as regular traffic and we do not introduce any special mechanisms to treat the defects as irregular behavior.

[8]https://en.wikipedia.org/wiki/Tail_call

4.5.2 Used data structures

Before we can jump into the individual steps of the algorithm, we would like to present data structures used in the algorithm. We provide a quick reference of structures in Table 5.

Table 5: Used data structures

Structure name	Properties
Item	$[path, nametype, mode, isSticky]$
AuditEntry	$[id, uid, proctitle, syscall, items]$
AbsolutePath	$[path, depth]$
PathEntry	$[path, isDirectory, isAddition, isSticky, isRecursive]$
FsVirtualSpace	$[name, pathEntries]$
ProcessVirtualSpace	$[uid, proctitle]$
Rule	$[processVirtuaSpace, fsVirtualSpace, accTypes]$

Item is representing an input argument with metadata to the syscall. It is combined from name, nametype and mode fields in the entry. Mode contains information whether the entry is a file or a directory and whether it has a sticky bit. In our case, we are only interested in sticky bit on directories, therefore in case the path of an item points towards a file, the *isSticky* flag is automatically set to *false*.

AuditEntry represents a single entry in a log file. Field *items* contains instances of *Item* data structure. Other fields remains the same as displayed in Table 4.

AbsolutePath is representing the path of a file or a directory. For purposes of the algorithm, it is required to store one metadata information called "depth". The depth is the distance from the root directory. To provide some examples, path "/" has depth 0, path "/var" has depth 1, path "/var/lib" has depth 2 etc. We will refer to the ancestor directories of a path in the hierarchy as parent paths.

PathEntry is representing one path entry in the filesystem VS. Field *path* is the absolute path. Flag *isDirectory* is used for distinguishing whether the pointed target on the path is pointing to a directory or a file. Flag *isAddition* represents whether this entry adds new path which can be accessed or adds an exception, meaning that this path is forbidden. Flag *isSticky* is applicable only for directories and represents

whether the directory has sticky bit set. Flag *isRecursive* is again applicable only for directories and represents whether only given path is included or also its children. If path entry contains parent path of another path entry's path we will refer to this entry as parent path entry.

FsVirtualSpace represents filesystem VS (object). It consists of its name and list of path entries.

ProcessVirtualSpace represents process VS (subject). Uid is used to identify user and proctitle to identify the executed command.

Rule groups filesystem VS and process VS together. It defines which subject has which abilities on object.

4.5.3 Virtual space naming

The names of filesystem VSs are based on path entries. The name of filesystem VS is inferred from absolute path with following pattern shown below:

$$/path/to/file \rightarrow path\_to\_file \tag{18}$$

We will refer to the name displayed on the right side in 18 as *path name*. For the naming of process VSs, we used only the UID of a process and path to the executed binary. The filesystem VS is represented mostly by path entries, and those path entries can have different flags turned on or off. For now, we didn't consider the *isSticky* flag and it is not used in the evaluation of a name. We used the following rules to compute the name for a filesystem VS. The syntax $|x|$ is used to represent that x is a placeholder.

all_files The filesystem VS contains only one path entry, where path entry has path "/".

|path-name| The filesystem VS contains only one path entry.

|parent-path-name|_some The filesystem VS contains multiple path entries where all the path entries have value of *isAddition=true* and the same depth.

|parent-path-name|_with_exceptions The filesystem VS contains multiple path entries and all path entries have value of *isAddition=false* and the same depth.

|parent-path-name|_with_children The filesystem VS contains multiple path entries. Not all path entries have the same depth. There is only one path entry which has the minimum depth (i.e. only one entry has depth 2 while others have 3 or higher). This path entry is considered to be the parent of other path entries. All path entries have value of *isAddition=true*

|parent-path-name|_with_children_with_exceptions The filesystem VS contains multiple path entries where not all path entries have the same depth and different value of *isAddition*. There is only one path entry which has the least depth (i.e. only one entry has depth 2 while others have 3 or higher). Path entry with least depth is considered to be the parent of other path entries.

UNDEFINED The filesystem VS contains multiple path entries whose paths do not have common parent path (with exception of "/").

The patterns described above are ways how to name the VSs but we can encounter name conflict under following circumstances:

- There are filesystem VSs with the same name containing different path entries in itself, which cannot be grouped together because different access types are required.
- There are filesystem VSs with the same name containing different path entries but they cannot be grouped because they are defined for different process VSs.

We have proposed solution for both of these problems. The first problem can be solved with encoding required access types into the name as its suffix. This solves the collision because all the path entries in one directory sharing the same access types are grouped together and that ensures that we won't get any conflict. The second problem can be solved by prefixing the created filesystem VS with the UID of process VS name.

4.5.4 Initial rules creation

As usual, we outlined the function body in Algorithm 2. The goal of initial rules creation step is to process the log file and output the initial rules that it is possible to work with in later stages of the algorithm. There are couple of functions that should be explained in more details because they are are not clear implicitly from the context. The initial value of R argument can be empty set.

First function, which is worth of attention is *createPathEntriesFrom()*. In Algorithm 3 we provide a pseudocode for creation of one path entry, because this function can be then used repeatedly for each item in a list.

- **path**, **nametype**, **isDirectory** and **isSticky** are fields directly retrieved from the entry
- **isAddition** is assigned to *true* each time because of the assumption that we made in chapter 4.5.1. We assume that the system is functioning normally and we are

Algorithm 2 Generate initial rules function

```
1: function GENERATE_INITIAL_RULES(L_N, C, R)
2:     if L_N.isEmpty() then
3:         R ← R.removeDuplicateRules()
4:         return R
5:     else
6:         [uid, proctitle, syscall, items] ← L_N.first()
7:         accTypes ← C.getRequiredAccessTypesFor(syscall)
8:         pathEntries ← createPathEntriesFrom(items)
9:
10:        fsVs ← createFsVirtualSpaceFrom(pathEntries)
11:        processVs ← [uid, proctitle]
12:        R_new ← R ∪ createRules(processVs, fsVs, accTypes)
13:        L_N,new ← L_N.remove([[R_new]])
14:        return GENERATE_INITIAL_RULES(L_N,new, C, R_new)
15:    end if
16: end function
```

gathering access requests that should be allowed. If the requirements and environment changes, and the goal would be to mine an existing policy enforced in the system, then it would be required to check whether the access request isn't denied with *EPERM* or *EACCESS* failure code.

- **isRecursive** is assignable only to path entries which are directories. Before we dive into the second part of AND clause on line 7, it would be fair to recall, that if a path entry has *isRecursive* flag set to true, it means that access is given to the directory and the children of this directory. In another words, this flag represents means to generalize access requests. The way how the algorithm is designed, the generalization happens by induction. If the algorithm can access a certain file and perform read or write operation, the change is affecting only that one file. However, if the process can alter the structure of a directory, it is in most cases safe to assume that it has the privilege to modify other files in the directory the same way. Therefore, if a process could successfully create a new directory (which is altering a structure of a parent directory), then the access must be allowed not just in the created directory, but also in its parent and therefore the *isRecursive* flag should be enabled. The condition $nametype = PARENT$ is evaluated to *true* if the item is passed to the

Algorithm 3 Create path entry

```
1: function createPathEntryFrom(item)
2:     path ← [item.name, getPathDepth(item * path)]
3:     nametype ← item.nametype
4:     isDirectory ← item.mode = DIR
5:     isSticky ← item.isSticky
6:     isAddition ← true
7:     isRecursive ← isDirectory ∧ nametype = PARENT
8:     pathEntry ← [path, isDirectory, isAddition, isSticky, isRecursive]
9:     return pathEntry
10: end function
```

syscall as a parent directory of another item, on which the action is performed.

Function *createFsVirtualSpacesFromEntries()* creates one VS for each path entry. We consider this being a straightforward operation and we omit the further details. The name of a created VS is assigned according to naming rules in chapter 4.5.3.

Function *createRules()* iterates over filesystem VSs and creates one rule for each. Process VS and access types are fixed for each rule.

The line 13 in Algorithm 2 contains action $L_N.remove([[R_{new}]]_{L_N})$. Syntax $[[R_{new}]]_{L_N}$ retrieves all entries from L_N which are allowed by rules R_{new} (Definition 11). The whole expression can be interpreted as "remove those entries from L_N, which are allowed by rules R_{new}".

4.5.5 Rule merging

The goal of this step is to group the related path entries together and reduce the number of rules as much as possible. The algorithm uses bottom-up approach. We find optimal number of rules for each path depth starting with higher depths and iteratively ascending to lower depths (higher depth means that the path is lower in the hierarchy). The initial value of *currentDepth* is maximum depth in a path entry among all path entries in the security policy.

The code for each of the functions is in this case implementation-dependent and we won't try to provide pseudocode but rather description of goals of these functions.

- Function *getRulesWithPathDepthsGreaterOrEqual()* on line 4 iterates over the set of rules R and partitions them into two groups. One group is $R_{relevant}$ which represent rules that contain only paths with depth greater or equal to *currentDepth*. The

Algorithm 4 Merge rules algorithm

1: **function** MERGE_RULES($R, currentDepth$)
2: **if** $currentDepth \leq 1$ **then return** R
3: **else**
4: $[R_{relevant}, R_{remaining}] \leftarrow getRulesWithPathDepthsLessOrEqual(R, currentDepth)$
5: $R_{currentDepth} \leftarrow getRulesWithOnlyCurrentDepthPaths(R_{relevant})$
6: $paths \leftarrow getUniqueParentPaths(R_{currentDepth})$
7: $[groups, R_{unmergeable}] \leftarrow groupRulesByPathSubjectActionVs(paths, R_{relevant})$
8: $R_{merged} \leftarrow mergeRulesBy(groups)$
9: $R_{new} \leftarrow R_{remaining} \cup R_{merged} \cup R_{unmergeable}$
10: **return** $MERGE\_RULES(R_{new}, currentDepth - 1)$
11: **end if**
12: **end function**

second group $R_{remaining}$ contains rules which do not fall into the first category and are irrelevant for this recursive call.

- Function *getRulesWithOnlyCurrentDepthPaths()* on line 5 selects rules which contain only path entries whose all paths have depth equal to the current depth.
- Function *getUniqueParentPaths()* on line 6 iterates over rules, and it selects the path of first path entry from each rule. Then the function computes parent path for each selected path. The result set may contain duplicities, we want to return only the unique paths.
- Function *groupRulesByPathSubjectActionVs()* on line 7 groups the rules $R_{relevant}$ by subject, parent path and by access types (tuple $[subject, path, accTypes]$). The rules are at first grouped by process VS. Then the rule can be grouped under some parent path if path in path entries is descendant of a parent path (i.e. /var/lib/mysql/mariadb is descendant of path /var/lib). Then the groups are further split into subgroups, in which all rules share the same access types. The acquired groups are returned in variable *groups*. We would like to recall, that $R_{currentDepth} \neq R_{relevant}$ and the algorithm can encounter a scenario where some rules in $R_{relevant}$ cannot be grouped under any parent path because these parent paths were taken from $R_{currentDepth}$. These ungrouped rules are then assigned to set $R_{unmergeable}$.
- Function *mergeRulesBy()* on line 8 iterates over groups. A new rule is created for

each group and the rule contains path entries of all rules in the group. Process VS and access types are copied from the group. The created rules are then stored in set R_{merged}.

- Line 9 concatenates sets $R_{remaining}$, $R_{unmergeable}$ and R_{merged} into joined set R_{new}.
- Line 1 is a base case for recursion and line 10 is a recursive call.

The initial value of *currentDepth* in the first recursive call is the maximum depth among all the path entries. The goal in each recursive call is to find optimal number of rules for *currentDepth*.

The important property that is not clear implicitly, is that in each step, the algorithm merges only rules that contain path entries with the shared parent path. In the function *groupRulesByPathSubjectActionVs()* on line 7, we mentioned that it is sufficient to check whether the path is a parent path of the first path entry in a filesystem VS. The reason why we don't need to check all of the path entries in a VS is because the VSs that haven't been merged before, will contain only the initial path entry. The filesystem VSs which contain multiple path entries are the results of merging in previous recursive calls. If a filesystem VS is a result of merging, then the property that all path entries in a rule need to share the same parent must hold. Therefore if a path entry is a parent entry of at least one path entry, then we can be sure that it is a parent entry of all path entries in a filesystem VS.

As mentioned earlier, line 1 is a base case for recursion. The algorithm terminates when $currentDepth \leq 1$ and since it covers two cases that can occur, namely when the *currentDepth* equals either to 0 or 1, we can explain these cases in detail.

currentDepth = 1 The algorithm can execute all steps without any special handling. However, we receive undesired result on line 6 after *getUniqueParentPaths()* is executed. The problem is that the function can yield as a result only path "/" for any path with depth equal to 1. Path "/" is parent path of all paths (each path is absolute path and starts with "/"), meaning that all path entries can be possibly merged into one rule. The distinction by access types would split the one large group into subgroups with different access types, creating up to 7 different rules in total (all possible combinations of access types read, write and see). While this would create less VSs and it seems that it aligns with the goal of this phase, we don't consider it to be a step towards better policy. The problem with having VSs where depth 1 paths are grouped together is that the logical connections are lost. The way

how UNIX-like systems are organized is that depth 1 paths are directories that split the system logically. If we merge these path entries together, we get only VSs which vary only in the assigned access types rather than what the VS represents.

currentDepth = 0 This case would require special handling because the only path that has depth equal to 0 is the root path itself. We would encounter a problem in the function *getUniqueParentPaths()* on line 6 because the root directory doesn't have a parent if we don't consider that root is parent to itself. If we consider the first option, we would not get any results if we try to group by "undefined path". If we consider the latter option, we would run into the same problem as described in the previous case.

4.5.6 Rule simplification

The goal of simplification phase is to reduce the number of elements in each rule while not changing its meaning. We cannot reduce the number of subjects or access types because that would change the meaning of a rule. However, we can try to simplify the number of path entries in filesystem VSs by removing redundant path entries. Redundant path entries are those that are already covered by other more general path entries.

Perfect example are two path entries where one path entry is recursive and another is its child entry. If both path entries share the same *isAddition* value then the child path entry is already contained within its recursive parent entry and therefore can be removed.

The Algorithm 5 is initialized with *currentDepth* equal to the minimum depth that can be found in a path entry of a filesystem VS in given *rule*. The *maxDepth* is the maximum depth that can be found in a path entry.

- Line 2 represents base case for recursive call. Lines 4 and 5 are only assignments.
- Function *getRecursiveEntriesWithDepth()* on line 6 iterates over pathEntries and yields only entries matching the value of *currentDepth* and having flag *isRecursive* enabled.
- Function *removeChildEntriesOf()* removes all path entries for which the relevant entry is considered to be parent entry and they share the same value of *isAddition*.
- Function *getFsVirtualSpaceName()* creates name for filesystem VS from path entries according to rules described in chapter 4.5.3.

The simplify phase contrary to merge phase uses top-down approach instead of bottom-up. Both approaches work and return the same result, but top-down saves computation

Algorithm 5 Simplify rule algorithm

1: **function** SIMPLIFY_RULE($rule, currentDepth, maxDepth$)
2: **if** $currentDepth \geq maxDepth$ **then return** $rule$
3: **else**
4: $[subject, fsVirtualSpace, actionVs] \leftarrow rule$
5: $pathEntries \leftarrow fsVirtualSpace.pathEntries$
6: $relevantEntries \leftarrow pathEntries.getRecursiveEntriesWithDepth(currentDepth)$
7: $remainingEntries \leftarrow pathEntries.removeChildEntriesOf(relevantEntries)$
8: $name \leftarrow getFsVirtualSpaceName(remainingEntries)$
9: $newFsVirtualSpace = [name, remainingEntries]$
10: $newRule \leftarrow [subject, newFsVirtualSpace, actionVs]$
11: **return** $Simplify\_Rule(newRule, currentDepth + 1, maxDepth)$
12: **end if**
13: **end function**

resources. Since we keep track of depth in paths, the organization of path entries is hierarchical. Bottom-up approach starts iterating through path entries with higher depth first. If recursive path entries are successfully found, the children that are covered by these path entries are deleted. The problem is that some computation was performed for these path entries with recursive flag but there can exist some path entries higher in the hierarchy which are covering these recursive path entries with higher depths. If we start the simplification from the top (from root directory) and remove children which are already covered by recursive path entries, some of these children could also be recursive and cover some of their children. If we remove these entries, there will be less computation in the later recursive calls and we avoid the problem that bottom-up approach has.

4.6 Output module

Last module in our program is the output module. The goal of the output module is to serialize result from previous module to a file. We transformed the output rules from previous module to DTO and serialized it using JSON format. An example output entry is shown in Listing 2

```
{
   "SubjectUid": "mysql",
   "SubjectProctitle": "/usr/libexec/mariadbd",
   "Object": {
```

```
      "Identifier": "var_ws",
      "Paths": [
        {
          "IsRecursive": false,
          "IsSticky": false,
          "FullPath": "/var",
          "IsAddition": true,
          "Type": "DIR"
        }
      ]
    },
    "Permissions": "ws"
}
```

Listing 2: Example output entry

Figure 5: DTO Output structure diagram

4.7 Generation of Constable configuration file

This chapter describes which techniques we used for generation of Constable configuration file. The generation of Constable configuration is implemented as context-free grammar. The input to the generation procedure is JSON output shown in Listing 2.

We had following goals in mind when designing the grammar:

- The grammar was designed to produce strings that are formatted and human-readable.
- The grammar consists only from rules which start with a set of terminals, a set of terminals followed with a set of non-terminals or a set of non-terminals. We avoided combinations of terminals and non-terminals. This simplifies the implementation because our grammar could be reduced and redesigned to regular grammar if required. However that would introduce a lot of new rules and since we wanted to keep the grammar simple, we stick with a context-free grammar.
- The grammar is intentionally non-generic. It does not describe the Constable configuration syntax and cannot be used as algorithm for parsing. The rules are

designed to be able to generate the configuration in a valid format and to cover everything that was required in order to test the mined policy.

4.7.1 Used symbols

The grammar defines two sets of symbols – terminals and non-terminals. We won't provide the full list of terminals and non-terminals because they can be inferred from the derivation rules. The non-terminals are all symbols in square brackets, i.e. [start]. Any other symbol is considered to be terminal.

However there is special type of terminal known as *placeholder*, which is identified with double opening and closing angle-brackets, i.e. «id». Placeholders are replaced during generation with user-defined values.

4.7.2 General derivation rules

This chapter provides overview of derivation rules that do not fit into any category of Constable configuration.

Table 6: General derivation rules

Input		Output
[start]	→	[file_tree] [newl] [process_tree] [newl] [vs_defs] [newl] [rules] [newl] [functions]
[newl]	→	\n

- Non-terminal [**start**] represents the outline of the whole configuration file. The rule is designed in such a way, that it is clear what is the high-level structure of the configuration file.
- Non-terminal [**newl**] is used to represent newline symbol.

4.7.3 Tree definition

The chapter specifies derivation rules for generation of trees that are defined as children of the virtual root node of the Constable tree.

Table 7: Derivation rules related to Constable trees

Input		Output
[file_tree]	→	[file_tree_def] [newl] [is_primary_tree]
[file_tree_def]	→	tree "«id»" clone of file by getfile getfile.filename;
[is_primary_tree]	→	primary tree "«id»"; | ϵ

[process_tree]	→	[process_tree_def]
[process_tree_def]	→	tree "«id»" of process;

- Non-terminal [**file_tree**] encapsulates required information for definition of filesystem tree in Constable config.
- Non-terminals [**file_tree_def**] and [**is_primary_tree**] contain details about the filesystem tree. The **«id»** terminal in this context is the placeholder for the name of a filesystem tree, we have given it the name "fs" by default.
- Non-terminal [**process_tree**] is only substituted by non-terminal [**process_tree_def**] and could be possibly ignored. We defined it only for better readability of rules and for consistency with [**file_tree**] non-terminal. The non-terminal [**process_tree_def**] defines the tree of processes.

4.7.4 VS definitions

This chapter provides overview for derivation rules for VS – both filesystem VS and process VS.

Table 8: Derivation rules related to VS

Input		Output
[vs_defs]	→	[vs_fs_defs] [newl] [vs_proc_defs] [newl] [vs_proc_inits]
[vs_fs_defs]	→	[vs_fs_def] [newl] [vs_fs_defs] | ϵ
[vs_fs_def]	→	space «id» = [path] [add_paths]
[path]	→	recursive «path_value» | «path_value»
[add_paths]	→	[op] [path] [add_paths] | ϵ
[op]	→	+ | -
[vs_proc_defs]	→	[vs_proc_def] [newl] [vs_proc_defs]
[vs_proc_def]	→	space «uid» = «domainName»/«uid»;
[vs_proc_inits]	→	[vs_proc_init] [newl] [vs_proc_inits] | ϵ
[vs_proc_init]	→	«id» \t ENTER «id», READ «id», WRITE «id», SEE «id»;

- Non-terminal [**vs_defs**] encapsulates the definitions of process and filesystem VSs.
- Non-terminal [**vs_fs_defs**] compounds definitions of filesystem VSs together. The

derivation rule contains two branches. The first branch, where new non-terminals are created is used if not all filesystem VSs contained in rules are covered. The ϵ branch is used if there is no filesystem VS to be covered. The list of filesystem VSs is extracted from input rules.

- Non-terminal [**vs__fs__def**] encapsulates single filesystem VS definition. The property that needs to be kept is that each filesystem VS needs to have at least one path to be considered valid.
- Non-terminal [**path**] represents one path entry which is contained in a filesytem VS. The non-terminal has two possible branches. The first branch is used in case the path entry is a directory and has *isRecursive* flag. The latter branch is used in case the path entry is a file or a non-recursive directory.
- Non-terminal [**add__paths**] is a non-terminal whose use case is to define additional path entries in case the filesystem VS contains multiple path entries. The first branch of this non-terminal is used if there are some uncovered path entries, the ϵ branch is used if all path entries are covered.
- Non-terminal [**op**] is related to a path. It is required only if a filesystem VS contains multiple path entries because it represents relationship between them. It evaluates to "+" if the path entry has value of *isAddition=true*, otherwise it evaluates to "-".
- Non-terminal [**vs__proc__defs**] is similar to the non-terminal [**vs__fs__defs**], but it encapsulates process VSs. Process VSs are extracted from rules and since the rules can very likely contain duplicates, we select only unique combinations of $[uid, proctitle]$.
- Non-terminal [**vs__proc__def**] is used to encapsulate a single process VS definition.
- Non-terminal [**vs__proc__inits**] initializes the process VSs in the list. The first branch of a rule is used if there are some unprocessed process VSs in the list. ϵ rule is used in case all process VSs were initialized.
- Non-terminal [**vs__proc__init**] represents single process VS initialization. «id» is a placeholder for process VS UID.

4.7.5 Security policy rules

This chapter specifies derivation rules for generation of configuration for mined policy rules.

Table 9: Derivation rules related to security policy rules

Input		Output
[rules]	→	[rule] [newl] [rules] | ϵ
[rule]	→	«id» \t [ability] [add_ability]
[ability]	→	READ «id» | WRITE «id» | SEE «id»
[add_ability]	→	, [ability] [add_ability] | ;

- Non-terminal **[rules]** represents the set of input rules to the algorithm. The non-terminal can expand to two possible branches. The first branch is used if there are some uncovered rules in the set. The ϵ rule is used if there are no more rules to be covered.
- Non-terminal **[rule]** encapsulates single rule definition.
- Non-terminal **[ability]** represents single part of a relationship between process VS and filesystem VS.
- Non-terminal **[add_ability]** contains two branches. The first branch is used in case the rule has some uncovered abilities. The second branch is used if all abilities were being covered.

The non-terminals *[left_bracket*, *[right_bracket]* are mappings from a non-terminal to a single terminal. We used this trick to avoid combinations of terminals and non-terminals and at the same time minimize the number of rules in the grammar.

4.7.6 Functions and events

This chapter specified overview of derivation rules for events and functions. Since there is only small part related to functions we provide rules for functions and events in single chapter.

Table 10: Derivation rules related to functions and events

Input		Output
[functions]	→	[handlers] [newl] [enter_domain] [newl] [getprocess] [newl] [constable_init_func] [newl] [init_func] [newl]
[enter_domain]	→	function enter_domain {\n\t enter(process,str2path("«domainName»/"+$1));\n}

[getprocess]	→	* getprocess [left_bracket] [newl] [enter_space] [return_ok] [newl] [right_bracket]
[left_bracket]	→	{
[right_bracket]	→	}
[enter_space]	→	[branching] | [default_enter_space]
[default_enter_space]	→	enter(process,@"«domainName»/init");
[branching]	→	if [expression] [branch_body] [other_branch]
[expression]	→	(process.cmdline == "«proctitle»")
[branch_body]	→	[left_bracket] [newl] [enter_domain_call] [newl] [right_bracket] [newl]
[enter_domain_body]	→	enter_domain(process,@"«domainName»/«uid»");
[other_branch]	→	elseif [expression] [branch_body] [other_branch]
[other_branch]	→	else [expression] [left_bracket] [newl] [default_enter_space] [newl] [right_bracket] [newl]
[return_ok]	→	return OK;

- Non-terminal [**functions**] represents special handling functions in Constable configuration.
- Non-terminal [**handlers**] should represent set of handlers for events in case special handling is required for certain actions. However, in our research, we weren't concerned in collecting handlers from logs, meaning that this non-terminal is currently unapplicable.
- Non-terminal [**constable_init_func**] is used to produce an empty skeleton for *constable_init()* function. The derivation rule is intentionally omitted because it is not required to define.
- Non-terminal [**init_func**] is used to produce an empty skeleton function for *init()* function. The derivation rule is intentionally omitted because it is not required to be defined.
- Non-terminal [**enter_domain**] represents utility function which puts the process VS under a domain.
- Non-terminal `[getprocess]` represents *getprocess* event handler. Since `[getprocess]` is special type of event required for initialization, we specified it explicitly.

- Non-terminal [**enter_space**] represents the body of *getprocess* event handler. It contains two branches. Branch with non-terminal [**branching**] is used if there is more than one unique combination of [uid, proctitle] contained in rules. Other branch is used if we have 0 or 1 process VS.
- Non-terminal [**branching**] represents control flow in a function. The non-terminal has specific use case to represent branches in getprocess function.
- Non-terminal [**expression**] represents expression which evaluates to true or false. Its use case is to check for a certain executed binary.
- Non-terminal [**branch_body**] encapsulates the body of a branch. The only place in the configuration file where the control-flow is used is in *getprocess* function, therefore the non-terminal is bound to only one use-case.
- Non-terminal [**enter_domain_call**] represents call of *enter_domain* function.
- Non-terminal [**other_branch**] is used to add additional control-flow branches to the config. Since we assume that if branching was required, it is either if-else or if-elseif-else control flow. The if-else control flow is used if we need to consider only two branches. If it is required to have 3 or more branches, the program should use if-elseif-else control flow.
- Non-terminal [**return_ok**] represents simple return statement with OK response from the authorization server.
- Non-terminals [**left_bracket**] and [**right_bracket**] are mappings to "{" and "}" terminals.

4.8 Evaluation

We tried to analyze the algorithm and the properties of security policy from three perspectives, but only two succeeded.

4.8.1 Metric evaluation

As we stated in the Definition 16, we chose to optimize the policy according to *WSC* policy quality metric. We tried to find out the relationship between the *WSC* and number of entries that are provided as input to the algorithm. We mined the security policy for 1 entry from L and then computed the *WSC* after each phase of the algorithm. Then we were iteratively increasing the number of selected entries and repeated the process.

Labels We computed the *WSC* of security policy after each phase of the algorithm because we tried to infer the properties of the algorithm and the generated security policies. To distinguish in which phase the *WSC* was computed, we introduced the following labels:

INITIAL The *INITIAL* label represents the computed *WSC* after the Algorithm 2 finished and duplicit rules were removed.

AFTER MERGING The *AFTER MERGING* label represents the computed *WSC* after the Algorithm 4 finished.

FINAL The *FINAL* label represents the computed *WSC* after all the algorithms finished, and the program gave mined rules as the output.

Weights Before we could even compute the *WSC*, we had to select the weights for each set defined in equation 16. The weight configuration overview is shown in Table 11.

Table 11: Configuration of weights for WSC

Weight name	Weight symbol	Value
Weight for subject context set S_C	w_s	0.5
Weight for object context set S_O	w_o	1
Weight for operation context set OP_C	w_{op}	0
Weight for rules set R	w_r	0.75

Weight for operation context set OP_C The w_{op} was given zero because the operation context in case of Medusa is still the same regardless of the security policy. It only contains the information about the required access type for each operation and the name of the operation and that does not add up to the overall complexity in our case.

Weight for subject context set S_C The w_s is given 0.5 because when we were designing the algorithm, we weren't really concerned with reducing the *WSC* by optimizing the subject context. We decided to take subject context into the account, but it shouldn't be as important as object context, which had most of the attention in our algorithm.

Weight for object context set O_C The w_o is set to 1 to give it express higher significance comparing to other sets. Since our algorithm tried to optimize mostly filesystem VSs, the most impact on the complexity should have the object context.

Weight for rules set R A fewer number of rules leads to fewer filesystem VSs with more path entries. We chose the number 0.75 to express higher significance in the security policy than subject context and less significance than object context.

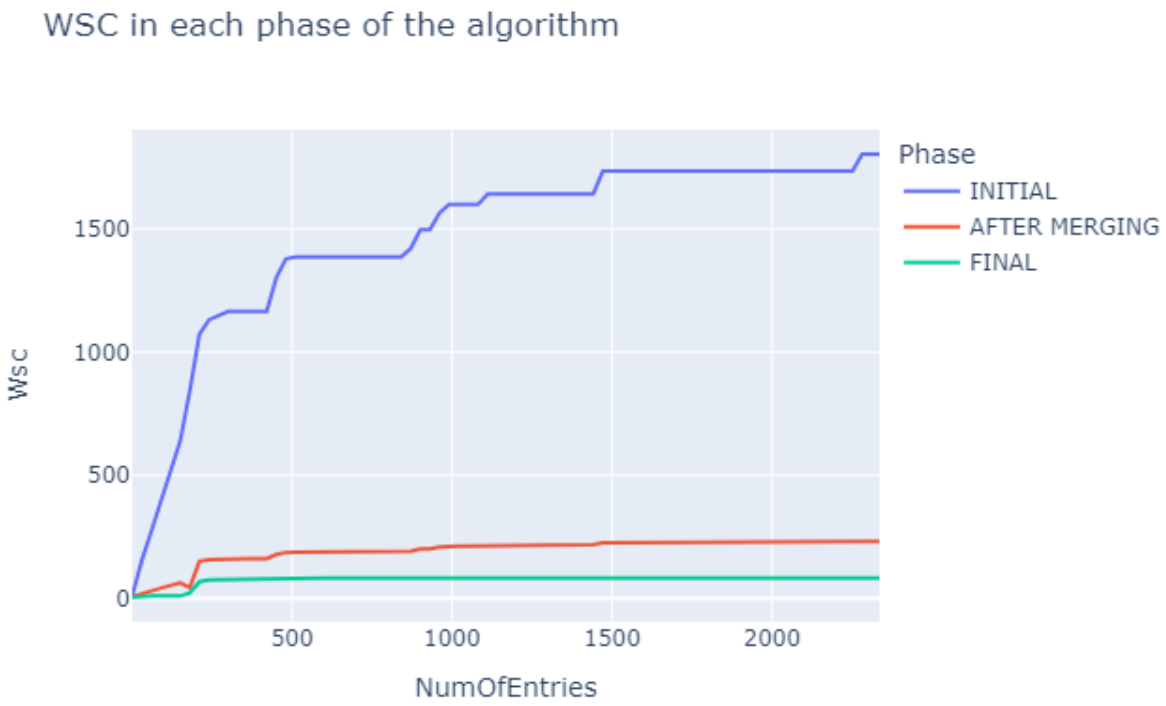

Figure 6: Relationship between WSC and number of entries

Interpretation of WSC by labels in isolation We would like to provide first the interpretation of data in Figure 6 from the standpoint of individual labels.

INITIAL The values on y-axis follows two phenomenons — either the *WSC* increases with number of input entries (x-axis) or the *WSC* stagnates. As can be seen, the line *WSC* is never decreasing and that is because of the nature of how the rules are generated in the Algorithm 2. We generate new rule for each file that appears in any log entry in log file L. After we process each log entry, we can remove the duplicit rules. Therefore, the number of rules can increase with each processed entry or it can stay the same, but it will never decrease.

Another thing that we would like to draw attention to is the slope of the line. The higher increase of *WSC* can be seen in the interval $x \in [0, 500]$. This is the phenomenom that we described in chapter 2.4.1 with the real logs. Most of the access requests at some point start to repeat. If we haven't covered many entries, there is higher chance that new rules are created and this it the reason why the line in mentioned interval has steep slope.

On the other hand, the more entries we processed, the higher the likelihood that we encounter a duplicit access request. We visually illustrated the case in the Figure 7 with

the yellow color.

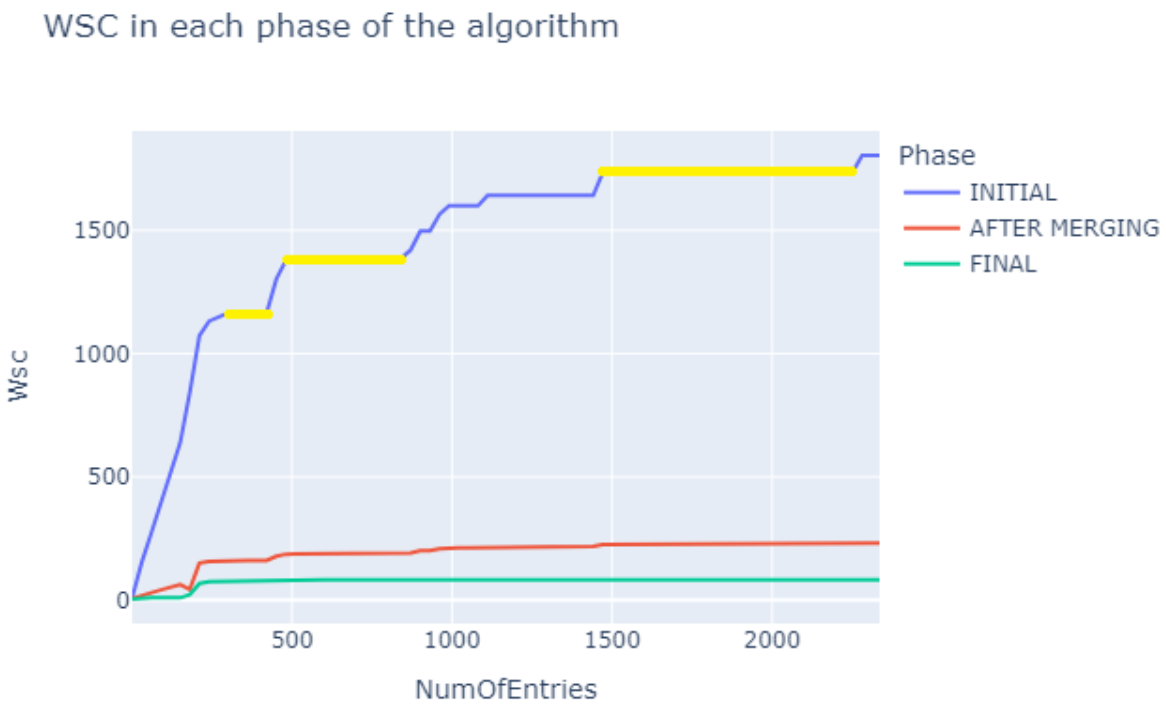

Figure 7: The increasing stagnation of WSC with increasing number of log entries

AFTER MERGING The values on the y-axis can at any point increase, decrease or stagnate. The increase and stagnation of values correlates with data with *INITIAL* label. The increase is logical because if initial number of rules increases, that means that when rules are merged, the filesystem VSs will contain more path entries and that leads to higher *WSC*. The stagnation occurs if no new rules are created with additional log entries.

However, a very interesting phenomenon is decrease of *WSC* with increasing number of log entries after merge phase. At first, we thought that it is not possible and that the decrease shown in Figure 8 has occurred due to some defect in our implementation. After further investigation, we found out that actually the decrease of *WSC* in some edge cases is possible.

Let's consider the snippet of Constable configuration shown in Listing 3. Let's define filesystem VSs `a, b` and `c`. These three VSs could not be merged because there is no VS that contains path `/var/log`,

```
...
space a = /var/log/mariadb/mariab.log;
```

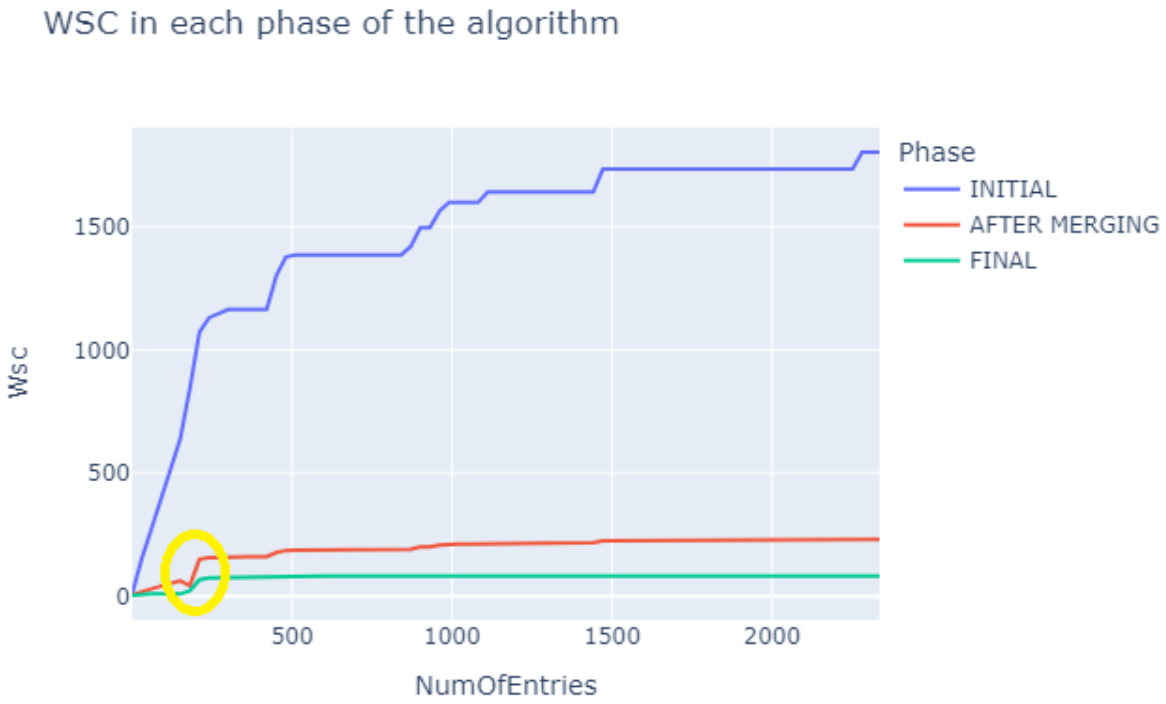

Figure 8: The decrease of WSC on increasing number of log entries in AFTER MERGING data

```
space b = /var/log/mysql/mysql.log;
space c = /var/log/postgress/postgress.log;
...
```

Listing 3: Example entry in a log file

Let's compute the WSC for these two filesystem VSs, which can be accessed by subject `mysql`. We will use the weight $w_o = 1.0$ as we declared in Table 11.

$$\begin{aligned} WSC(S_C) &= WSC(a) + WSC(b) + WSC(c) = \\ &= (1.0 * (1+1)) + (1.0 * (1+1)) + (1.0 * (1+1)) = \\ &= 6 \end{aligned} \tag{19}$$

As additional step, let's add a new log entry, in which the `mysql` accesses the directory `/var/log`. The filesystem VS `a, b` and `c` could now be merged together. The result is shown in Listing 4.

```
...
space d = /var/log + /var/log/mariadb/mariadb.log
```

```
    + /var/log/mysql/mysql.log + /var/log/postgress/postgress.log;
...
```

Listing 4: Example entry in a log file

If we try to compute the *WSC* for Listing 4 and compare the results of equations 19 and 20 we can see that the VS `d` has lesser complexity than `a,b` and `c` combined together. We would like to also point out, that since we merged 3 filesystem VSs, we also reduced the number of rules in R by 3, which reduces the *WSC* even further.

$$WSC(S_C) = WSC(d) = 1.0 * (1 + 4) = 5. \tag{20}$$

FINAL The values on y-axis can similarly to *AFTER MERGING* data increase, decrease or stagnate. It can increase as a result of increasing number of rules in the policy. It can stagnate in case the security policy generalizes enough and it is not required to add anything to the filesystem VSs even if we add new entries to the log file L. It can also decrease for the very same reason as described with *AFTER MERGING* data.

Relationships between lines We described the phenomenons of labeled data when we look at them individually. However the conclusions can be made if we try to look at the relationships between the data. By analysis, we can better describe the behavior that we observed when analyzed the data individually.

Relationship INITIAL - AFTER MERGING We can observe a rapid decrease in *WSC* between the *INITIAL* and *AFTER MERGING* lines. During the execution of the Algorithm 4, the *WSC* is reduced by reducing the number of rules in R, but also by merging the filesystem VSs together.

Relationship INITIAL - FINAL We can conclude from the relationship, that the mined security policy is either overpermissive or generalizes well. If we take a look at Figure 6, the *WSC* of the *INITIAL* line non-linearly increases, the *WSC* of the *FINAL* line at some point stagnates. The stagnation of *FINAL* line in the context of security policy can be interpreted as a scenario, in which a new rules are created due to more log entries during the execution of Algorithm 2, but those rules are already covered in the security policy and do not contribute to the final security policy in any way. Just by looking at the Figure 6, we cannot conclude whether it is overpermissiveness or generalization. We cannot make any conclusions because from the figure, we don't know any details about the log entries. The log entries could be any of the following.

1. A new log entries can contain access requests, where object is a temporary file. The

temporary files have different name in each run of the program and therefore if those files are already covered in the security policy, it means that the security policy generalizes well.

2. A new log entries can contain access requests, where object is a configuration file. The configuration file could be accidentally covered by the security policy because some created rule is overly permissive.

Relationship AFTER MERGING - FINAL The difference between the *AFTER MERGING* line and *FINAL* line is that the *FINAL* line includes also simplification phase (Algorithm 5). The simplification phase only optimizes the filesystem VSs. Therefore we can consider it being a minor optimization of the security policy.

4.8.2 Manual execution

We considered that we could evaluate the security policy by running it manually on the system. We could see, how the access requests are allowed or denied by the Medusa or the authorization server. However, currently the Medusa does not implement the filesystem open LSM hook. Since most of our acquired access requests, from the audit system, used the filesystem open syscall, it would be pointless to run any manual tests with the generated configuration. Therefore this evaluation technique could not be executed.

4.8.3 Manual evaluation

We tried to evaluate manually the generated configuration file. The concrete analyzed configuration file can be found in attachments as Listing D.1. The generated policy in a file had following properties.

Filesystem VSs with 1 path entry The recursion in Algorithm 4 terminates when it reaches depth 0 or 1. As a result, there are, the filesystem VSs with depth 1 or 0 will never be merged with other filesystem VSs. The problem is that we can have filesystem VS such as `space var_ws = /var`, which could be possibly merged with the filesystem VS such as `space var_lib_with_children = recursive /var/lib`. This shows that the algorithm can be further improved and there is still an opportunity to reach lower *WSC*.

Fine-grained path entries in filesystem VSs The Algorithm 2 creates the initial rules, the filesystem VSs and the process VSs. As we described earlier, the path entries can be considered `recursive`, if the operation, which was performed on this path entry modified the directory structure in any way. However, in some cases, such as accessing a shared libraries in directory `/lib64`, the subject will only access the

files and will not change anything. The limitation here is that the algorithm cannot infer that the subject should be able to access any library in the `/lib64` directory. As a result, the path entries are fine-grained, meaning that the individual files that appeared in the log entries, are enumerated in the filesystem VS.

In the case of the shared libraries, we can consider the enumeration as being a bad property. However in case of the configuration files, it is probably safer when the access is granted only on certain files rather than on any file in some directory.

Filesystem VSs are grouped by hierarchy The merging phase (Algorithm 4) merges the path entries in the filesystem VSs based on hierarchy. As a result, the filesystem VSs have many path entries and each filesystem VS will be accessible at most by 1 process VS.

Filesystem VSs are not modular The created filesystem VSs are grouped by hierarchy and therefore we will not be able group the files modularly. To provide an example, if the user data for some program are scattered across the system, they will not appear in single filesystem VS, but will be part of various filesystem VSs.

4.8.4 Summary of comcluded properties of mined security policy

This chapter presents the conclusions from the previous chapters about the properties of generated security policies.

- The security policy is either overly permissive or can generalize well.
- The path entries in filesystem VSs can be fine-grained.
- The security policy can contain filesystem VSs with depth 1, which can be potentially merged with other filesystem VSs.
- The security policy covers every access request that appeared in the log file L.
- The filesystem VS contains path entries that are grouped hierarchically.

4.8.5 Summary of comcluded properties of policy mining algorithm

This chapter presents the conclusions from the previous chapters about the properties of policy mining algorithm, which was presented as Algorithm 1.

- The algorithm is deterministic and each time provides the same output for the same input.

- The algorithm ensures that each access request in the log file is covered in generated security policy.
- The higher number of input entries to the algorithm can lead to more generic filesystem VSs.
- The algorithm is data sensitive. If an operation in the access request do not modify the structure of a directory, the algorithm cannot produce security policies that have ability to generalize. As a result, the accessed files are enumerated in the filesystem VSs.

Conclusion

All the objectives specified in the assignment are completed. We analyzed the current state-of-the-art policy mining researches and described their findings in the chapter 2.1 with our own terminology. We also designed and implemented a new policy mining algorithm.

In chapter 4.8 we evaluated the algorithm and mined policies from different perspectives. However, we could not test the mined policy in the real system due to a missing implementation of the filesystem open operation in Medusa. The evaluation of mined policy have shown that the proposed algorithm still has its flaws. The next research can potentially try to fix the identified imperfections. The next step towards mining of more practical security policies is to adapt the algorithm to be able to mine the restrictive policies.

Another step forward is to investigate the generalization property of the algorithm. As we discussed in the evaluation chapter 4.8, we could not conclude whether the mined policy is overly permissive or has the ability to generalize well.

We are also not sure whether the naming rules proposed for VSs covers all possible scenarios that can occur during the policy mining. However, missing rules can eventually be found when more complex policies are mined.

However, we are sure that policy mining of security policies for operating system is nowadays possible. Although the operating system is a complex entity and requires a lot of knowledge, the first experiments have shown that we could achieve relatively successful results. The hard part was coming up with our own algorithm since we have not found a lot of research related to the policy mining of security policies for operating systems.

Resumé

Táto práca sa venuje problematike ťažby bezpečnostnej politiky (ďalej už len politika). Nami naštudovaná literatúra v oblasti ťažby politík sa zameriavala na získanie politiky zo záznamového súboru alebo informácií o už existujúcej politike. Naša práca sa zameriava na ťažbu politiky pre bezpečnostný modul Medúza, ktorý je vyvíjaný na Fakulte elektrotechniky a informatiky na Slovenskej technickej univerzite.

Prvá kapitola práce sa primárne venuje predstaveniu problematiky Medúzy. Medúza je vyvíjaná v súčasnosti primárne ako bezpečnostné riešenie pre operačný systém GNU/Linux. Medúza implementuje tzv. LSM framework, pretože vývojári Medúzy sa dlhodobo snažia o jej začlenenie do Linuxového jadra. Unikátnosť riešenia Medúzy oproti iným bezpečnostným riešeniam spočíva v tom, že iba časť bezpečnostnej rozhodovacej logiky je implementovaná v Linuxovom jadre. Zodpovednosť rozhodovania je prenesená na proces bežiaci v užívateľskom priestore, tzv. autorizačný server.

V kapitole o Medúze sme zahrnuli jej architektúru, proces rozhodovania a definície pre jej spôsob riadenia prístupu. Architektúra Medúzy pozostáva zo 4 vrstiev, ktoré sú pomenované L1 až L4. V našej práci boli podstatné najmä vrstvy L2 a L4. Ak proces (ďalej len subjekt) v systéme chce vykonať operáciu nad súborom (ďalej len objekt), jadro vytvorí žiadosť o prístup. V prípade, že Medúza monitoruje operáciu v žiadosti, môže na ňu patrične zareagovať a žiadosť zamietnuť. Žiadosť je spracovávaná Medúzou na vrstve L2 a v prípade, že Medúza prístup nezamietla, kontaktuje autorizačný server z vrstvy L4, ktorý môže na žiadosť dodatočne zareagovať a žiadosť zamietnuť.

Model riadenia prístupu, tzv. VS model, si vyžaduje zaradenie všetkých subjektov a objektov do doménových skupín, tzv. virtuálnych svetov. V našej práci sme začali rozlišovať dve veľké kategórie virtuálnych svetov — virtuálne svety, do ktorých patria subjekty, nazvané "procesové virtuálne svety" a virtuálne svety, do ktorých patria objekty, nazvané "virtuálne svety súborového systému". Zaradenie subjektov a objektov do virtuálnych svetov vykonáva autorizačný server podľa konfigurácie a tieto informácie sú následne predané Medúze. Autorizačný server taktiež definuje, aký typ prístupu môže subjekt nad objektom vykonať. V súčasnej situácii sú definované 3 typy prístupu - READ (čítanie), WRITE (zápis) a SEE (videnie).

Nakoľko v rozhodovacom procese figuruje aj samotný autorizačný server, v práci je obsiahnutá aj kapitola venovaná popisu vybraných častí, týkajúcich sa autorizačného servera Constable. Konkrétne sme popisovali, ako Constable spracováva požiadavku od Medúzy, akým spôsobom je požiadavka predaná, ako je vykonávaný rozhodovací proces a

prešli sme základnú syntax konfiguračného súboru.

Druhá kapitola je venovaná teórii o ťažbe politík. V kapitole sa snažíme objasniť základnú terminológiu, ktorá sa ku ťažbe politík viaže, spracovali sme poznatky z naštudovanej literatúry, rozoberáme požiadavky na algoritmus a na záver sa venujeme metrikám, s pomocou ktorých môžu byť vyťažené politiky vyhodnocované.

Predošlé výskumy v oblasti ťažby politík sa zameriavali na ťažbu politík pre modely riadenia prístupu ABAC, ReBAC alebo RBAC. Použitá terminológia v každej literatúre bola rozličná a bola adaptovaná pre konkrétny model riadenia prístupu. Nakoľko sme nenašli literatúru, ktorá by popisovala problematiku ťažby politiky pre MAC model riadenia prístupu, terminológiu sme si museli zaviesť vlastnú. Poznatky o ťažbe politík sme teda popisovali s pomocou nami zadefinovanej terminológie. V rámci terminológie sme zadefinovali všeobecne aj problém ťažby bezpečnostnej politiky pre model riadenia prístupu MAC.

Kapitola ďalej pokračuje popisom vlastností, ktoré vyťažená politika môže mať, a využitia ťažby politík v praxi. Identifikovali sme 5 základných vlastností.

Schopnosť zovšeobecňovať Politika by mala byť schopná vzťahy medzi subjektami a objektami zovšeobecňovať. To znamená, že politika by mala byť schopná správne reagovať aj na žiadosti o prístup, pre ktoré záznamový súbor nevykazuje žiadnu evidenciu.

Úplnosť Politika by mala byť úplná. To znamená, že by mala byť schopná zareagovať na akúkoľvek žiadosť o prístup.

Správnosť Politika by mala byť schopná zareagovať správne na akúkoľvek požiadavku o prístup. Správne sa v tomto prípade myslí vzhľadom na politiku, z ktorej nová politika vychádza, alebo vzhľadom na záznamový súbor, z ktorého bola nová politika vyťažená.

Vylepšenie Ak nová politika vychádza z inej politiky, mala by byť vylepšením predošlej politiky.

Stručnosť Politika by mala generovať pravidlá, ktoré sú ľahko interpretovateľné človekom.

Algoritmy na ťažbu politík sú vyvíjané za účelom splnenia určitého cieľa. V našej práci dokumentujeme 3 unikátne ciele, ktoré môžu byť motiváciou pre vývoj nového algoritmu.

Zjednodušenie existujúcej politiky Algoritmus dostane na vstup už existujúcu politiku a záznamový súbor. Cieľom je vytvoriť novú politiku alebo rozšíriť pravidlá pre

existujúcu.

Vygenerovanie novej politiky Algoritmus dostane na vstup záznamový súbor a vytvorí politiku, ktorá bude pokrývať žiadosti o prístup v súbore.

Migrácia politiky na iný bezpečnostný model Algoritmus dostane na vstupe politiku a záznamový súbor. Výstupom algoritmu je politika, ktorá je ekvivalentná vstupnej politika, ibaže v požadovanej reprezentácii. Tento prístup je vhodný v prípade, že chceme zmeniť model riadenia prístupu a prejsť napríklad z modelu riadenia prístupu RBAC na model riadenia prístupu ABAC.

Záverom teórie o ťažbe politík je popis typov záznamových súborov, ktoré sa môžu objaviť ako vstup do algoritmu, a metriky na analýzu výstupu z algoritmu. Záznamové súbory sú rozdelené na dva základné typy — syntetické a reálne záznamové súbory. Pre analýzu výstupu uvádzame 3 metriky, ktoré sú zvolené tak, aby vyhodnocovali rôzne vlastnosti politík a aby bolo možné s nimi prípadne aj vygenerované politiky medzi sebou porovnať. V práci prezentujeme metriku označenú pod skratkou *WSC*, ktorá sa používa na vyhodnotenie stručnosti politiky, metriku *F1-score*, ktorá hodnotí presnosť politiky a metriku *spoľahlivosť* (ang. *confidence*), ktorá nie je vhodná na vyhodnotenie politiky ako celku, ale je vhodná na výber pravidiel do politiky, pre dosiahnutie lepšej schopnosti zovšeobecňovania.

Treťou kapitolou je definícia problému ťažby politiky pre bezpečnostný model Medúza. Nakoľko v druhej kapitole sme sa snažili popísať teóriu všeobecne, v tretej kapitole sme zadefinovali konkrétnu inštanciu problému, ktorý riešime v rámci praktickej časti.

Štvrtá časť reprezentuje praktickú časť práce. Jadrom praktickej časti je nami navrhnutý algoritmus na ťažbu politiky. Celkovo v praktickej časti popisujeme ciele praktickej časti, nami použité spôsoby získavania záznamových súborov, nami navrhnutý algoritmus na ťažbu politiky, generovanie konfiguračného súboru pre autorizačný server Constable a následne vyhodnotenie vygenerovanej politiky.

Algoritmus pozostáva z troch fáz.

1. Vygenerovanie počiatočnej politiky
2. Makrooptimalizácia politiky
3. Mikrooptimalizácia politiky

1.fáza sa zameriava na vygenerovanie počiatočnej politiky, ktorá inicializuje virtuálne svety súborového systému, procesové virtuálne svety a vytvorí pravidlá tak, aby bola pokrytá každá žiadosť o prístup v záznamovom súbore.

2.fáza je zameraná na makrooptimalizáciu politiky (taktiež nazvaná aj "fáza spájania pravidiel"). Cieľom tejto fázy je znížiť počet pravidiel a virtuálnych svetov, a zároveň zaistiť, aby boli stále pokryté všetky žiadosti o prístup v záznamovom súbore. Algoritmus spája dohromady pravidlá a virtuálne svety, ktoré sú vyhodnotené ako súvisiace.

3. fáza je zameraná na mikrooptimalizáciu politiky (taktiež nazvaná aj "fáza zjednodušovania pravidiel"). Cieľom tejto fázy je optimalizovať virtuálne svety súborového systému vystupujúce v pravidlách. V rámci tejto fázy sme navrhli aj pravidlá na pomenovávanie virtuálnych svetov. Po zjednodušení virtuálnych svetov sa vygeneruje unikátny názov pre každý virtuálny svet.

Výstup z algoritmu je spracovaný modulom na generovanie konfiguračného súboru pre autorizačný server Constable. Generovanie konfiguračného súboru bolo realizované s pomocou bezkontextovej gramatiky. V práci sme uviedli kompletný prehľad derivačných pravidiel pre navrhnutú gramatiku.

Záverom implementačnej časti je vyhodnotenie vygenerovanej politiky s *WSC* metrikou. Následne, podľa hodnôt *WSC*, sme analyzovali rozdiely v politikách medzi jednotlivými fázami a vyvodili závery.

Táto práca je prínosom z hľadiska spracovania problematiky, zadefinovania problému ťažby politík pre model riadenia prístupu MAC, zadefinovania novej terminológie, ktorá môže byť používaná pri opise poznatkov o ťažbe politík, navrhnutie nového algoritmu na ťažbu politiky a navrhnutie gramatiky, pre generovanie konfiguračného súboru pre autorizačný server Constable.

Bibliography

1. HU, Vincent C, KUHN, Rick, YAGA, Dylan, et al. Verification and test methods for access control policies/models. *NIST Special Publication*. 2017, vol. 800, p. 192.
2. TRACY, Miles, JANSEN, Wayne and MCLARNON, Mark. Guidelines on Securing Public Web Servers Web Servers. *NIST Special Publication*. 2002, vol. 800, p. 44.
3. FORCE, Joint Task. *Security and privacy controls for information systems and organizations*. 2017. Tech. rep. National Institute of Standards and Technology.
4. ZELEM, Marek and PIKULA, Milan. *ZP Security Framework*. Faculty of Electrical Engineering and Information Technology, Slovak University of Technology in Bratislava, 2000. Available at URL: `http://medusa.terminus.sk/English/medusa-paper.ps`.
5. HALLYN, Serge and KEARNS, Phil. Domain and type enforcement for Linux. In: *4th Annual Linux Showcase & Conference (ALS 2000)*. 2000.
6. LOSCOCCO, Peter and SMALLEY, Stephen. Integrating flexible support for security policies into the Linux operating system. In: *2001 USENIX Annual Technical Conference (USENIX ATC 01)*. 2001.
7. WRIGHT, Chris, COWAN, Crispin, SMALLEY, Stephen, MORRIS, James and KROAH-HARTMAN, Greg. Linux Security Modules:General Security Support for the Linux Kernel. *Proceedings of the 11th USENIX Security Symposium* [online]. 2002 [visited on 2020-05-06]. Available from: `https://www.usenix.org/legacy/event/sec02/full_papers/wright/wright.pdf`.
8. PIKULA, Milan. *Distribuovaný systém na zvýšenie bezpečnosti heterogénnej počítačovej siete*. 2002. MA thesis. Katedra informatiky a výpočtovej techniky, Fakulta elektrotechniky a informatiky Slovenskej technickej univerzity v Bratislave.
9. KÁČER, Ján. *Medúza DS9*. 2014. Available also from: `https://opac.crzp.sk/?fn=detailBiblioForm&sid=FF3F620FA7CAEDC57A1EA38ABC7C`. MA thesis. Ústav informatiky a matematiky, Fakulta elektrotechniky a informatiky Slovenskej technickej univerzity v Bratislave. EČ: FEI-5384-64746.
10. ZELEM, Marek. *Integrácia rôznych bezpečnostných politík do OS Linux*. 2001. MA thesis. Katedra informatiky a výpočtovej techniky, Fakulta elektrotechniky a informatiky Slovenskej technickej univerzity v Bratislave.

11. KYSEĽ, Matúš, MAJZEL, Michal, KRAJČÍR, Martin, PROCHÁZKA, Matej and KÚKEĽ, Matúš. *Medúza Voyager*. 2015. Tech. rep. Ústav informatiky a matematiky, Fakulta elektrotechniky a informatiky Slovenskej technickej univerzity v Bratislave.

12. KOŠARNÍK, Peter. *LSM Medusa*. 2020. Available also from: `https://opac.crzp.sk/?fn=detailBiblioForm&sid=AFB64E0160B4F4E92F46D69F9648`. B.S. th. Institute of Computer Science and Mathematics, Faculty of Electrical Engineering and Information Technology, Slovak University of Technology in Bratislava. RN: FEI-5382-86207.

13. LORENC, Václav. *Konfigurační rozhraní pro bezpečnostní systém*. 2005. MA thesis. Fakulta informatiky Masarykovej univerzity v Brne.

14. XU, Zhongyuan and STOLLER, Scott D. Mining attribute-based access control policies. *IEEE Transactions on Dependable and Secure Computing*. 2014, vol. 12, no. 5, pp. 533–545.

15. COTRINI, Carlos, WEGHORN, Thilo and BASIN, David. Mining ABAC rules from sparse logs. In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018, pp. 31–46.

16. BUI, Thang, STOLLER, Scott D. and LI, Jiajie. *Greedy and evolutionary algorithms for mining relationship-based access control policies*. 2019. ISSN 0167-4048. Available from DOI: `https://doi.org/10.1016/j.cose.2018.09.011`.

17. ABU JABAL, Amani, BERTINO, Elisa, LOBO, Jorge, LAW, Mark, RUSSO, Alessandra, CALO, Seraphin and VERMA, Dinesh. Polisma-a framework for learning attribute-based access control policies. In: *European Symposium on Research in Computer Security*. 2020, pp. 523–544.

18. LI, Yun, HUANG, Chenlin, YUAN, Lu, DING, Yan and CHENG, Hua. ASPGen: an Automatic Security Policy Generating Framework for AppArmor. In: *2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. 2020, pp. 392–400.

19. MOLLOY, Ian, PARK, Youngja and CHARI, Suresh. Generative models for access control policies: applications to role mining over logs with attribution. In: *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*. 2012, pp. 45–56.

20. AGRAWAL, Rakesh, SRIKANT, Ramakrishnan, et al. Fast algorithms for mining association rules. In: *Proc. 20th int. conf. very large data bases, VLDB*. 1994, vol. 1215, pp. 487–499.

21. KARIMI, Leila, ALDAIRI, Maryam, JOSHI, James and ABDELHAKIM, Mai. An Automatic Attribute Based Access Control Policy Extraction from Access Logs. *IEEE Transactions on Dependable and Secure Computing*. 2021, pp. 1–1. Available from DOI: 10.1109/TDSC.2021.3054331.

22. BECKERLE, Matthias and MARTUCCI, Leonardo A. Formal definitions for usable access control rule sets from goals to metrics. In: *Proceedings of the Ninth Symposium on Usable Privacy and Security*. 2013, pp. 1–11.

23. MOLLOY, Ian, CHEN, Hong, LI, Tiancheng, WANG, Qihua, LI, Ninghui, BERTINO, Elisa, CALO, Seraphin and LOBO, Jorge. Mining roles with multiple objectives. *ACM Transactions on Information and System Security (TISSEC)*. 2010, vol. 13, no. 4, pp. 1–35.

24. OLSON, David L and DELEN, Dursun. *Advanced data mining techniques*. Springer Science & Business Media, 2008.

Appendix

A Electronic medium structure

`/thesis.pdf` (Thesis document)

`/PolicyMiningMedusa` (F# project)

`/src` (The source code directory)

`/Fei.Backend.Main` (The project for policy mining program)

`/Fei.Backend.Tests` (Unit tests for `/Fei.Backend.Main`)

`/Fei.ConstableConfig.Main` (The project for config generation)

`/Fei.ConstableConfig.Tests` (Unit tests for `/Fei.ConstableConfig.Main`)

`/Fei.Backend.MetricCollection` (The project for collection of metrics)

B Technical guide

This technical guide should provide information about the requirements for running the F# project and also how to run the project.

B.1 Prerequisites

The project requires the installation of .NET core of version 6.0.100 or higher. After the .NET core is installed, the correct version can be verified with the following command.

Check version command

```
Command | dotnet --list-sdks
```

B.2 Program: Policy mining

The source code for policy mining can be found in the directory `Fei.Backend.Main`. It is required to have the current working directory set to directory `src`. The program requires two input arguments – the log file, from which the access requests are being taken and the output file, to which the mined policy is saved. Both files can be supplied as both relative or absolute paths. We prepared some input files in a directory `src/Fei.Backend.Main/Resources`.

Run program: Policy mining

```
Command | dotnet run --project ./Fei.Backend.Main
        | <path_to_log_file> <path_to_output_file>
```

B.3 Program: Configuration file generation

The source code for generation of Constable configuration can be found in the directory `Fei.ConstableConfig.Main`. Similarly to running the policy mining program, it is required to have the current working directory set to directory `src`. The program requires two input arguments – the file with the mined policy (output from policy mining program) and output file, to which the Constable configuration is saved. The input file can be generated by running the policy mining program first.

Run program: Configuration file generation

Command	`dotnet run --project ./Fei.ConstableConfig.Main <path_to_policy_file> <path_to_output_file>`

C Full grammar table

Table C.1: Defined grammar for derivation of Constable configuration

Input		Output
[start]	→	[file_tree] [newl] [process_tree] [newl] [vs_defs] [newl] [rules] [newl] [functions]
[file_tree]	→	[file_tree_def] [newl] [is_primary_tree]
[file_tree_def]	→	tree "«id»" clone of file by getfile getfile.filename;
[is_primary_tree]	→	primary tree "«id»"; | ϵ
[process_tree]	→	[process_tree_def]
[process_tree_def]	→	tree "«id»" of process;
[vs_defs]	→	[vs_fs_defs] [newl] [vs_proc_defs] [newl] [vs_proc_inits]
[vs_fs_defs]	→	[vs_fs_def] [newl] [vs_fs_defs] | ϵ
[vs_fs_def]	→	space «id» = [path] [add_paths]
[path]	→	recursive «path_value» | «path_value»
[add_paths]	→	[op] [path] [add_paths] | ϵ
[op]	→	+ | -
[vs_proc_defs]	→	[vs_proc_def] [newl] [vs_proc_defs]
[vs_proc_def]	→	space «uid» = «domainName»/«uid»;
[vs_proc_inits]	→	[vs_proc_init] [newl] [vs_proc_inits] | ϵ
[vs_proc_init]	→	«id» \t ENTER «id», READ «id», WRITE «id», SEE «id»;
[rules]	→	[rule] [newl] [rules] | ϵ
[rule]	→	«id» \t [ability] [add_ability]
[ability]	→	READ «id» | WRITE «id» | SEE «id»
[add_ability]	→	, [ability] [add_ability] | ;
[functions]	→	[handlers] [newl] [enter_domain] [newl] [get_process] [newl] [constable_init_func] [newl] [init_func] [newl]
[enter_domain]	→	function enter_domain {\n\t enter(process,str2path("«domainName»/"+$1));\n}
[get_process]	→	* getprocess [left_bracket] [newl] [enter_space] [return_ok] [newl] [right_bracket]

[left_bracket]	→	{
[right_bracket]	→	}
[enter_space]	→	[branching] | [default_enter_space]
[default_enter_space]	→	enter(process,@"«domainName»/init");
[branching]	→	if [expression] [branch_body] [other_branch]
[expression]	→	(process.cmdline == "«proctitle»")
[branch_body]	→	[left_bracket] [newl] [enter_domain_call] [newl] [right_bracket] [newl]
[enter_domain_body]	→	enter_domain(process,@"«domainName»/«uid»");
[other_branch]	→	elseif [expression] [branch_body] [other_branch]
[other_branch]	→	else [expression] [left_bracket] [newl] [default_enter_space] [newl] [right_bracket] [newl]
[return_ok]	→	return OK;
[newl]	→	\n

D Example Constable configuration

```
// autogenerated configuration file
// any manual changes are lost when the file is pregenerated
tree "fs" clone of file by getfile getfile.filename;
primary tree "fs";
tree "domains" of process;

// virtual space definitions
space var_ws = /var;
space usr_ws = /usr;
space var_with_children_rws = recursive /var/tmp
              + /var/tmp
              + recursive /var/lib/mysql
              + recursive /var/log/mariadb;
space etc_with_children_rws = /etc/passwd
              + /etc/services
              + /etc/nsswitch.conf
              + /etc/localtime
              + /etc/my.cnf.d
              + /etc/my.cnf
              + /etc/ld.so.cache
              + /etc/my.cnf.d/spider.cnf
              + /etc/my.cnf.d/mysql-clients.cnf
              + /etc/my.cnf.d/mariadb-server.cnf
              + /etc/my.cnf.d/cracklib_password_check.cnf
              + /etc/my.cnf.d/client.cnf
              + /etc/my.cnf.d/auth_gssapi.cnf;
space dev_with_children_rws = /dev/null
              + /sys/devices/system/cpu/online;
space lib64_some_rws = /lib64/libnss_sss.so.2
              + /lib64/libgpg-error.so.0
              + /lib64/libgcc_s.so.1
              + /lib64/libgcrypt.so.20
              + /lib64/libcap.so.2
              + /lib64/libzstd.so.1
              + /lib64/libc.so.6
              + /lib64/libm.so.6
              + /lib64/libstdc
              + /lib64/libcrypto.so.1.1
              + /lib64/libssl.so.1.1
              + /lib64/libz.so.1
              + /lib64/libaio.so.1
              + /lib64/libsystemd.so.0
              + /lib64/liblzma.so.5
              + /lib64/liblz4.so.1
              + /lib64/libcrypt.so.2
              + /lib64/libpcre2-8.so.0;
space sys_with_children_rws = /sys/block
              + /sys/block/zram0/dev
              + /sys/block/zram0/queue/rotational
```

```
              + /proc/sys/crypto/fips_enabled
              + /sys/block/sda/queue/rotational
              + /sys/block/dm-0/queue/rotational;
space var_lib_with_children_ws = /var/lib
              + recursive /var/lib/mysql
              + /var/lib/mysql;
space run_mariadb_ws = recursive /run/mariadb;
space usr_share_mariadb_charsets_Index.xml_rws = /usr/share/mariadb/charsets/Index.xml;

space mysql = domains/mysql;

mysql  ENTER mysql, READ mysql, WRITE mysql, SEE mysql;


// rule definitions
mysql  READ usr_share_mariadb_charsets_Index.xml_rws, WRITE usr_share_mariadb_charsets_Index.xml_rws, SEE
    usr_share_mariadb_charsets_Index.xml_rws;
mysql  WRITE run_mariadb_ws, SEE run_mariadb_ws;
mysql  WRITE var_lib_with_children_ws, SEE var_lib_with_children_ws;
mysql  READ sys_with_children_rws, WRITE sys_with_children_rws, SEE sys_with_children_rws;
mysql  READ lib64_some_rws, WRITE lib64_some_rws, SEE lib64_some_rws;
mysql  READ dev_with_children_rws, WRITE dev_with_children_rws, SEE dev_with_children_rws;
mysql  READ etc_with_children_rws, WRITE etc_with_children_rws, SEE etc_with_children_rws;
mysql  READ var_with_children_rws, WRITE var_with_children_rws, SEE var_with_children_rws;
mysql  WRITE usr_ws, SEE usr_ws;
mysql  WRITE var_ws, SEE var_ws;


function init
{}
function constable_init
{}
function enter_domain {
        enter(process,str2path("domains/"+$1));
}
* getProcess *
{
       if (proctitle.cmdline == "/usr/libexec/mariadbd") {
              enter(process,@"domains/mysql");
       } else {
              enter(process,@"domains/init");
       }
       return OK;
}
```

Listing D.1: Example generated Constable configuration