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

Ing. Roderik Ploszek

Dissertation Thesis Abstract

Operating Systems Security

to obtain the Academic Title of *philosophiae doctor*,
abbreviated as *PhD.*

in the doctorate degree study programme	Applied Informatics
in the field of study	Computer Science
form of study	full-time

Bratislava, 2023

Slovak University of Technology in Bratislava
Faculty of Electrical Engineering and Information Technology

Ing. Roderik Ploszek

Dissertation Thesis Abstract

Operating Systems Security

to obtain the Academic Title of *philosophiae doctor*,
abbreviated as *PhD.*

in the doctorate degree study programme	Applied Informatics
in the field of study	Computer Science
form of study	full-time

Bratislava, 2023

Dissertation thesis has been prepared at Institute of Computer Science and Mathematics, Faculty of Electrical Engineering and Information Technology, Slovak University of Technology in Bratislava (FEI STU)

Submitter: Ing. Roderik Ploszek
FEI STU
Ilkovičova 3, 812 19 Bratislava 1

Supervisor: doc. Ing. Milan Vojvoda, PhD.
FEI STU
Ilkovičova 3, 812 19 Bratislava 1

Consultant: Mgr. Ing. Matúš Jókay, PhD.
FEI STU
Ilkovičova 3, 812 19 Bratislava 1

Readers .....

.....

.....

.....

.....

.....

.....

.....

Dissertation thesis abstract was sent out on .....

Dissertation thesis defence will be held on ..... at .....

at FEI STU, Ilkovičova 3, 812 19 Bratislava 1, in room C-502.

prof. Ing. Vladimír Kutiš, PhD.
dean of FEI STU

Contents

Introduction

This dissertation deals with the security of operating systems. Specifically, it concerns mandatory access control (MAC) in the Linux operating system, which is implemented using the Linux security modules (LSM) interface. This interface allows modules to manage security in the system, in such a way that it monitors all security-related operations and, depending on the subject and the object of the operation, decides whether the operation is allowed according to the current security policy.

The research problem we are dealing with in our work is the automatic creation of security policy. A user of a Linux operating system will mostly receive this policy from the developers of the distribution they are using. This is how it works for popular distributions such as Fedora, Ubuntu or even mobile operating systems such as Android. It is not standard for the average user to create their own security policy.

The situation is same with Medusa, a security module developed at FEI STU. For proper functioning system, the user must create his own configuration. In order to do this, the user must have a perfect overview of which services and applications are running on his system and which authorizations are required for their correct operation. Since this is not an easy task, there is an incentive to design a way to automate this activity as much as possible, with minimal user intervention.

Therefore, the research goal of this work is to design algorithms that can determine the security policy, or the set of objects which the application is authorized to access. To do this, it is necessary to analyze the behavior of applications on the Linux

system and find out the properties from which information about the security policy can be derived. There are several sources of this information: we are mainly interested in every operation that the program performs and the files in the filesystem that are associated with the given program.

The scope of our work is limited only to a dynamic analysis of programs, where we focus on file operations. The resulting policy is applicable to the Constable authorization server. The main contribution of the work is a set of algorithms for security policy mining for the Medusa security module and the implementation of these algorithms in a Python application.

1 Research objectives

The main objective of this thesis is to design and implement an algorithm that creates a functional security policy for the Medusa security module with minimal administrative intervention. This is a general objective that might be too overwhelming to achieve. To make it more focused and specific, we introduce these constraints:

1. Security policy will only take into account filesystem operations and accesses.

2. Creation of security policy will focus on limiting the set of available objects to a user or system application. This concept is similar to sandboxing. By limiting access of the application, we limit the *attack surface* of the application. Our focus is on creating a policy for system services as

opposed to creating a policy for individual users that use the system.

3. Created policy is static. This means that if the administrator wants to later update the policy, she has to run the policy mining algorithm again.

The expected output of the thesis can be summarized in the following research questions:

1. If we construct an algorithm that would create security policy just from operation logs, how would it compare to an administrator-authored policy?

2. If we construct an algorithm that would create security policy from operation logs including some external information, how would it compare to an administrator-authored policy?

2 Solution design and methods used

Our proposed solution takes inspiration from domain-type enforcement [1, 2], specifically it creates domains for each executed application. By observing activities of the application (on the premise that the application is not malicious), we can construct a list of objects that the application should have access to.

Brief summary of the proposed solution is:

1. Monitor operation of an application for which the security policy will be created.

2. Preliminary policy will be created for each subject[1] (represented by an execution domain) based on the name of the object[2] and requested operation.

3. Preliminary policy will be analyzed for missing rules that create underpermissions and additional rules will be added to the policy. We call this step *generalization*. After this step, a full usable policy for an application should be available.

2.1 Getting Audit Logs

Ideal way of getting logs on the Linux operating system is the audit system. It is able to log system calls and various security-related events in the operating system based on the settings provided by the `auditd` deamon.

Using contributions from [3], we modified Medusa security module to audit every hooked operation of a chosen process. Process to audit can be selected by a `fexec` handler in the authorization server configuration. Once Medusa-specific auditing is enabled for a thread, each hook call will create an audit record containing information about the thread, the operation, the object of the operation and any other useful information provided through the hook interface.

[1] Entity that executes operations on some object, e.g, a process.
[2] Entity on which the operation is executed, e.g., a file.

2.2 Generalization

From the nature of the audit logs, we can identify the following problems that cause underpermission[3]:

1. Based on the execution of the application, not all execution paths may have been executed and thus some accesses may not have manifested. These accesses will be denied once the policy will be enforced.

2. Accesses to temporary files or newly created files will refer to paths that were not captured in the original audit logs. Note that compared to the previous point, the access was requested, but the path is different in the next execution. The consequence is the same — after the policy is enforced, these accesses will be denied.

Solution to this problem is generalization — the policy mining module has to relax the generated rules so they will match a larger set of possible paths. This causes overpermission, which is undesirable. Policy mining has to solve an optimization problem — keep overpermission low while causing as few access misses as possible.

Generalization creates rules that apply to multiple paths which may not be present in the filesystem. This can be achieved using regexp and/or recursive rules. As an example, take rule ("`/var/log/pgsql/`$.*$", do, P) which allows processes under domain do to execute operations that require permissions P on any file under `/var/log/pgsql` directory.

[3] Denied operation that should be allowed.

In the following subsections we briefly present the proposed algorithms. Their full description can be found in the thesis.

2.2.1 Tree coverage generalization

In this generalization, we assume that if all files in a folder have the same access permission, we can generalize this access permission for the entire content of the folder.

This generalization takes into account only information from the audit logs. This means that if a process didn't access some path that exists in the filesystem, the generalization algorithm assumes the path doesn't exist.

We expect that this generalization will work well for services that store their files together in folders and they access all or most of these files. It won't work well for services that have many files in different folders and access them only sporadically.

2.2.2 Filesystem Hierarchy Standard generalization

This generalization takes into account standard hierarchy of folders in Linux systems as defined in File Hierarchy Standard [4] (FHS) and systemd's file-hierarchy [5] and Linux's hier(7) [6] manual pages.

For example, folder `/proc` contains information about running processes. This information is available under numerical subdirectories for each running process in the form of `/proc/<pid>`, where `<pid>` is the PID of the process. The role of this generalization algorithm is to add rules for accessing these folders. Similar reasoning can be used for system-wide files, such as libraries in `/usr/lib64` or binaries in `/usr/bin` that should automatically get *read* permission for all processes.

2.2.3 Generalization based on non-existing files

This generalization is based on two path sets. Path set PA' is created from the real filesystem before the service(s) for which the policy is mined are started. Path set PA is derived from access set A created from the audit log.

Algorithm for this generalization takes every path from PA' that is not present in PA, computes its parent directory and applies read and write permission to it.[4]

This generalization method is just supplementary with a specific focus on non-existent files, it cannot provide general generalization. Therefore it is expected that it might improve performance of other generalization algorithms, such as the tree coverage generalization, when used together.

2.2.4 Generalization based on UGO permissions

This generalization relies on external information stored on the filesystem. Most of the services on the Linux system are assigned a special user ID. These IDs are used as eUIDs when a service is running, but also as owner and group owner IDs of files associated with the service. We can use this information when constructing the policy. We propose these generalization strategies that can be used independently:

1. **Generalization by directory owner UID** Access to directories that are owned by eUID of the domain can

[4] *Write* permission to the parent directory is applied automatically after loading to audit log, since to create a file in the directory, process has to have a write permission to that directory.

be generalized so that the domain gets privilege to access (read/write) any file in those directories.

Rationale for this strategy is that it *approximates* standard UGO permissions. Namely, if a user owns a directory, he can access all files in this directory as well with high certainty.

2. **Generalization by file UID** This strategy is similar to the previous one, with the difference that it considers files *inside* a directory. For an access to any file in a directory to be generalized, all accessed files have to be owned by the effective user of the running domain.

3. **Generalization by read access to files** If the directory contains items that are readable by the effective user of the process, read access to files in this directory can be generalized. This considers computation of DAC permissions according to UGO permissions of each file (see section 2.1.1 in the thesis). There is an equivalent method that generalizes write access.

2.2.5 Generalization based on owner directory

Unlike *generalization by directory UID*, this generalization relies just on external information and not on information from audit logs. Generalization algorithm searches for folders in the filesystem that match UIDs or GIDs of the generalized service. Files inside these folders are then generalized for read and write access. We presume that this generalization will achieve the best results, since it relies entirely on external information.

2.2.6 Generalization based on multiple runs

This generalization method takes advantage of the fact that temporary files change names across executions of a service. Thus we can start a service multiple times, get audit log information from each *run* and compare them. Paths that are unique across all runs can be considered to be ephemeral and can be generalized accordingly.

Interesting problem is the generalization of the paths. They usually consist of static and dynamic parts. The problem lies in identifying these dynamic parts and providing generalization of them. We present a solution to this problem using cosine similarity of TF-IDF N-gram vectors [7].

3 Achieved results of the dissertation

This section presents the evaluation of suggested algorithms. First, we introduce the methodology used to test our algorithms. Then, we present results from two tests: generating policy for individual services and generating policy for multiple services using the tree generalization algorithm (cumulative mining). For the individual services, we have evaluated four applications that are standard components of a Linux server system. For the cumulative mining, we have evaluated three services in different combinations.

3.1 Methodology

Testing was performed on a Fedora Server 37 distribution with Medusa running on a 6.2 Linux kernel. Testing consisted of following operations:

1. System is booted with the Fedora kernel.
2. Filesystem snapshot is created before running any services, this results in a set of paths PA'. This set will be used when evaluating owner, owner directory and non-existent generalization algorithms.
3. System is rebooted with the Medusa kernel.
4. Constable configuration for a specific service is prepared and Constable is started (see `run_service.sh` script).
5. Service is started, it is left running for a few seconds and then stopped.
6. Constable is stopped and audit log is retrieved for analysis.
7. Steps 2.–4. are repeated to get an alternative audit log that will be used for multiple runs generalization.
8. System is rebooted with the Fedora kernel.
9. Policy mining is executed with specific test cases. These test cases are described in the following subsections.

Resulting mined policy is compared to the reference SELinux policy present in Fedora. This is done by comparing a specific set of paths (containing paths of files and directories) for two access types supported by both Medusa and SELinux: read and write.

Resulting permission values are evaluated using standard binary classification techniques. Results can be classified into 4 categories:

hit (TP) Both mined policy and reference policy allow the operation.

overpermission (FP) Mined policy allows the operation while reference policy denies it.

underpermission (FN) Mined policy denies the operation while reference policy allows it.

correct denial Both mined policy and reference policy allow the operation.

Because of the evaluation methodology, absolute values of these four categories for each individual service can't be compared directly and a relative metric is needed. When searching for suitable metrics, we discarded metrics that determined the relative value from correct denials. This is because of the permissive nature of LSM — accesses that are not listed in the policy are automatically denied. Since we could put accesses to all the other files on the filesystem into correct denials and thus artificially inflate the metric, it is not usable for our purpose. Two basic metrics that are suitable to compare mined and reference policies are sensitivity (equation 2) and precision (equation 1). Note that sensitivity is more important since it represents underpermission — accesses that are not permitted cause denial of service. Overpermission, while undesirable, can be tolerated since it doesn't cause the program to stop functioning.

For a combined metric, we have chosen F_β (equation 3), specifically F_2. F_1 is a harmonic mean of precision and sensitivity. By using value of $\beta = 2$ we weigh sensitivity twice more than

precision. This well expresses our intention to have sensitivity more important than accuracy.

$$PPV = \frac{TP}{TP + FP} \tag{1}$$

$$SEN = \frac{TP}{TP + FN} \tag{2}$$

$$F_\beta = (1 + \beta^2) \cdot \frac{PPV \cdot SEN}{\beta^2 \cdot PPV + SEN} \tag{3}$$

Table 1 shows short names that are used in evaluation tables. Combinations of generalizations are represented by a plus sign in the order the generalizations were applied.

Table 1: Legend of generalization names used in evaluation tables

Generalization	Short name
Tree coverage	T
Owner	O
Owner directory	OD
Nonexistent	N
Multiple runs	M

3.2 Individual mining

This subsection presents result from evaluating single services.

3.2.1 PostgreSQL

Results for PostgreSQL mining are available in table 2. Generalization with lowest number of underpermission accesses (5) was combination OD+T. This is also the combination of generalizations with the best sensitivity. However, as it had more overpermissions (597), the best generalization according to the F_2 metric was OD (145 overpermissions, 17 underpermissions). This generalization was so effective because PostgreSQL contains a large number of files under `/var/lib/pgsql` that represent the database. The audit log covered only some of them and OD generalization was able to cover all except for a small anomaly in subfolders of `/usr/share/pgsql/timezonesets`, which is owned by root and not postgresql. This was improved by combining OD and T generalizations (5 underpermissions), but at the cost of increased overpermission (597).

3.2.2 OpenSSH SSH Daemon

Results for OpenSSH SSH daemon mining are presented in table 3. In this case, only T and O generalizations had any effect on the generated policy. Other generalization algorithms didn't provide any improvement over policy with no generalization.

Tree coverage generalization had 11 underpermission accesses, mostly files related to the `/proc` filesystem. Owner generalization fixed 9 underpermission accesses compared to no generalization, with the total number of underpermissions of 362. However, most of these accesses were in `/usr/sbin` directory and it is assumed that after manual review these underpermissions can be ignored.

Table 2: Results of policy mining for PostgreSQL

Generalization	SEN	PPV	F_2
no gen.	0.7673	0.9778	0.8018
T	0.7770	0.9157	0.8013
O/M+O	0.8775	0.9806	0.8963
OD/OD+O/M+OD	0.9980	**0.9829**	**0.9949**
N/M+N	0.7704	0.9767	0.8044
M	0.7677	0.9779	0.8021
M+T	0.7774	0.9157	0.8016
N+T	0.7802	0.9160	0.8041
O+T	0.8856	0.9252	0.8933
OD+T	**0.9994**	0.9332	0.9854
O+N	0.8775	0.9795	0.8961
OD+N	0.9980	0.9820	0.9947

Table 3: Results of policy mining for OpenSSH

Generalization	SEN	PPV	F_2
no gen.	0.9209	**0.9769**	0.9316
T	**0.9977**	0.8902	**0.9741**
O	0.9228	0.9586	0.9297

3.2.3 Postfix

Results for the Postfix mail transfer agent policy mining are presented in table 3. The lowest number of overpermissions (662) was in the policy without generalization. Every other generalization algorithm increased the number of overpermission accesses, as expected. Non-existent generalization didn't provide any effect when used on it's own and also in most of the pairs. There is one interesting exception with combination of M+N, that achieved 882 underpermission accesses with F_2 metric considering this to be the best method for this service. However, it must be mentioned that this pair had the worst result in overpermission with 1655 accesses.

The most interesting thing about M+N method is that M nor N on its own couldn't provide such good results and this means that interactions between these two generalization algorithms produced this result. The underpermissions were mostly located in `/usr/libexec`.

3.2.4 Apache

For the web server Apache only algorithm capable of generalizing policy was the T algorithm with 336 overpermissions and 258 underpermission accesses. The T algorithm achieved F_2 metric of 0.95. Without generalization, the overpermission was just 85 accesses with F_2 score of 0.81.

Apache does not use a lot of owned files or temporary files, so the other generalization algorithms could not manifest themselves in the resulting policy.

Table 4: Results of policy mining for Postfix

Generalization	SEN	PPV	F_2
no gen./N	0.8484	**0.9509**	0.8671
T/N+T/OD+T	0.8886	0.9268	0.8960
O/O+N	0.8541	0.9487	0.8715
OD/OD+N	0.8484	0.9500	0.8669
M/M+OD	0.8510	0.9487	0.8689
M+T	0.8912	0.9256	0.8979
O+T	0.8943	0.9249	0.9003
OD+O	0.8541	0.9478	0.8714
M+O	0.8549	0.9464	0.8718
M+N	**0.9456**	0.8961	**0.9352**

Table 5: Results of policy mining for Apache

Generalization	SEN	PPV	F_2
no gen. and others	0.8090	**0.9800**	0.8382
T	**0.9500**	0.9358	**0.9471**

3.2.5 Individual mining summary

We can see that the policy mining results for individual services depended on the evaluated service. The best algorithm for generalization came out based on which files the service accessed and how the files are distributed on the disk, whether they have metadata, such as owners.

See table 6 for the summary of the expriments with the best algorithm for each metric. The best methods according to F_2 metric were tree coverage and owner directory. In one service, combination of multiple runs and non-existent files proved to be the best. On the contrary, other generalization methods did not show better results.

Table 6: Summary of policy mining for individual services

Service	Best SEN	Best PPV	Best F_2
PostgreSQL	OD+T	OD	OD
OpenSSH	T	no gen.	T
Postfix	M+N	no gen.	M+N
Apache	T	no gen.	T

3.3 Cumulative mining

This subsection contains evaluation of cumulative mining, meaning evaluating how the policy changes as more services are added to the mining algorithm. This evaluation is specifically intended for the tree algorithm, as its generalization is based on the coverage of the tree. The more services are used in the algorithm,

more files and directories from the real filesystem are available for the algorithm to work with. Our hypothesis is that more services we use for the mining, the precision should go up. We will test this with services from the previous evaluation: PostgreSQL, OpenSSH, Postfix and Apache HTTP server.

Results of the cumulative mining are presented in table 7.[5] This small example meets our hypothesis. By adding one or two service logs to the tree coverage algorithm, the resulting precision increases. However, this doesn't mean that any other combination will also show a similar pattern. We can prove this by adding the Apache service log, when precision drops to 0.972.

Table 7: Results of cumulative mining

Services	PPV
s	0.891
postg	0.916
postf	0.936
postg + s	0.965
postg + postf	0.968
s + postf	0.941
postg + s + postf	0.974
postg + s + postf + a	0.972

[5]Abbrevations used in the table — s: OpenSSH, postg: PostgreSQL, postf: Postfix, a: Apache.

3.4 Conclusion

Sensitivity results for the *best* algorithms ranged from 95.6% to 99.9%. These results show the good ability of our algorithm to cover the program accesses that should be allowed according to the priciple of least privilege. However, for the correct functionality of the program, the sensitivity must be 100%, and thus even after using our algorithm, manual intervention and correction of the security policy will be necessary. This correction should be simplified by the fact that most of the rules will be created automatically. The precision of our algorithm in experiments ranged from 93.4% to 98.3%.[6]

4 List of author's publications

Projects

- Preparation of study materials for project "Informatika ako nástroj rozvoja znalostnej ekonomiky" 312011G208 (principal investigator Ing. Fedor Lehocki PhD. MPH) funded by European social fund, 2018–2020.

- Researcher on "Ontologická reprezentácia pre bezpečnosť informačných systémov" APVV-19-0220 (principal investigator prof. Ing. Pavol Zajac, PhD.) funded by Slovak Research and Development Agency, 2020–2024.

- Researcher on "Postkvantová kryptografia odolná voči pos-

[6] As in multiple cases the best precision was achieved by using no generalization at all, we are listing the second best precision of a generalization algorithm.

tranným kanálom" VEGA 1/0105/23 (principal investigator prof. Ing. Pavol Zajac, PhD.) funded by Slovak Research and Development Agency, 2023–2026.

Reviews

- 1 paper in Tatra Mountains Mathematical Publications 73 (2019)
- 1 paper in 22nd Central European Conference on Cryptography (2022)

Conferences

- PLOSZEK, Roderik. Linux security modules overview. In *ELITECH´18 [electronic source] : 20th Conference of doctoral students. Bratislava, Slovakia. May 23, 2018.* 1st ed. Bratislava : Vydavateľstvo Spektrum STU, 2018, CD-ROM, [7] p. ISBN 978-80-227-4794-3.
- PLOSZEK, Roderik. Upgrading complex single-threaded application to support concurrency. In *ELITECH´19 [electronic source] : 21st Conference of doctoral students. Bratislava, Slovakia. May 29, 2019.* 1st ed. Bratislava : Vydavateľstvo Spektrum STU, 2019, CD-ROM, [8] p. ISBN 978-80-227-4915-2.
- PLOSZEK, Roderik. Using self-organizing maps for security module configuration. In *ELITECH´20 [electronic source] : 22nd Conference of doctoral students. Bratislava, Slovakia. May 27, 2020.* 1st ed. Bratislava : Vydavateľstvo Spektrum STU, 2020, [6] p. ISBN 978-80-227-5001-1.

- ŠVEC, Peter - PLOSZEK, Roderik. A review of encryption schemes used in modern ransomware. In *CECC 2020 : Book of abstracts : 20th Central European conference on cryptology. Zagreb, Croatia. June 24-26, 2020.* Zagreb : University of Zagreb, 2020, p. 50-51.

- PLOSZEK, Roderik - JÓKAY, Matúš. A look into security policy mining. In *Application of Knowledge Methods in Information Security : Bratislava, Slovakia. September 18, 2021.* 1st ed. Bratislava : SRDA, 2021, [2] p. ISBN 978-80-970145-2.

- PLOSZEK, Roderik. Inductive logic programming and description logics. In *Application of Knowledge Methods in Information Security : Smolenice, Slovakia. June 27-29, 2022.* 1st vyd. Bratislava : SRDA, 2022, [1] p. ISBN 978-80-974468-0-2.

Papers in journals

- PLOSZEK, Roderik - ŠVEC, Peter - DEBNÁR, Patrik. Analysis of encryption schemes in modern ransomware. In *RAD Hrvatske akademije znanosti i umjetnosti : Matematičke znanosti, Vol 25, No. 546.* Zagreb : Hrvatska akademija znanosti i umjetnosti, 2021, S. 1-13. ISSN 1845-4100.

- BALOGH, Štefan - GALLO, Ondrej - PLOSZEK, Roderik - ŠPAČEK, Peter - ZAJAC, Pavol. IoT security challenges: Cloud and blockchain, postquantum cryptography, and

evolutionary techniques. In *Electronics*. Vol. 10, iss. 21 (2021), Art. no. 2647 [22] p. ISSN 2079-9292. DOI: 10.3390/electronics10212647.

- ČUŘÍK, Peter - PLOSZEK, Roderik - ZAJAC, Pavol. Practical use of secret sharing for enhancing privacy in clouds. In *Electronics*. Vol. 11, iss. 17 (2022), Art. no. 2758 [18] p. ISSN 2079-9292. DOI: 10.3390/electronics11172758.

Editor of the proceedings

- NEMOGA, Karol (comp.) - PLOSZEK, Roderik (comp.) - ZAJAC, Pavol (comp.). *CECC 2022 : 22nd Central European Conference on Cryptology. Smolenice, Slovakia. June 26 - 29, 2022*. Bratislava : SAS, 2022. 112 p. ISBN 978-80-968374-6-5.

Seminars

- Introduction to Spectre Vulnerabilities, CRYPTO seminar, 21.11.2018

Pedagogical Work

- Selected lectures on Operating Systems in 2019
- Operating Systems seminars in 2016–2022
- Operating Systems seminars for mobility students (Erasmus) in 2018–2022
- Computer Criminality seminars in 2018–2022

- Consultant on 7 master theses, with four expected to finish in 2023 and advisor of 15 bachelor theses, four of which are expected to finish in 2023

5 Rezumé

Cieľom tejto dizertačnej práce bolo navrhnúť a implementovať algoritmy, ktoré automaticky generujú bezpečnostnú politiku pre bezpečnostný modul Medusa. Tento cieľ sa nám podarilo splniť a výsledkom je hotový produkt v podobe aplikácie, ktorú môže používateľ použiť na automatickú konfiguráciu autorizačného servera Constable. Výsledná politika je vytvorená zo záznamov operácií bežiacej aplikácie, napríklad systémovej služby. Riešenie je schopné vytvoriť bezpečnostnú politiku pre niekoľko aplikácií naraz.

Výslednú implementáciu sme porovnali so štandardnou referenčnou politikou bezpečnostného modulu SELinux v distribúcii Fedora 37. Implementáciu sme porovnávali na štyroch bežných službách: PostgreSQL, Open SSH server, Postfix a Apache HTTP server. Výsledky citlivosti pre najlepšie algoritmy sa pohybovali od 95,6 % do 99,9 %. Tieto výsledky preukazujú dobrú schopnosť nášho algoritmu pokryť prístupy programov, ktoré by mali byť povolené podľa *princípu najmenších privilégií*. Pre správnu funkčnosť programu však musí byť citlivosť 100 %, a preto aj po použití nášho algoritmu bude potrebný manuálny zásah a korekcia bezpečnostnej politiky administrátorom. Táto korekcia ale bude zjednodušená tým, že veľká časť pravidiel sa bude vytvárať automaticky. Presnosť nášho algoritmu sa pri experimentoch

pohybovala od 93,4 % do 98,3 %.[7] Zníženie presnosti spôsobili generalizačné algoritmy, ktoré do politiky pridávajú cesty, ktoré sa nevyskytovali v záznamoch operácií, z ktorých bola politika vytvorená.

Je potrebné uznať, že náš výskum bol v niektorých ohľadoch obmedzený. Vyhodnotenie našich algoritmov záviselo od manuálneho nastavenia generalizačného algoritmu pre *File Hierarchy Standard*. Naše výsledky tiež nemožno považovať za úplné, pretože sme našu aplikáciu testovali len na štyroch systémových službách.

Budúci výskum sa môže zamerať na nové algoritmy, ktoré lepšie analyzujú požiadavky aplikácie a vylepšujú generovanú bezpečnostnú politiku tak, aby bola v súlade s princípom najmenších privilégií. Ďalším smerom, ktorým sa práca môže uberať, je statická analýza aplikácií, ktorou sme sa nezaoberali. Iným prínosom môže byť analýza väčšieho počtu služieb a hľadanie vzájomných súvislostí medzi nimi, napríklad pomocou strojového učenia, ontológií alebo induktívneho logického programovania.

Hlavným prínosom tejto práce je hotová aplikácia, ktorú možno použiť na konfiguráciu bezpečnostného modulu Medusa. Jej modulárna konštrukcia umožňuje pridávať ďalšie generalizačné algoritmy, čo otvára priestor skúmaniu ďalších metód na zlepšenie vlastností výslednej generovanej bezpečnostnej politiky. Po určitých úpravách by sa dala použiť aj na generovanie politík pre iné bezpečnostné moduly.

[7] Keďže v niekoľkých testoch mal najlepšiu presnosť prípad bez generalizácie, uvádzame druhú najlepšiu presnosť generalizačného algoritmu.

Bibliography

1. WALKER, Kenneth M., STERNE, Daniel F., BADGER, M. Lee, PETKAC, Michael J., SHERMAN, David L. and OOSTENDORP, Karen A. Confining Root Programs with Domain and Type Enforcement. In: *6th USENIX Security Symposium (USENIX Security 96)*. San Jose, CA: USENIX Association, 1996. Available also from: `https://www.usenix.org/conference/6th-usenix-security-symposium/confining-root-programs-domain-and-type-enforcement`.
2. HALLYN, Serge E. *Domain and Type Enforcement for Linux*. 2003. PhD thesis. The College of William & Mary in Virginia.
3. ŇAŇKO, Peter. *Podpora audit systému pre bezpečnostný model Medusa*. 2020. Available also from: `https://opac.crzp.sk/?fn=detailBiblioForm&sid=201CE56335A527AB040B96791929`. Bc. pr. Ústav informatiky a matematiky, Fakulta elektrotechniky a informatiky Slovenskej technickej univerzity v Bratislave. EČ: FEI-5382-86243.
4. YEOH, Christopher, RUSSELL, Rusty and QUINLAN, Daniel (eds.). *Filesystem Hierarchy Standard* [online]. The Linux Foundation, 2015-03-19 [visited on 2023-04-18]. Available from: `https://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.pdf`.
5. *file-hierarchy — File system hierarchy overview* [online]. 2023. [visited on 2023-04-18]. Available from: `https://www.freedesktop.org/software/systemd/man/file-hierarchy.html`.

6. KERRISK, Michael. *hier(7) — Linux manual page* [online]. 2021. [visited on 2023-04-18]. Available from: `https://man7.org/linux/man-pages/man7/hier.7.html`.

7. BERG, Chris van den. *Super Fast String Matching in Python* [online]. 2017-10-14. [visited on 2023-05-03]. Available from: `https://bergvca.github.io/2017/10/14/super-fast-string-matching.html`.