Intellectual Method of Program Interactions Visualisation in Unix-like Systems for Information Security Purposes

Mikhail Buinevich[a], Konstantin Izrailov[b,c] and Grigory Ganov[b]

[a]Saint-Petersburg University of State Fire Service of EMERCOM of Russia, Moskovskiy prospect 149, Saint-Petersburg, 196105, Russia

[b]The Bonch-Bruevich Saint-Petersburg State University of Telecommunications, 22 Prospekt Bolshevikov, Saint-Petersburg, 193232, Russia

[c]St. Petersburg Federal Research Center of the Russian Academy of Sciences, 14 Line, 39, Saint-Petersburg, 199178, Russia

Abstract

One of the tasks of information security audit is to monitor data exchange processes between the programs of operation system. Expert can't monitoring data exchange manually because of huge amount of data files and dissimilar data exchange between programs and operation system. But full automation of this process is very complicated for implementation due to weak formalization of information about data exchange processes and criteria of their insecurity. In this paper, we propose a partial solution to the issue by visualizing programs interactions for an expert in the form of an appropriate method, consisting of 8 steps; the scheme of the method is attached. In order to implementing the method, the following types of programs are identified: PE, ELF, bytecode and script. Program interactions as direct and emulated calls, direct and emulated imports, indirect exchange are considered as well. The underlying formal model of the method is presented. The expediency of using artificial intelligence as one of the steps of the method is justified. The applicability of such machine learning areas as classification, anomalization, clustering, regression and dimensionality reduction is described. The developed prototype of the tool and its modules are described. The prototype of the tool is being tested on the Unix-like application Termite. The resulting visualization of data files interactions in the attachment is given, including details of individual links that are hypothetically of interest to an information security expert.

Keywords

operating system, software, information security, artificial intelligence, static analysis, audit, interaction, visualization

1. Introduction

Lately software have been becoming the basis for the functioning of most modern devices. As a result, it leads to an increase in the number of information security threats (hereinafter –

Proceedings of the 12th Majorov International Conference on Software Engineering and Computer Systems, December 10–11, 2020, Online Saint Petersburg, Russia

bmv1958@yandex.ru (M. Buinevich); konstantin.izrailov@mail.ru (K. Izrailov); ganov99@rambler.ru (G. Ganov)

0000-0001-8146-0022 (M. Buinevich); 0000-0002-9412-5693 (K. Izrailov); 0000-0001-5236-3805 (G. Ganov)

IS), which also affect the correct performance of aforementioned devices. And if at the dawn of this trend, IS threats problem was solved via the search in individual program for the presence of errors in the code, now the scale and structure of software do not allow us to apply such approach. The reason for this is the following scientific contradiction. On one hand, it is necessary to determine the existence and types of control-flows and data between parts of the software; since the emergence of a new flow may be indicative of unauthorized call (one module calls another module to bypass the standard procedure) or an invasion of privacy (unauthorized data is sent to a module) and the disappearance of the old data — about the malfunction (one module is not able to call alleged module) or the interruption of availability (legitimate data transfer was interrupted). On the other hand, modern “medium” and “large” size software present itself as entire operating systems (hereinafter – OS), where the aforementioned flows are provided by the interaction of programs through the file system (hereinafter – FS). The situation is complicated by the lack of formalization of information for such flows, which is necessary for the involvement of IS experts to manually analyze relationships between programs, identify suspicious locations, and conduct specialized analysis of them. A partial resolution of the contradiction can be obtained by providing a complete visual picture of the OS program interactions (or OS critical parts, for example, the system directory), while being able manually configure visualization process. Also, considering the huge number of files even in the "average" OS, the use of intelligent analysis methods (as an aid to the expert) will increase analysis effectiveness, reduce the cost of analysis time and expert’s psycho-emotional resources with virtually unchanged performance. Thus, the task of intelligent visualization of program interaction is certainly relevant; one of its possible solutions in the form of a corresponding method (hereinafter – the Method) is proposed by the authors of the article. It is important to note that the formulation of the problem assumes static analysis of the FS, which unlike dynamic analysis, does not imply real program calling. Although, dynamic analysis has a number of advantages over static analysis, in case of a large file number, the coverage of all interactions will be ineffective and the static analysis is to be preferred.

2. State of Art

Let’s review the existing scientific works devoted to the static analysis of OS programs and their interconnections, as well as the use of intelligent methods for file analysis.

Scientists in [1] use machine learning (hereinafter — ML) classification methods to separate programs into FS blocks. Article [2] describes the application of the ML clustering method for grouping text files. Publication [3] is devoted to detecting packed Windows programs using the Support Vector Machine. An intelligent way of detecting executable files in other programs is described in [4] using anomaly detection method. The classical classification of files (programs, documents, etc.) using a complex of ML methods is given in [5]. In [6] is described file identification of the Unix OS based on the Chi-square and Kolmogorov-Smirnov criteria. In [7] is provided an algorithm for detecting library functions in Windows programs using signatures. In [8] is proposed to counteract virus software, identification of which is possible based on group file characteristics.

Despite certain work in certain areas related to the current task of the study, not even close

solution was found. Thus, the development of the author's method, in addition to being relevant, certainly has a novelty.

3. Model of the interactions of programs

Since the essence of the method is to visualize interactions between OS programs, it is obvious that its algorithms should work on some representation of such relationships – that is, on the corresponding model (hereinafter – the Model). Therefore, before creating the Method itself, the structure of the Model needed to be defined (the main elements and their relationships). In the interests of this, we will review the main types of programs and data transfer mechanisms between software (limited only to the FS), the analysis of which will allow the expert to audit the OS IS [9].

3.1. Program types

Initially, it was indicated that interactions of OS programs is considered, but not text files or images. This is justified by the fact that any data transfer implies some action, files that provide the transfer must perform certain operations – that is, files must be executable. Based on this, we will define the following types of modern OS programs (directly related to their binary format):

1. PE-program are programs that run in the Windows family of operating systems [10]; the prefix PE (abbr. from Portable Executable) means the structure of header and format of the file itself. Dynamic link libraries files are considered of this type as well.
2. ELF-programs are similar to PE-programs, but run in the Unix-like OS family [11]; the ELF prefix (abbr. from Executable and Linkable Format) means the structure of header and file format. So-called shared objects with functionality of libraries are belong to ELF-programs type, similar to libraries for PE-programs;
3. Bytecode-programs that contain a set of instructions that are executed on a virtual machine; the most well-known are the following: Java-programs that have JBC (abbr. from Java Byte Code) executed by the Java virtual machine [12]; .Net-programs that have CIL bytecode (abbr. from Common Intermediate Language), executed by the .Net virtual machine [13]; Ruby-programs that have bytecode executed by the YARVM (abbr. from Yet Another Ruby Virtual Machine) [14]; Python-programs that have bytecode executed by the corresponding VM [15];
4. Script-programs are source code executed by interpreter without explicitly compiling intermediate assemble code or machine code; for example, command-line scripts for bash (in Unix-like OS) and PowerShell (in Windows OS).

Naturally, there are other types of programs (with their own formats), which are not considered due to obsolescence or rarity of use; for example, LE (addr. from Linear Executable) for running on OS/2, the latest version of which was released in 2001.

Each of these program types can interact with other programs of the same type, as well as with most other program types, by calling them directly. Also it should be noted that programs

can interact via intermediate files. For example, one program can generate a set of structured data (often in JSON or XML format), while another program reads this data.

3.2. Types of interaction

While selecting involved in the interaction file types, possible types of interactions were partially specified. However, let's look at them in more detail:

1. *Direct call*, which means calling one program directly by another: for example, a PE-program can call another PE-program, then run a Java-program and a PowerShell script;
2. *Emulated call*, similar to the Direct call, but with the difference that a program created for another OS is called; for this, a special software environment is used that integrates applications, data and resources of these operating systems; for example, Windows Subsystem for Linux allows you to run ELF-programs from PE-programs, and in wine environment - vice versa.
3. *Direct import*, which means loading an external library by the program: for example, the executable code calls functions from the system library;
4. *Emulated import* is completely analogous to Direct import, but using the Emulated call mechanism – i.e. through a software environment to "provide" execution in another OS family.
5. *Indirect exchange* that provides data exchange between programs through intermediate files: for example, a bash script can generate a configuration file, pass the path to it as an argument when calling a group of ELF- and Java-programs, and then read and analyze the results of this group's work saved in separate files. In this case, the indirect interaction of programs is precisely carried out through these intermediate files.

It should be noted that there are other types of interactions that are not considered due to more complex methods of detection. For example, one program can use the functionality of another, using RPC (abbr. from Remote Procedure Call). Also, in Unix-like systems, processes and sockets can be reflected in the FS in the form of files – i.e. some intermediaries through which other programs can interact.

Based on the importance of file links in the aspect of OS IS, when visualizing interactions, it is advisable to display each of the types in a different way – with its own color, line shape, comment, etc.

3.3. Model structure

Since the Model must display all possible information exchanges between all programs, its logical structure can be presented as system of individual levels for each type of program, within and between which there are all types of relationships. In this case, it is advisable to take into account the fact that PE- and ELF-programs are directly executed (i.e. without emulation) on different operating systems. Generalized Model (since it is a kind of template that is not tied to a specific OS implementation) is shown in Fig. 1.

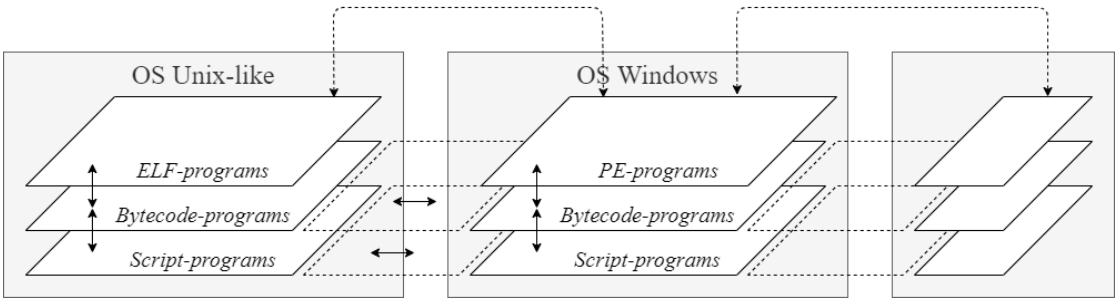

Figure 1: Generalized model of program interactions

Note. Direct Call, Direct Import, and Indirect Exchange are indicated on the Model by a continuous arrow, while Emulated Call and Import are indicated by a dashed arrow. Also, the Model displays the potential for interaction with other operating systems, which have their own individual levels – in the right part of the figure.

3.4. Method of interactions visualization

Based on the generalized Model of program interactions, a visualization Method is proposed that would display such connections in an understandable to an expert form. The intellectual characteristic of the Method will mean the use of special ML methods on it, which simplify the subsequent analysis. The Method consists of the following steps.

Step 1. Preparation. The first step in any multi-step method usually is to perform general preparatory steps. First of all, it should be possible to set a path (or a set of them) to investigated files. It is necessary for the subsequent establishment of relationships and visualization, settings for intelligent algorithms, visualization parameters, etc.

Step 2. Scanning files. Since the examination is carried out within the framework of static analysis, information about the state of processes in the system does not play a big role. Therefore, at this stage, files are collected and processed according to the paths, extensions and other information indicated in the preparatory step. This is done by a trivial recursive traversal of directories and getting their contents.

Step 3. Determining types of programs. At this step, programs types are determined. Although this action is not trivial, there are a number of algorithms for its implementation. First of all, the type of program can be partially determined by the file extension. Second, since PE- and ELF-programs have a header with a specific signature at the beginning of the files, their type can be determined from it; a similar situation will be with Bytecode programs. Thirdly, Script programs that have the form of text can also contain comments at the beginning of the code, which determine their belonging to the programming language. In the general case, the prediction of the file type can be made on the basis of obtaining its frequency-byte characteristics.

Step 4. Determine types of interactions. The step is intended to determine types of interactions that have distinct features for each type of program; These are indicated below.

1. *Direct call*: the machine code of PE- and ELF-programs can contain system calls of functions for launching external programs of the same level; Bytecode-program instructions may contain system calls of functions for launching external programs of any level; the

source text of the Script-Programs may contain commands for launching external programs;

2. *Emulated call*: the machine code of PE- and ELF-programs may contain system calls of functions for launching external programs from the level of another OS family;
3. *Direct import*: the header sections of PE- and ELF-programs can contain names of external dynamic link libraries of the same level; a special pool of constants of the Bytecode-programs can contain names of external dynamically connected classes; the source text of Script-programs may contain commands for connecting external libraries;
4. *Emulated imports*: sections of the header of PE- and ELF-programs may contain names of external dynamic link libraries from the level of another OS family;
5. *Indirect exchange*: the machine code of PE- and ELF-programs, as well as instructions of Bytecode-programs, may contain system calls of functions for accessing (reading, writing) external files; the source text of Script-programs may contain commands for access (read, write) to external files;

Thus, each of the designated types of interactions can theoretically be defined for each of the types of programs. To define some types of interactions, it may also be necessary to define types of machine code processor architectures [16, 17, 18].

Step 5. Building a model of interactions. Having defined types of programs (at Step 3) and interactions (at Step 4), it is possible to build a full-fledged formal model of interaction, for which this step is intended. Thus, each detected and scanned program in Step 2 is located at its own level, and then communicates with other programs according to its interactions. Formally, such a model can look like this:

$$\begin{cases} M = \left\langle \{L_i\}, \left\{F_j\right\}, \{R_k\} \right\rangle \\ L_i = \left\langle L^T, \{P_l\} \right\rangle, L^T \in \{PE, ELF, ByteCode, Script\} \\ F_j = \left\langle F^T, \{L_i\} \right\rangle, F^T \in \{UnixLike, Windows\} \\ R_k = \left\langle R^T, \langle P_{l1}, P_{l2} \rangle \right\rangle, P_l = P^{FileName} \\ R^T \in \left\{ \begin{array}{c} DirectCall, EmulateCall, \\ DirectImport, EmulateImport, IndirectSwap \end{array} \right\} \end{cases}$$

where M is an interaction model that is a tuple of a set of levels of types of programs with $\{L_i\}$ programs, a set of $\{F_j\}$ OS families and a set of links between $\{R_k\}$ programs; each L_i level is a tuple of L^T type and the set of $\{P_l\}$ programs; the level type takes one of the previously described values: *PE, ELF, ByteCode, Script*; each F_j OS family is defined by a tuple of its F^T type and a set of levels of program types; OS family type takes one of the previously described values: *Unix-like, Windows*; each connection between R_k programs is defined by a tuple of its R^T type and a laid out tuple of the initial and final P_{l1} and P_{l2} programs; each P_l program is identified by its file name $P^{FileName}$; the link type takes one of the previously described values – *DirectCall, EmulateCall, DirectImport, EmulateImport, IndirectSwap*.

Thus, the Model determines the levels of program L_i types, some of which exist only in certain OS families – F_j; P_l programs at each level have R_k interactions with other programs.

Step 6. Intellectual analysis. The construction (at Step 5) of the Model makes possible, on its basis, to implement ML algorithms for the analysis of interactions [19, 20]. Here are the possible applications of the main methods of ML in the interests of the set research problem.

Classification, after preliminary training, allows one to classify groups of programs into legal and hostile, relying solely on the signs of their internal interactions, as well as OS system calls. Anomalization allows you to identify programs, specifics of which are significantly different from all others. Clustering provides a grouping of logically related programs, which makes it possible to analyze interactions between groups. The use of regression, which is the prediction of new data using existing one, will hypothetically allow predicting the appearance of module interactions in subsequent releases of the OS or its programs, which will give the expert information about future IS. Dimension reduction is rarely used to detect malicious programs; nevertheless, with the help of this ML method, it is possible to significantly optimize the information visualized by an expert (by reducing its volume).

A feature of the ML application is that in some cases manual adjustment are necessary. Thus, in addition to adjusting the operation of a step, an expert may need to perform it again if the result obtained by the Method turns out to be incorrect or insufficient.

Step 7. Visualization of interactions. Visualization transforms results of Step 6 into a form suitable for graphical analysis by an expert. So, for example, it is possible to create a description of the Model with additional information in the form of a graph or generate a complete listing with detailed information about all elements of the Model.

Step 8. Making adjustments. The final step in the Method is intended to provide the expert with the ability of customization of the previous steps results, i.e. correct the Method. The most obvious application, already mentioned earlier, is to return to Step 6 if the resulting visualization is not suitable for the expert. Naturally, in this way, an expert will only be able to fine-tune the ML algorithms, although the Method can be extended to adjust for earlier steps.

The described diagram of the method with highlighting the areas of interaction with an expert, steps and input/output data is shown in Fig. 2

3.5. Experiments

To test the performance of the Method, a prototype of the visualization software was created, on which the experiment was then carried out. The prototype supports scanning ELF- and PE-programs and detecting direct imports of libraries. The prototype architecture ensured consistent execution of the Method and consisted of the 8 modules. The composition and order of execution of modules corresponds to the steps of the Method, which, if the prototype is operational, will also speak of the Method's operability. Note, that currently, intellectual analysis (Step 6) will be performed manually.

The directory with the Termite application in the Unix-like OS – *Arch Linux 5.9.6-arch1-1* was taken as path to the directory for analysis. Despite modest functionality of the prototype and lack of Step 6 implementation (i.e. intellectual analysis), the experiment was carried out resulting in synthesizing certain information for the expert. For visualization was used the Gephi tool with the Force Atlas algorithm (which showed good representativeness for perception by an expert), the result of which was the graph in Fig. 3; the graph contains both ELF-programs and imported libraries. An expert analysis was conducted on the resulting visualization. First of all, there are two nodes worth noting, the "usr/liblibc.so.6" library and the "/bin/termite" utility, these are located in the center of the graph and therefore linked to the most of other nodes. The first one is a system library and therefore is often used. The second is the main

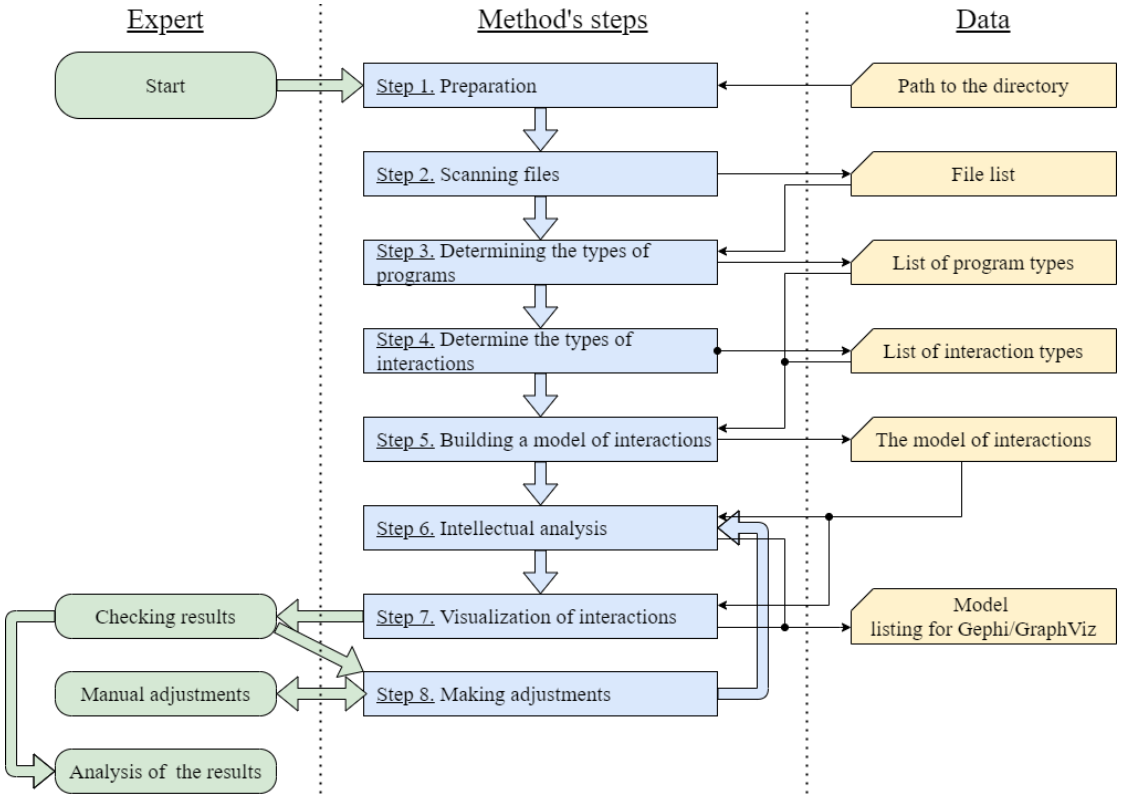

Figure 2: Diagram of the method for visualizing interactions

program of the application in question. This situation is quite natural and do not pose any interest to an expert, since system libraries are usually called (imported) by many user programs, and each software utility has single launch point.

Secondly, the graph contains 5 isolated groups of nodes (enclosed in dotted circles), connected with only one local central node, which in turn is connected to the main part of the graph; they are shown in Fig. 4 (designations of groups a) – e) correspond to those in Fig. 3).

These isolated groups (see Fig. 4) should be examined by an expert for security purposes. Possible prerequisites for the examination may be the presence of local centers and libraries exclusively imported to the local center, which indicates of specific functionality of group. Functionality can be determined by investigation of these centers. Thus, central nodes have following functionality: 1) */usr/lib/libgnutls.so.30* – implementation of the TLS (Transport Layer Security), SSL (Secure Sockets Layer) and DTLS (Datagram Transport Layer Security) protocols; 2) */usr/lib/libsystemd.so.0* – management of Unix-like OS services; 3) */usr/lib/libcairo.so.2* – drawing vector graphics; 4) */usr/lib/libgtk-3.so.0* – displays the GTK (GIMP ToolKit) GUI; 5) */usr/lib/libgdk-3.so.0* – wrapper over low-level functions of graphical primitives.

The use of such clustering allows expert to restore the architecture of applications, which will allow the detection of high-level vulnerabilities [21]. The use of ML (Step 6) will make this process automated.

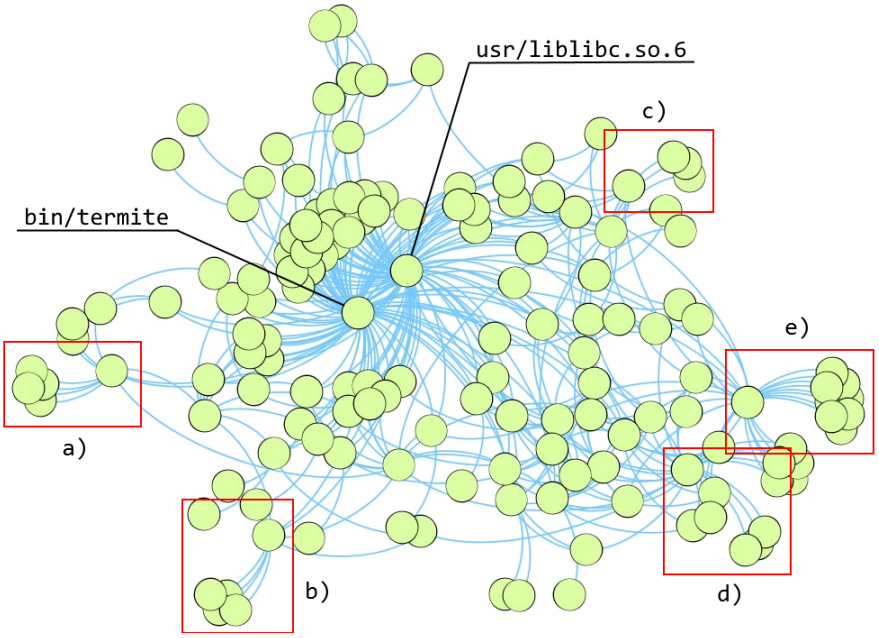

Figure 3: Visualization of programs interaction synthesized by Gephi tool

4. Discussion

Let's consider the main disadvantages of the proposed Method and ways to eliminate or mitigate them.

Firstly, not all anomalous connections unambiguously indicate a violation of IS and require special attention from an expert; nevertheless, the development of the automated intellectual analysis will partially reduce the costs of the expert's time and resources. Also, applying filtering to directories and file names will cut off objects that are not essential for analysis in advance.

Secondly, even for the average volume of the directory, the visualization of all its programs and links will receive an image that is huge in size and number of links, which will significantly complicate the understanding of the Model by an expert. In this case, the decisive factor is the reduction of dimensions at Step 6, which will highlight only the most essential programs and connections.

Thirdly, the Method does not consider more complex types of interactions (RPC, Unix-like intermediaries, etc.). To take them into account, a deeper analysis of the program code, operating mechanisms of the OS and partial use of dynamic analysis or logging will be required.

And, fourthly, the problems that hypothetically arising in any automation of the process can be smoothed out by providing an expert with opportunities to influence configuration of the Method's steps. So, in addition to Step 6, the expert can customize paths in Step 1, types of scanned files in Step 2, features of analyzed program types in Step 3, configure algorithms determining interaction types in Step 4, explicitly indicate not automatically determined interactions in Step 5, and adjust synthesized visualization in Step 7.

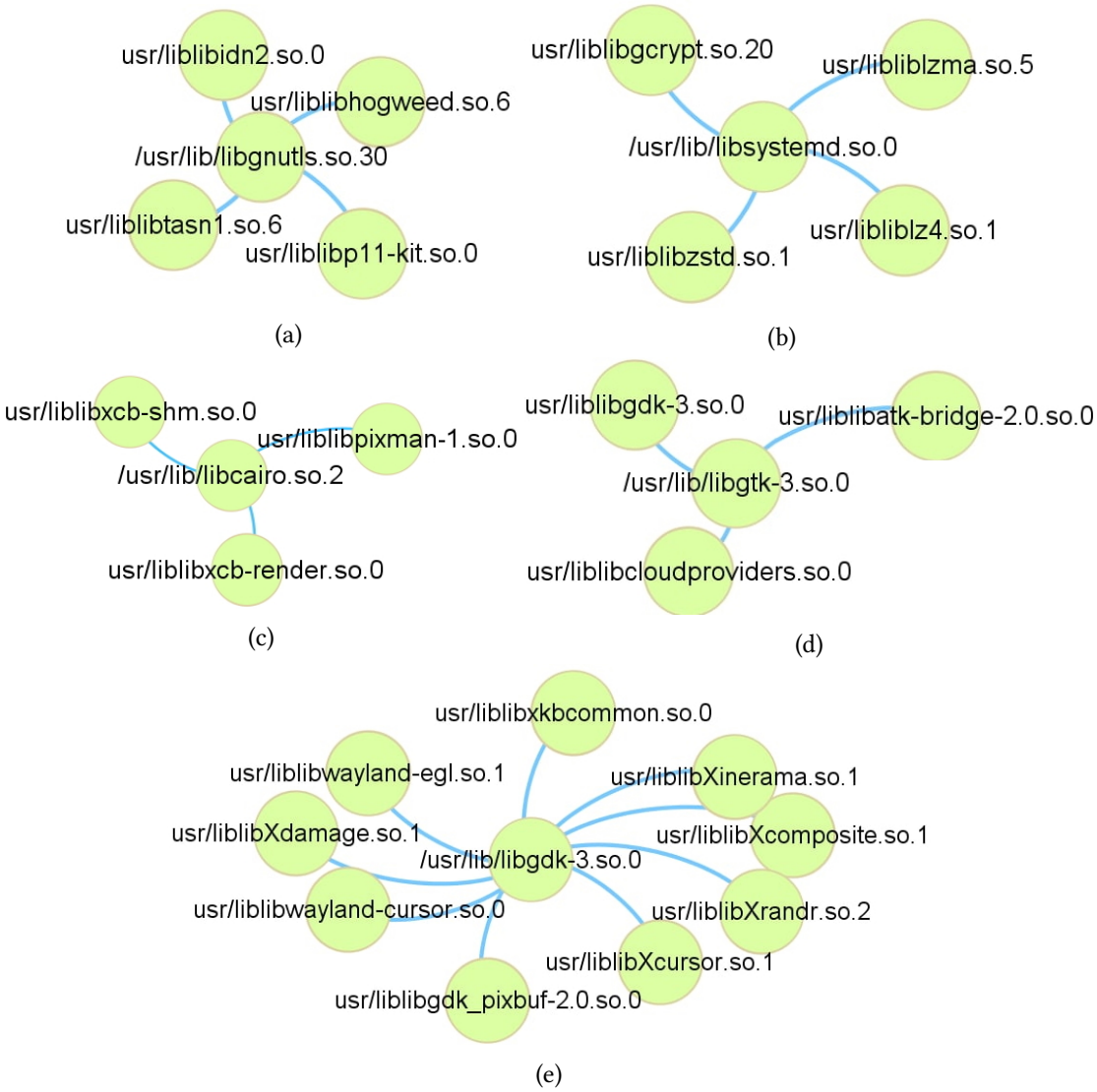

Figure 4: Visualization of the isolated groups interaction synthesized by Gephi tool

5. Conclusions

The Method proposed by the authors has an novelty, since it seeks to take into account all possible interactions of programs (and not just one of its types), while processing the results during its performance full machine learning functionality.

Created prototype, which possesses the minimum necessary functionality, confirmed efficiency of the Method (with the exception of intellectual analysis), having already received results that allow making certain conclusions regarding examined directories.

A further direction of research should be a full-fledged implementation of algorithms for determining all types of programs, as well as their interactions (including those mentioned,

but not taken into account in the Method). At the same time, following the current trend, special efforts must be paid to the application of ML methods in the interests of the problem under consideration, since it is obvious that no matter how successful the visualization of the entire set of programs and their connections is, without the help of artificial intelligence, the expert will not be able to conduct a full audit in the foreseeable future. There is no conflict of interest.

References

[1] L. Sportiello, S. Zanero, File block classification by support vector machine, in: 2011 Sixth International Conference on Availability, Reliability and Security, 2011, pp. 307–312. doi:10.1109/ARES.2011.52.

[2] W. Arif, N. A. Mahoto, Document clustering – a feasible demonstration with k-means algorithm, in: 2019 2nd International Conference on Computing, Mathematics and Engineering Technologies (iCoMET), 2019, pp. 1–6. doi:10.1109/ICOMET.2019.8673480.

[3] T. Wang, C. Wu, Detection of packed executables using support vector machines, in: 2011 International Conference on Machine Learning and Cybernetics, volume 2, 2011, pp. 717–722. doi:10.1109/ICMLC.2011.6016774.

[4] N. Hubballi, H. Dogra, Detecting packed executable file: Supervised or anomaly detection method?, in: 2016 11th International Conference on Availability, Reliability and Security (ARES), 2016, pp. 638–643. doi:10.1109/ARES.2016.18.

[5] B. Shravan Kumar, R. Vadlamani, Text document classification with pca and one-class svm, in: 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications, 2017, pp. 107–115. doi:10.1007/978-981-10-3153-3_11.

[6] I. E. Krivtsova, I. S. Lebedev, K. I. Salakhutdinova, Identification of executable files on the basis of statistical criteria, in: 2017 20th Conference of Open Innovations Association (FRUCT), 2017, pp. 202–208. doi:10.23919/FRUCT.2017.8071312.

[7] Q. Su, S. Si, W. Wu, J. Huang, W. Fan, X. Li, The library function recognition algorithm of pe file disassembler research and implementation, in: 2011 IEEE International Symposium on IT in Medicine and Education, volume 2, 2011, pp. 132–135. doi:10.1109/ITiME.2011.6132073.

[8] I. Seo, I. Kim, J. Yoon, J. Ryou, Detection of unknown malicious codes based on group file characteristics, in: 2010 Proceedings of the 5th International Conference on Ubiquitous Information Technologies and Applications, 2010, pp. 1–6. doi:10.1109/ICUT.2010.5677901.

[9] I. I. Livshitz, K. A. Nikiforova, P. A. Lontsikh, S. N. Karasev, The new aspects for the instantaneous information security audit, in: 2016 IEEE Conference on Quality Management, Transport and Information Security, Information Technologies (IT MQ IS), 2016, pp. 125–127. doi:10.1109/ITMQIS.2016.7751920.

[10] H. Shukla, S. Patil, D. Solanki, L. Singh, M. Swarnkar, H. K. Thakkar, On the design of supervised binary classifiers for malware detection using portable executable files, in: 2019 IEEE 9th International Conference on Advanced Computing (IACC), 2019, pp. 141–146. doi:10.1109/IACC48062.2019.8971519.

[11] H. Jeong, J. Baik, K. Kang, Functional level hot-patching platform for executable and linkable format binaries, in: 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC), 2017, pp. 489–494. doi:10.1109/SMC.2017.8122653.

[12] A. Amine, B. Mohammed, L. Jean-Louis, Generating control flow graph from java card byte code, in: 2014 Third IEEE International Colloquium in Information Science and Technology (CIST), 2014, pp. 206–212. doi:10.1109/CIST.2014.7016620.

[13] P. Ferrara, A. Cortesi, F. Spoto, Cil to java-bytecode translation for static analysis leveraging, in: 2018 IEEE/ACM 6th International FME Workshop on Formal Methods in Software Engineering (FormaliSE), 2018, pp. 40–49.

[14] N. Feng, J. Xie, Y. Wu, Comparison of ruby on rails development tools, in: 2009 WRI World Congress on Software Engineering, volume 4, 2009, pp. 290–294. doi:10.1109/WCSE.2009.229.

[15] Z. Chen, L. Chen, B. Xu, Hybrid information flow analysis for python bytecode, in: 2014 11th Web Information System and Application Conference, 2014, pp. 95–100. doi:10.1109/WISA.2014.26.

[16] M. Buinevich, K. Izrailov, Identification of processor's architecture of executable code based on machine learning. part 1. frequency byte model, Proc. of Telecom. Universities 6 (2020) 77–85. doi:10.31854/1813-324X-2020-6-1-77-85.

[17] M. Buinevich, K. Izrailov, Identification of processor's architecture of executable code based on machine learning. part 2. identification method, Proc. of Telecom. Universities 6 (2020) 104–112. doi:10.31854/1813-324X-2020-6-2-104-112.

[18] M. Buinevich, K. Izrailov, Identification of processor's architecture of executable code based on machine learning. part 3. assessment quality and applicability border, Proc. of Telecom. Universities 6 (2020) 48–57. doi:DOI:10.31854/1813-324X-2020-6-3-48-57.

[19] M. Berman, S. Adams, T. Sherburne, C. Fleming, P. Beling, Active learning to improve static analysis, in: 2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA), 2019, pp. 1322–1327. doi:10.1109/ICMLA.2019.00215.

[20] A. M. Radwan, Machine learning techniques to detect maliciousness of portable executable files, in: 2019 International Conference on Promising Electronic Technologies (ICPET), 2019, pp. 86–90. doi:10.1109/ICPET.2019.00023.

[21] M. Buinevich, K. Izrailov, A. Vladyko, The life cycle of vulnerabilities in the representations of software for telecommunication devices, in: 2016 18th International Conference on Advanced Communication Technology (ICACT), 2016, pp. 430–435. doi:10.1109/ICACT.2016.7423420.