Method and algorithms of visual audit of program interaction

Mikhail V. Buinevich[1,3], Konstantin E. Izrailov[2,3*], Igor V. Kotenko[2], and Pavel A. Kurta[3]
[1]Saint-Petersburg University of State Fire Service of EMERCOM of Russia, Saint-Petersburg, Russia
bmv1958@yandex.ru
[2]St. Petersburg Federal Research Center of the Russian Academy of Sciences, Saint-Petersburg, Russia
konstantin.izrailov@mail.ru, ivkote@comsec.spb.ru
[3]The Bonch-Bruevich State University of Telecommunications, Saint-Petersburg, Russia
expert@kurta.ru

Received: December 3, 2020; Accepted: February 11, 2021; Published: February 28, 2021

Abstract

Modern software products consist of a lot of executable files. Simultaneously, there are complex data flows between them. As a result, the task of auditing such data interactions of programs arises. In particular, this can be in demand in the field of information security for detecting holes in architectures of solutions. Such types of programs as PE (for Windows), ELF (for Linux), CIL (.Net bite code), JBC (Java bite code) and Script (interpretable code) should be mentioned. The types of interactions include direct program fetching, direct import of libraries and exchange of external files. The paper discusses the authors' own method of analysing software products and visualizing the interaction of programs to be studied by the Expert subsequently. The work of the method is based on the generalized model of interactions set analytically. The authors describe the results of the experiment with the developed prototype, implementing this method for five standardized and qualitatively different software products. The results of the experiment are analyzed by the expert from the point of view of product information security. A conclusion is made about the necessity of automatizing the expert's work. For this purpose, we propose the corresponding algorithms as the pseudo-code.

Keywords: Software, Interaction, Audit, Visualization, Information Security

1 Introduction

The application of different software types is constantly growing – from home PCs and secret computer centers to public and common use [27]. Obviously, this process is accompanied by numerous advantages brought about by the automatization of life processes, improvement of life quality via robotechnics and cybertechnology, exchange of data between any places in the world. Nevertheless, software, especially if it is implemented in physical processes has some drawbacks. Thus, the malevolent code, for instance, viruses in commonly used software products (e.g. for secure USB [29]), can bring about information threats to more users than when this software is used rarely or is only for personal use. The situation is aggravated by the fact that the malevolent code is developed on purpose, is implemented latently and tends to look inconspicuous for antiviruses – all this makes it more difficult to reveal and neutralize [24]. In some cases, vulnerability of security can be revealed only manually by the Security Software Expert (henceforth – the Expert). However, the task is being made much more difficult, because a far greater number of software features is needed, which leads to a bigger size and composition of its particular programs (such as executable files).

Journal of Internet Services and Information Security (JISIS), volume: 11, number: 1 (February 2021), pp. 16-43
DOI:*10.22667/JISIS.2021.02.28.016*

*Corresponding author: The Bonch-Bruevich State University of Telecommunications, Saint-Petersburg, Russia, Tel: +79215582389, Email: konstantin.izrailov@mail.ru

Vulnerabilities are becoming more abstract – from quite simple low-level ones (for example, errors in operations), via more complex ones of the intermediate level (for example, software implants in algorithms) to the extremely complex one of the highest level (for example, vulnerabilities in architecture) [11]. In order to reveal and analyse those, the Expert might need to form in his mind the general picture of software operation, the sophisticated interconnections of its components, their peculiarities and metrics [12]. Thus, for example, the non-typical fetching of the cryptographic calculations program by the other program working with the network can be a sign of the leakage of confidential data (usernames, passwords, etc.) This is a pressing challenge for information security.

To deal with this challenge in the past the method of visualizing the interactions of programs for the purpose of auditing information security of the operation system (henceforth – OS) was used. The method was tested at the IEEE-conference [10]; also, its performance was tested partially for Linux family programs. It was demonstrated that the analysis of big software products (a huge number of programs, complex connection of different types, the wide use of system libraries, etc. [26]) would be difficult for the Expert even with the total visualization of their interactions. Thus, a conclusion can be made that it would be useful to develop intelligent algorithms of automatic processing of such information. In the latter case, of course, we mean the work not with the graphical representation of interactions between programs, but with a textual description of interactions in the formalized graph-based form.

To make the method intelligent, an improved prototype was developed (henceforth – the Prototype), which has more functions and works in the environment and with Windows files. One of its practical peculiarities is the fact that the architecture of the Prototype allows implementing new processing algorithms without significant adaptation, i.e. to extend it in terms of intellectualization.

The essence of the proposed approach is statistical analysis of all executable files of the software product with the aim of detecting data flows between them. The construction of the interactions model based on the results will allow both using formal algorithms of intelligent analysis and providing the Expert with the information about interaction in the intuitively understandable way.

Our results were novel, because, unlike most instruments of statistical analysis aimed exclusively at work with separate files, the proposed approach is aimed at processing the whole set of files within the same software product. Moreover, if the existing solutions are aimed at automatic search for formalized vulnerabilities, the aim of our method is to search for poorly formalized weaknesses in product architectures, using the Expert's manual work.

Our research procedure can be described as follows. At first, using the developed scheme of the visualization method, we created the Prototype, working in the Windows environment. Prototype settings allow us to control the quality of the obtained visualization. After that we conducted an experiment by using the Prototype for five typical and qualitatively different software products. The obtained results were analyzed from the point of view of auditing inter-software interactions from the information security perspective. The conclusions were made that visualization of interactions suits the Expert only in some cases, and, in order to improve his efficiency intelligent analysis algorithms, have to be used. Finally, on the basis of the conducted experiment and the previous experience of the authors [8] 8 such algorithms were proposed, including the formal view of those.

The article is organized as follows. Section 2 reviews the works devoted to the issues of software analysis and visualization. Section 3 gives a brief description of the obtained results as the authors' intelligent visualization method. Also, this section describes the implementation of the method as the Windows Prototype. Section 4 discusses the experiment of using the Prototype for a number of popular software products. Based on the results of the experiment and the authors' practical experience, Section 5 briefly describes the algorithms of analysis of interactions. The obtained results are discussed in Section 6.

2 State of Art

Let us discuss briefly the works devoted to the ways of analysing software products, as well as the methods of visualizing their characteristics.

[45] proposes the method of Security Engineering Risk Analysis (SERA), which allows forecasting the risks of software security during its lifetime and updates. 3 main vulnerability causes are specified – design drawback, realization errors and incorrect configuration.

[2] describes the approach aimed at revealing the risks of information security in software products architecture. For this purpose, the formalization of attack scripts and security metrics with the help of the object constraint language (OCL) is used. Such attack scripts as Man-In-The-Middle, Denial-Of-Service, Data Tampering Attack and Injection Attack are described. Also, the following security metrics for architecture are discussed: Attack Surface, Compartmentalization, Fail Securely, Defense-In-Depth (Layered Security), Isolation Metric.

[46] is devoted to checking the security of the applied software product of the system by analysing system calls of its constituting programs. In order to obtain the sequence of calls, associative rules are used. After that they can be stored in the unified database and used for detecting malicious software. Experiments prove the efficiency of the proposed solution.

[1] compares the tools of statistical analysis of the source code, such as WASP Yasca Project, FindBugs and Microsoft Code Analysis Tool.NET (CAT.NET). The tools are used for the analysis of the source, bite- and binary codes, correspondingly.

[23] describes a method of fighting with the hidden features of software by analysing its architecture. For this purpose, the authors, among other methods, suggest using Software Architecture Visualization and Evaluation (SAVE). During the experiment the "image" of high-level vulnerability of the backdoor type was obtained. Using it, the intruder can get non-sanctioned access to the system with the administrator's rights.

[7] describes the interface which is supposed to help the user in choosing the method and visualization settings of the software, in particular, its external architecture and inner content. An example of two-dimensional visualization of relations between OSGi packages can be found in Figure 1.

[42] reviews the methods of 3D visualization of software, which are going to replace 2D very soon. The following types of quasi 3D-visualization are mentioned: Augmented 2D views, Adapted 2D views and Inherent 3D application domain views.

[37] describes interactive tools for representing the architecture of the real application with the help of the 3D virtual reality headset. Dependencies between packages are visualized with the help of the lines, whose color depends on the direction of interconnection (import or export).

[31] develops quite an interesting idea of "gamification of software development", i.e. the creation of a computer game analog. The interactive tool CityVR for software visualization is proposed. It is based on virtual reality and involves users in the development process.

[30] discusses the influence of visualization methods when solving various software-related tasks. A scene visualization experiment is carried out using such methods as a standard computer screen, an immersive 3D environment and a physical 3D printed model. It is demonstrated that every method has its own advantages.

[17] proves the advantages of using new VR technology, such as Oculus Rift, in different spheres.

[33] suggests the visualization framework for developing secure software. The framework allows developers to assess the complexity of a code and visualize the relations between classes and created objects.

The experience of researching information systems from the point of view of their information security suggests that this process can be divided into four major stages: information collection, data preparation, data processing and visualization of results. Based on the previously noted difficulties of manual

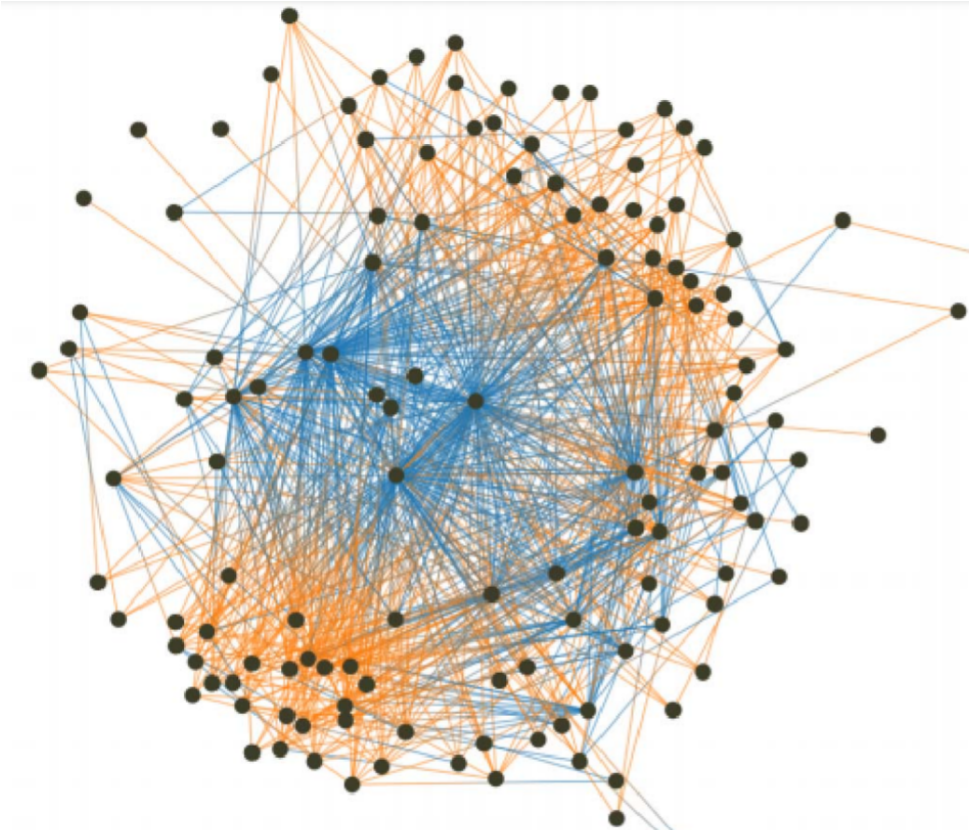


Figure 1: An example of two-dimensional visualization of relations between OSGi packages

analysis by the Expert of program interactions, it is advisable to add between the third and fourth stages an auxiliary stage intelligent analysis, which provides the last stage with information more adapted for the Expert. Let us compare the papers from the review from the point of view of their applicability for creating each of the stages (Table 1) in the interests of solving the problem considered in the paper. At the same time, the approach proposed in the paper should obviously include and fully implement all 5 stages.

In spite of the fact that some studies of the above issues do exist, no comprehensive and ultimate solution has been found yet. Thus, besides possessing high relevance, our method, is quite novel as well. Now we are going to give its general description. We will also demonstrate the process and the results of some experiments with the practical implementation of the method.

3 Visualization and analysis

3.1 Types of programs

Big software products are actually sets of particular interacting pieces of software, which, among other functions, ensure data flows exchange. In some cases, the analysis of these flows allows us to find vulnerabilities, and the complexity and poor formalization of such analysis requires the Expert's manual operations.

This is the list of the main types of programs for the majority of operating systems:

a) PE-programs (PE), characteristic for the Windows family, including DLL (Dynamic-link library)

Paper	Information collection	Data preparation	Data processing	Intelligent analysis	Visualization
[45]	Partially	No	Partially	Partially	No
[2]	Partially	Partially	Partially	Yes	No
[46]	Yes	Yes	Yes	Partially	No
[1]	No	Yes	Partially	No	No
[23]	No	Partially	Partially	Partially	Partially
[7]	No	No	Yes	Partially	Yes
[42]	No	No	Partially	No	Partially
[37]	Partially	Partially	Yes	No	Yes
[31]	No	Partially	No	No	Partially
[30]	Partially	No	Partially	Partially	Yes
[17]	No	No	Partially	No	Yes
[33]	No	Partially	No	Yes	Partially
This paper	Yes	Yes	Yes	Yes	Yes

Table 1: Comparative analysis of the possibility of using the results of various papers

libraries (of the same format and features) [39];

b) ELF-programs (ELF), characteristic for the Linux family, similar to PE-files (instead of DLL the so-called shared objects are used) [22];

c) JBC-programs (JBC), consisting of the bite code, executable on Java by the virtual machine, [4];

d) CIL-programs (CIL), also consisting of the bite code, executable on .Net by the virtual machine [16];

e) scripts such as Ruby and Python, executable by various interpreters and containing the source code [13].

It should be pointed out that each of the mentioned program types can be involved in interaction. However, it is worth while using various graphical objects (by shape, color, etc.) when dividing them during visualization.

3.2 Types of interactions

Programs within one product (including OS), as a rule, exchange information. Otherwise, it would be useless to unite them logically. This exchange is provided by various program interactions. The basic ones are as follows:

a) Direct Call or when the control is directly transferred from one program to the other one (including parallel, when the program calls the other one and continues running itself);

b) Direct Import or connecting to the program of dynamic libraries (with subsequent call of functions from them);

c) Indirect Swap or transferring information from one program to the other via temporary files (for example, the program writes data into the file which will be read by the other program).

The determination of direct import is the easiest one, because this information is stored in the executable file header (ELF or PE). For determining direct interaction, the complex analysis of the machine, bite or script code of the program will be needed. Indirect swap is seen as a more difficult task, because, besides code analysis, interconnections must be restored via temporary files.

It should be pointed out that although highly complex and specific ways of interaction do exist (via RPC, sockets, etc), they are not discussed here.

3.3 The model of interaction

Using the above types of programs and interactions, the authors have developed the generalized model, which can be written down formally as follows:

$$\begin{cases} Model = \langle Levels \mid Relations \rangle \\ Levels = \bigcup Level_i \\ Level_i = \langle Program_{Type} \mid Programs \rangle \\ Program_{Type} \in \{PE,\ ELF, JBC, CIL, Script\} \\ Programs = \bigcup Program_j \\ Program_j = Program_{Filename} \\ Relations = \bigcup Relation_k \\ Relation_k = \langle Relation_{Type} \mid Relation_{Direction} \rangle \\ Relation_{Type} \in \{DirectCall,\ DirectImport,\ IndirectSwap\} \\ Relation_{Direction} = \left\langle Program^{From} \,\middle|\, Program^{To} \right\rangle \\ Program^{From}, Program^{To} \in Program_j \end{cases}.$$

The model introduces the following entities and determines their relations:

- The model consists of Levels (sets of elements $Level_i$) and Relations (sets of elements $Relation_k$);
- $Level_i$ consists of program types ($Program_{Type}$) and Programs (sets of elements $Program_j$) of these types (determined by the paths to them);
- Program types include PE, ELF, JBC, CIL, Script;
- $Relation_k$ consists of relation type ($Relation_{Type}$) and relation direction ($Relation_{Direction}$);
- Interaction types consist of DirectCall, DirectImport, IndirectSwap;
- Relation direction consists of the paths of program-source ($Program^{From}$) and program-recipient ($Program^{To}$).

The visualization method was developed using the proposed model.

3.4 The visualization method

The developed method, described in our previous works, provides the visualization of the relation of programs and consists of the following seven steps.

At Step 1 the following general preparatory actions are carried out, such as showing the paths with programs for visualization of interactions, the settings of the programs processing procedure, etc.

At Step 2 the files are scanned by determining and selecting them, as well as parsing internal formats.

During Steps 3 and 4 the types of scanned programs are determined, as well as the types of relations between them.

Step 5 is the main one for the operation of the method, because it builds the model of interactions, which takes into account the features of the program and suits for analysis and visualization.

The intellectualization of the method is provided by Step 6, which makes it possible to use the algorithms of processing a model (representing a certain graph) with the aim of selecting meta-information, which will help the Expert to analyse visualization effectively.

It would be useful to employ machine learning methods at this stage [32, 35, 6, 20, 41]. Let us describe how it can be done:

- the classification will make it possible to detect malicious programs by patterns (for example, by dividing all programs into security classes);
- the detection of anomalies will help find programs which do not work in a typical (secure) manner;
- clusterization will allow us to detect logical groups in interacting programs, i.e. clusters, which will help the Expert to reconstruct the general architecture of the product;
- regression can provide information about the tendencies in the development of various versions of the product, which can, in this way, help us to understand deviations from it;
- reducing dimensionality optimizes the display of interactions during visualization, eliminating the information which is not relevant for the Expert.

Algorithmically, the method has the following code – Algorithm 1.

The algorithm at the input takes the path to the product, the interactions of the programs of which must be analysed, and returns the visual representation of this algorithm at the output. The body of the algorithm consists of the consecutive performance of all the steps of the visualization method: reviewing options, scanning files, obtaining the types of files and types of their interactions, building the model itself, using intelligent algorithms while obtaining additional meta-information, generating the image code by the model and meta-information (for example, in the DOT-format), as well as the creation of the final image. The algorithm is the main branch of the work of the method.

3.5 Intelligent analysis

Step 6 (intelligent analysis) is quite important and interesting (from the theoretical and practical perspectives) for the following reason. All the other Steps can be considered quite standard for this type of tasks, and this step depends directly on the analysis of the peculiarities of a software product and reflects the aim of the Expert's work – information security auditing.

The present article is devoted specifically to the study of the possibility of implementing this step. The basic variants of the algorithms of the step will be described later.

3.6 The prototype

For the purposes of this study, during the practical application of the visualization method the prototype of visualization software for Windows was developed. C# was selected as the programming language, because it allows fast and simple development and high speed of performance. Microsoft being its developer, it is close to Windows and it has a lot of libraries, which can be used for intelligent analysis (for example, ML.Net and Accord.Net for machine learning). The prototype is now at the stage of copyright registration.

The prototype (the name of the launched file: *PeLinkScan.exe* has the following functional possibilities and limitations, because the aim of the Prototype was the study of hypothetical possibilities of intelligent analysis:

Algorithm 1: VisualizationMethod

```
Input:
Path – Path to software product for analysis
Output:
Image – Image for visualization of a program interaction
begin
    // Step 1
    Options = ParseOptions();
    // Step 2
    List⟨File⟩ Files = ScanFiles(Path, Options.ForScan);
    List⟨Type⟩ Types;
    List⟨Relation⟩ Relations;
    // Step 3
    foreach File in Files do
        Type = GetType(File);
        Types.Add(Type);
    end
    // Step 4
    foreach File in Files do
        RelationsInFile = GetRelations(File);
        Relations.Add(RelationsInFile);
    end
    // Step 5
    Model = BuildModel(Files, Types, Relations, Options.ForBuild);
    // Step 6
    MetaInfo = IntelligentAlgorithms(Model, Options.ForAlgs);
    // Step 7
    ImageCode = GenerateImageCode(Model, MetaInfo, Options.ForImage);
    Image = BuildImage(ImageCode);
    return Image;
end
```

- work in the Windows environment;
- work only with PE-files (including DLL);
- work only with the direct import of libraries;
- the *OnlyForPart* mode (optional): only imported libraries, belonging to this software (i.e. without system ones) are taken into account;
- the minimal optimization of operation speed;
- omission of programs, which are not interacting with anyone (i.e. the ones representing separately standing objects during visualization);
- the generation of interactions visualization in the DOT language (description by the graph; represented graphically with the help of special programs and Internet resources).

The possible intelligent analysis algorithms will be discussed later. In order for the Prototype to work, the path of the software product must be specified. Its files will be processed, which will result in the visualization of interactions. During the visualization of final programs, not imported by the others, the rectangle shape is used, while for the others – the ellipsis shape; the shapes of the programs which are part of the product are highlighted green, the system ones – grey; the fact that programs have interacted is reflected by the line between the shapes. The width of this line is calculated by the following formula (for smoothing the lines): $width = log(numOfFunct)$, where width is the $width$ of the line, $numOfFunct$ – the number of functions imported from programs.

4 The experiment

The visualization of any software product has its own topology, peculiarities and, as a result, factors which can hypothetically disrupt information security. By way of example, let us describe the visualization of five software products, the first two of which (*Windows Photo Viewer* and *Windows Media Player*) are part of *Windows 10 Home*, the third one (*VMware Workstation Pro 16*) is quite popular for solving the task of virtualizing the OS, the fourth one (*Microsoft Office Professional Plus 2019*) is produced by *Microsoft* (as well as the first two ones), and the fifth one (*Paint.Net*) is the non-commercial graphic editor (by license agreement), which has proved its worth. The latter product, as it will be shown later, will be the most interesting one in terms of analysis of visualization and interactions in it.

4.1 Windows Photo Viewer (Windows 10 Home)

The product is a part of Windows 10 by the standard path "C:\Program Files\Windows Photo Viewer\" and is aimed at viewing graphic images. As it is quite compact (5 Mb) and has few files (eight executable ones), let us describe in more detail the process of visualizing interactions of software in it.

Running the Prototype for this product with the command prompt

```
PeLinkScan.exe "C:\Program Files\Windows Photo Viewer\" -OnlyForPart
```

will lead to the execution of the visualization method by scanning Windows Photo Viewer files, building the interaction model, its processing and visualization; the *OnlyForPart* option points out that we should analyze and visualize only the programs which are part of the product (i.e. without importing system libraries). At the output the Prototype will generate the following description of interactions graphs in the Dot format:

```
digraph G {
  2 [ label = "ImagingEngine.dll", shape = "ellipse", style = "filled",
    fillcolor = "green" ];
  3 [ label = "PhotoAcq.dll", shape = "ellipse", style = "filled",
    fillcolor = "green" ];
  4 [ label = "PhotoBase.dll", shape = "rect", style = "filled",
    fillcolor = "green" ];
  2 -> 4 [ penwidth = 4 ];
  3 -> 4 [ penwidth = 4 ];
}
```

According to the above description the graph consists of 3 nodes, corresponding to file names *ImagingEngine.dll*, *PhotoAcq.dll*, *PhotoBase.dll*. The first two nodes are ellipse-shaped, and the last one is a rectangle, because it does not import other libraries (it should be reminded that the import of system libraries is not considered). All nodes are highlighted green, because they are a part of the product. Also, the first two programs import the third one (which is written down as "2 → 4" and "3 → 4" in

the description of the graphs). The lines in the graph corresponding to this have the same width 4, because the number of imported functions coincides or nearly coincides. The graphic representation of such interactions (i.e. direct visualization, suitable for the Expert for analysis purposes), obtained via https:dreampuf.github.io (the visualization engine mode – FDP), is presented in Figure 2

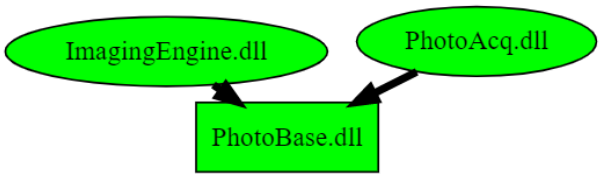


Figure 2: Visualization of interactions for the product "Windows Photo Viewer" (Windows 10 Home)

The Expert's main conclusion is that the product simplicity does not presuppose potential "security holes".

4.2 Windows Media Player (Windows 10 Home)

This product, as well as the previous one is a part of Windows 10 by the standard path "C:\Program Files\Windows Media Player\" and serves for playing audio and video files. The product is also quite compact (3 Mb), though it has far more files (26 files, 17 of which are executable).

If the Prototype of this product is run with the key *OnlyForPart*, the graph will be empty, because (as will be shown later) its programs do not import each other, i.e. they are isolated from the direct import perspective. It should be pointed out that they can exchange information using other mentioned interaction types – direct, indirect swap (and also not considered RPC, via sockets, etc.)

The call of the prototype for the product without the key *OnlyForPart* will cause only the generation of the description of interactions with the following graphical representation (Figure 3).

Unfortunately, it is difficult to read the text in Figure 3) (high sparsity of the nodes, while the size of the text is small) due to the peculiarities of the visualization engine operation. However, the problem can be solved by scaling, as well as studying graphs in editors working with the SVG format (into which the obtained graph can be transformed, too). It can be also seen in the figure that out of 17 programs of the product only 10 import any system libraries; the others have no imports at all.

The Expert's main conclusion is that (as it was found earlier), programs of the product do not import each other, realizing in this way isolated features. Therefore, if one of the programs has a virus, it will not influence the security of others. Also, studying the assignments of imported system libraries (nodes highlighted grey) will enable us to understand the assignment of the product programs, revealing the suspicious ones – for example, for the case when the text editor is importing network libraries extensively.

4.3 VMware Workstation Pro 16

The product is quite well-known and often used when solving virtualization tasks (for example, for the operation in the Linux family for Windows); it is installed by the standard path: "C:\Program Files (x86)\VMware\VMware Workstation\". The product is an average one, compared to the others (the size is 1 GB), containing 319 files (127 – executable).

The graphical representation of interactions can be found in Figure 4.

The Expert's main conclusions are as follows.

Firstly, several central nodes, are imported most often by the others: *zlib1.dll* (the compression library), *sigc-2.0.dll* (standard library C++ for typesafe callbacks), *vmwarewui.dll* and *vmwarecui.dll*

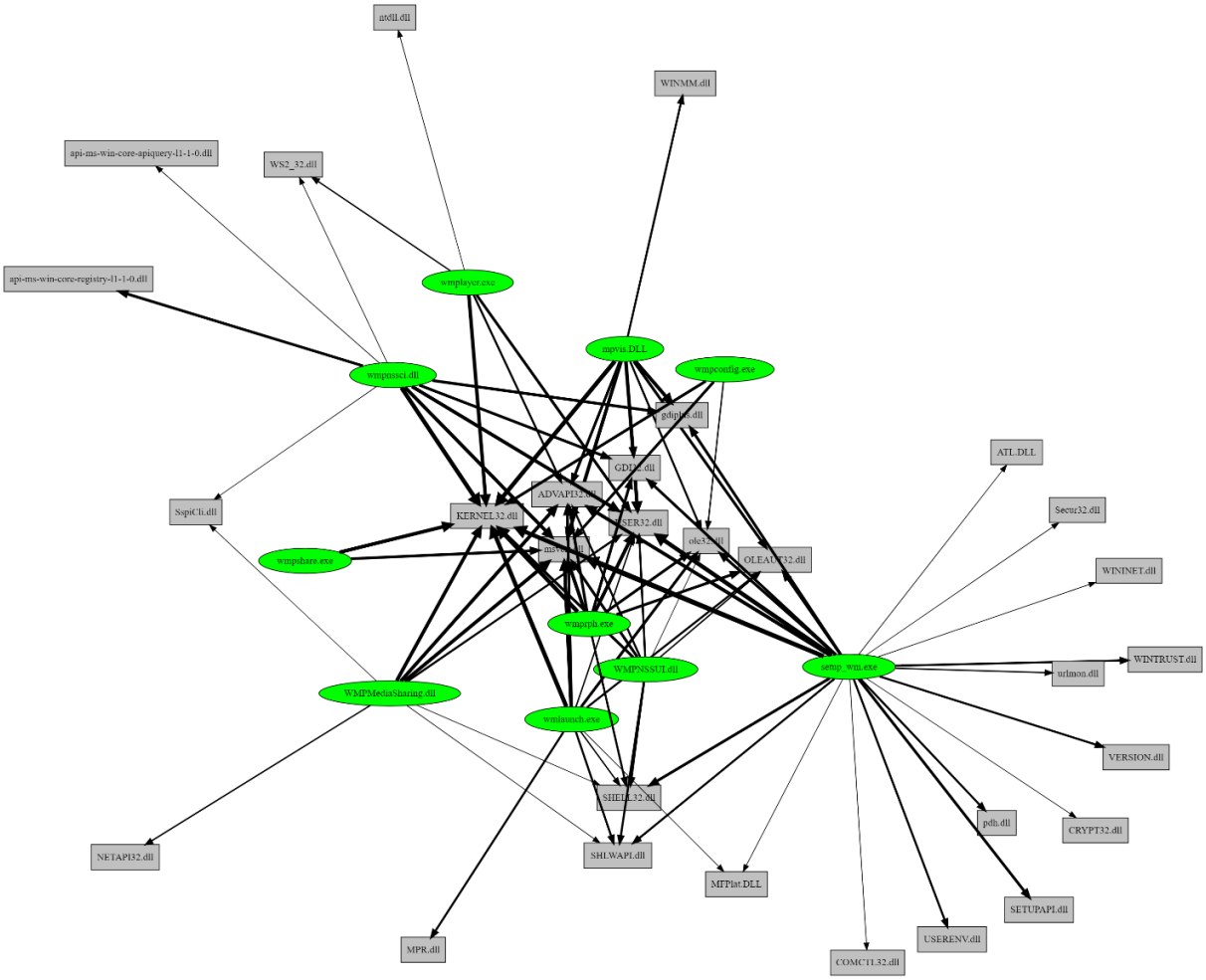


Figure 3: Visualization of interactions for the product Windows Media Player (Windows 10 Home)

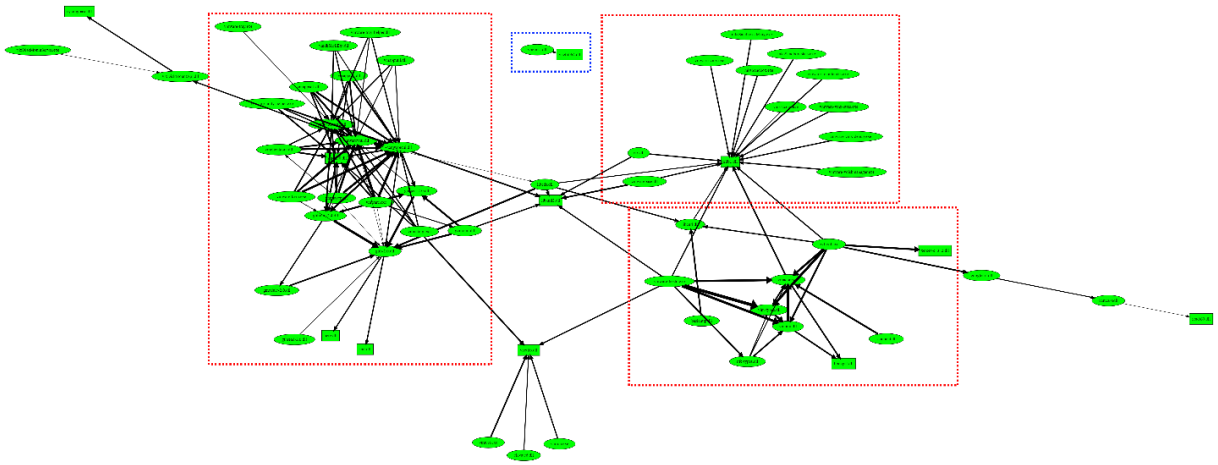


Figure 4: Visualization of interactions for the product VMware Workstation Pro 16

(service libraries *VMware*). The Expert should pay particular attention to their security, because, if they are infected, it will influence the other programs of the product.

Secondly, during visualization a certain clusterization of the nodes into 3 groups (marked by red rectangles) has already occurred, which is a sign of a certain logical structure of the product; it will allow the Expert to reveal high level (architecture) vulnerabilities.

Thirdly, there are two isolated nodes (marked by blue rectangles) in the graph: *vnetinst.dll* and *vnetlib64.dll*, responsible for *VMware Network Driver*. Their presence can also be interpreted as a separate utility program in the product, which means that the same study approach as to the whole product should be applied to (for example, searching for in points and analysis of command line options).

4.4 Microsoft Office Professional Plus 2019

The product is a popular set of office applications (*Word*, *Excel*, *Access* etc.) produced by *Microsoft*; it is installed by a standard path "C:\Program Files (x86)\Microsoft Office\root\Office16\" and is used for working with huge numbers of documents of different types. The product can be considered quite big (the size 1.5 Gb) – it consists of 3007 files (877 of which are executable).

It is obvious that for such a large number of files using the Prototype without the *OnlyForPart* mode will turn out to be not very useful, therefore let us provide here the graphical representation of interactions with the turned-on mode (Figure 5).

The Expert's main conclusion is that the visualization of products with such a big number of files is practically useless for the Expert. Nevertheless, proceeding from the graph topology (close to a star), he can assume that there is a certain core of closely interconnected programs in the product, which use a lot of secondary libraries. Consequently, this core is the attraction center for implanting the malevolent code by the intruder.

4.5 Paint.Net

The product is extremely popular with IT-specialists from different spheres, who need a graphic editor (developers, designers, scientists, etc.), because of its minimalism, features and high speed of operation. It is installed by the standard path "C:\Program Files\paint.net\". In terms of size it occupies a position between small and intermediate (among the considered products) – 100 Mb, with 321 files (129 of which are executable). As it will be demonstrated later, this product is quite interesting from the point of view of Expert's analysis from the information security perspective.

The interaction of the product programs presents 2 unconnected graphs, which are shown in Figure 5 and Figure 6.

The Expert's main conclusions are as follows. Firstly, the division of interactions into 2 graphs tells us that either there are two separate sub-products in the product, or that the architecture can be logically divided into 2 layers. Our more detailed analysis proved that the second option is correct.

For example, in the first graph (see. Figure 6)) the central node-program is *ucrtbase.dll* (dynamic link library file of *Microsoft Visual C++*), which imports a lot of other auxiliary libraries (with prefixes *api-ms-win-code*). This graph can be interpreted as the layer of the product's interaction with the OS. Thus, most interactions of the product with the OS will take place via it, and it is critical in terms of malevolent leakage or data modification.

The second graph (see. Figure 7)) does not have a central node, though the following nodes can be equivalently pseudo-central: *msvcp140.dll*, *msvcp140_2.dll*, *vcruntime140.dll*, *vcruntime140_1.dll*, *PaintDotNet.SystemLayer.Native.x86.dll*, *PaintDotNet.SystemLayer.Native.x64.dll*. The analysis of execution logic should start from them. The graph should be interpreted as the layer of the business logic of the product.

Secondly, we can notice that the *api-ms-win-crt* prefix are quite often used by the others; the Expert should pay particular attention to them.

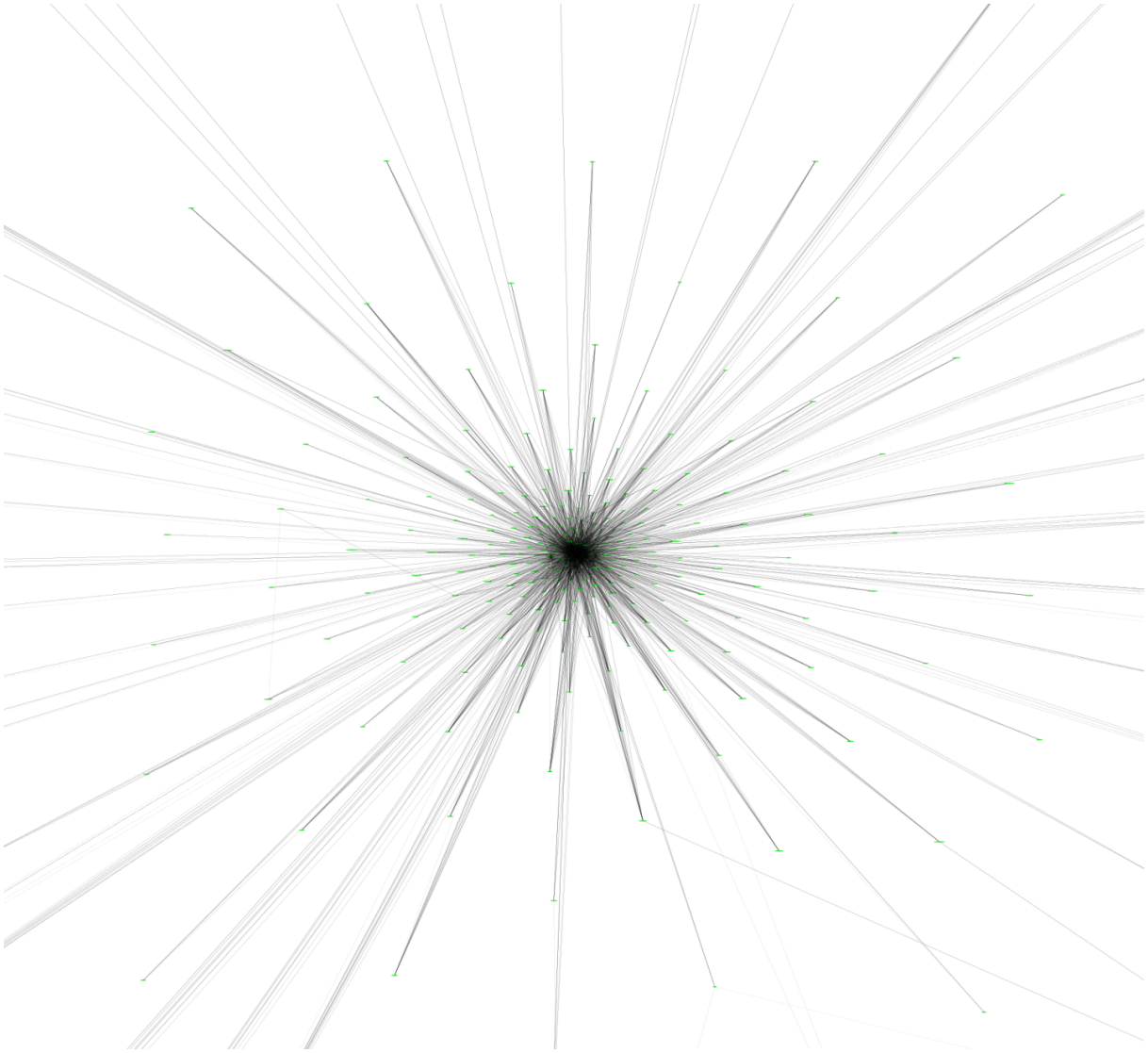


Figure 5: Visualization of interactions for the product Microsoft Office Professional Plus 2019

An important conclusion to be made from the examples of visualization of interactions of various software products is that the obtained graphic image suits well for the Expert's analysis only when the size is small, and sometimes – when the size is intermediate. In the general case we have to develop the algorithms, processing the graphs of links and providing the Expert with meta-information (as the information based on the already collected information), which will increase its efficiency of searching for the weak spots in the products.

5 Algorithms of analysis

Based on the example of visualization of various products, as well on the authors' vast experience in the field of information security of software systems, we can offer the following algorithms, which will allow us to automatize the Expert's manual efforts aimed at visual auditing of inter-software connections. This is supposed to increase the Expert's efficiency: analysis time will be reduced, demands to qualifications

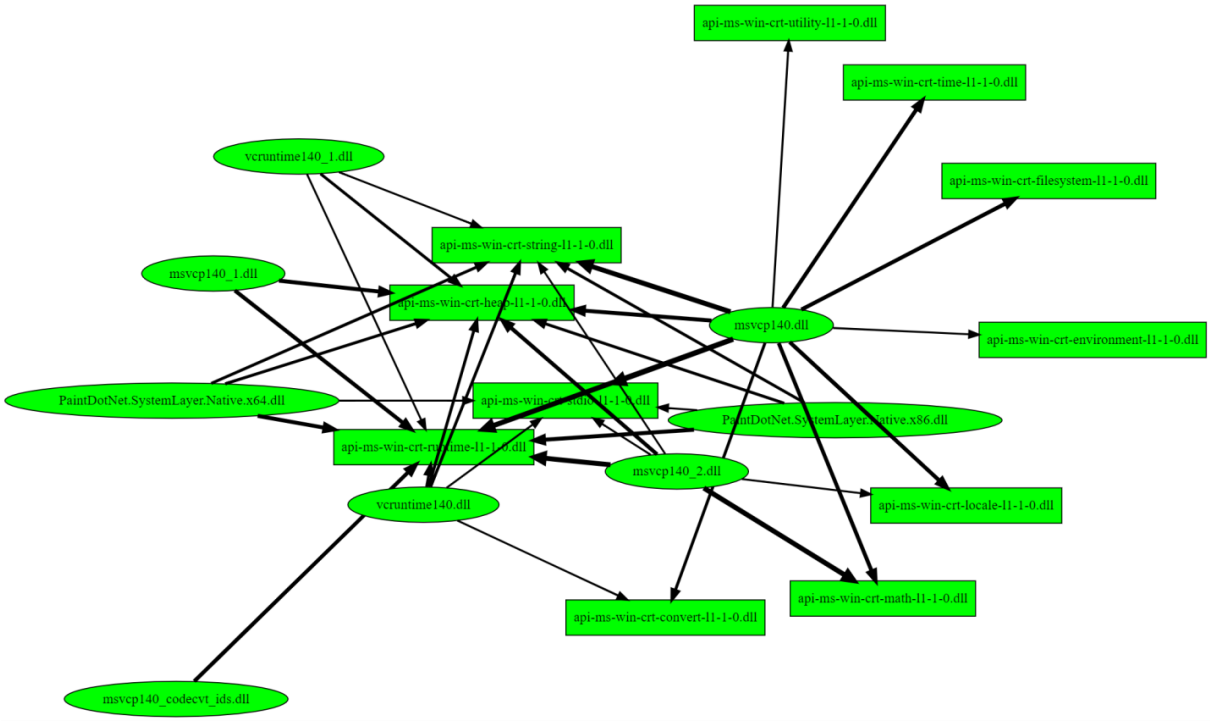


Figure 6: Visualization of interactions for the product Paint.Net (part 1)

will be reduced, more potentially weak spots will be revealed. It is important to point out that, although the basic one was the work of the Prototype supporting only PE-file with direct import, the algorithms can be used for analysing any types of programs and interactions.

We are going to describe briefly 8 algorithms, the first 5 of which have been compiled on the basis of the analysis of visualization examples, while the source of the last 3 is the previous authors' experience. As a matter of fact, each algorithm is a part of the procedure *IntelligentAlgorithms()* from the algorithmic code method (Algorithm 2).

Algorithm 2: IntelligentAlgorithms

Input:
Model – Model of program interaction
OptionsForAlgs – Options for algorithms
Output:
MetaInfo – Meta information with results of intelligent algorithms

```
begin
    MetaInfo.Alg3 = Algorithm_3(Model, OptionsForAlgs);
    MetaInfo.Alg4 = Algorithm_4(Model, OptionsForAlgs);
    ...
    MetaInfo.Alg10 = Algorithm_10(Model, OptionsForAlgs);
    return MetaInfo;
end
```

The algorithm at the input takes the model, built by the algorithm of the main branch of the method (*VisualizationMethod*), as well as some options related to the work of intelligent algorithms. At the

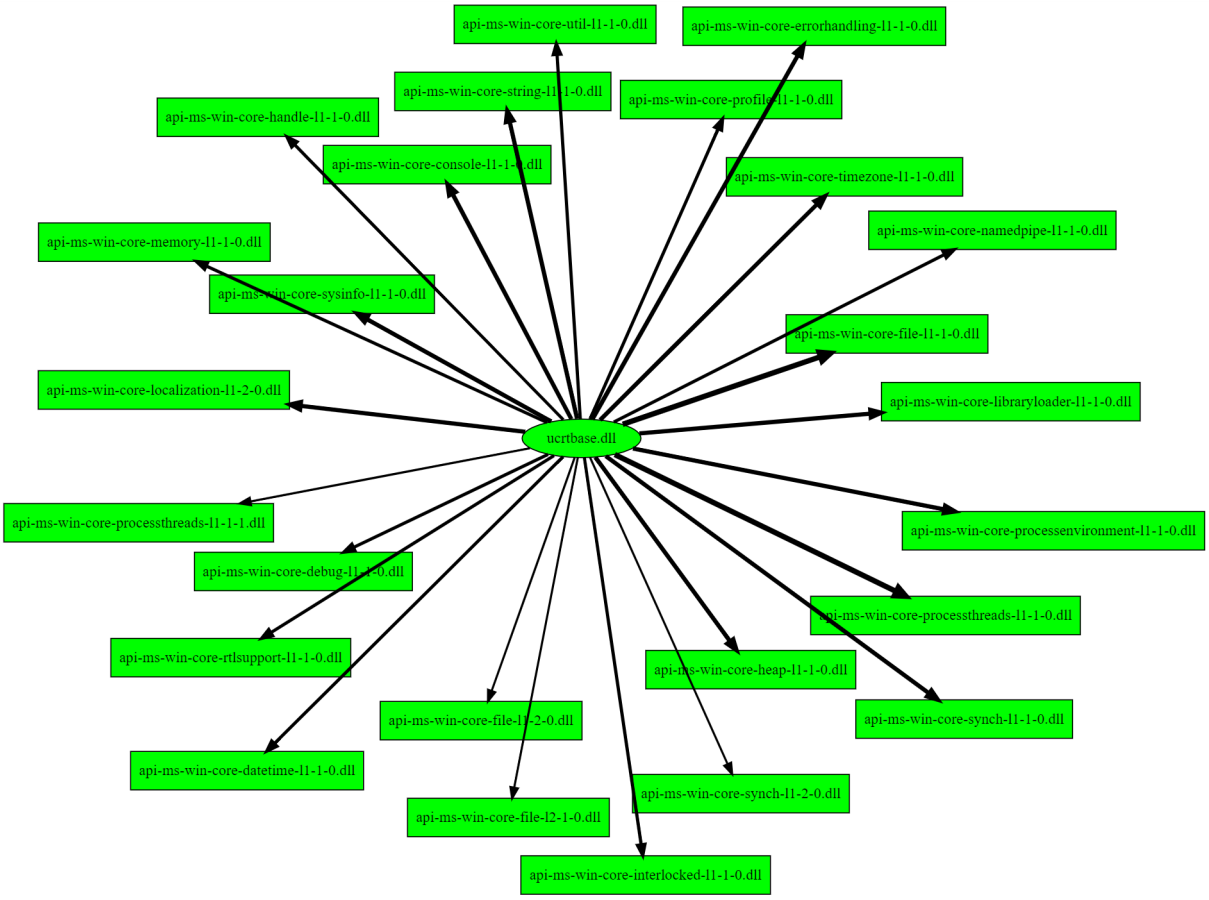


Figure 7: Visualization of interactions for the product Paint.Net (part 2)

output meta-information, obtained as a result of the work of all algorithms, is returned. The algorithm body consists of the consecutive call of each intelligent algorithm, while the results of their work are stored in the retrieved structure.

5.1 Detection of program clusters

The algorithm is aimed at detecting logically connected programs, which can be interpreted as the architectural module of the product [40, 34].

The principle of the algorithm's work is based on using machine learning methods with respect to clusterization.

In terms of information security, it will allow us to detect high-level interactions between clusters [36], "the suspiciousness" of which can give us a hint at the architectural vulnerabilities of the product.

Algorithmically, the algorithm has the following code – Algorithm 3.

The algorithm at the input takes the model, built by the algorithm of the main branch of the method (*VisualizationMethod*), as well as some options related to its work. At the output the interactions between various clusters of programs return. The body of the algorithm consists of using machine learning with respect to clusterization for detecting the groups of related programs. Following that the information about all interactions of programs in different clusters is collected. This information is recorded in the retrieved structure.

Algorithm 3: DetectionOfProgramClusters

```
Input:
Model – Model of a program interaction
OptionsForAlgs – Options for algorithms
Output:
Relations – All relations between clusters of a program interaction
begin
    List<Relations> Relations;
    Clusters = MachineLearning.Clustering(Model.Graph);
    foreach ClusterFrom in Clusters do
        foreach ClusterTo in Clusters do
            if ClusterFrom == ClusterTo then
                continue;
            end
            Relations12 = FindRelations(ClusterFrom, ClusterTo);
            Relations.Add(Relations12);
        end
    end
    return Relations;
end
```

5.2 Detection of several unconnected sub-graphs of interactions

The algorithm is aimed at determining the programs, whose interactions take place only within the framework of separate groups. The algorithm differs from the previous one by more rigid terms, because Algorithm 3 presupposes that such groups must have occasional interactions with each other, too.

The principle of the algorithm's operation is based on coloring the nodes of the graphs [38], in such a way that all the nearest nodes should be the same color. This will ensure coloring the groups in different colors.

In terms of information security the data of the group will denote two things: subdivision into internal sub-products – which will require using the same analysis approaches, as to a separate product, and division into architectural layers – the analysis of the assignment of which (for example, by program composition) will be a sign of the alternative (compared to the modular type) architecture division and will add its own peculiarities with regard to information security.

Algorithmically, the algorithm has the following code – Algorithm 4.

The algorithm at the input takes the model, built by the algorithm of the main branch of the method (*VisualizationMethod*), as well as some of the options related to its work. At the output the descriptions of the sub-graphs of programs in the model, not interacting with each other, are retrieved. The body of the algorithm consists of coloring the nodes of the graphs in such a way that all the nearest nodes should be the same color. After that all the programs connected with the nodes of the same color are united into sub-graphs and retrieved as a list.

5.3 Detecting the anomalies of outgoing interactions

The algorithm is aimed at detecting the nodes-programs, which are widely interacting with the others.

The principle of the algorithm is based on two approaches. Firstly, the direct comparison of the number of outgoing edges (i.e. the ones with a direction) with a certain extreme value set by the Expert;

Algorithm 4: DetectionOfSeveralUnconnectedSubGraphsOfInteractions

```
Input:
Model – Model of a program interaction
OptionsForAlgs – Options for algorithms
Output:
SubGraphs – Sub-graphs at graph of a program interaction
begin
    List<Graph> SubGraphs;
    NumOfColors = MarkNearNodesBySameColor(Model.Graph);
    for Color = 1 .. NumOfColors do
        Nodes = FindNodesByColor(Color);
        Graph = CreateGraph(Nodes);
        SubGraphs.Add(Graph);
    end
    return SubGraphs;
end
```

an anomaly occurs when this value is exceeded [19, 14]. Secondly, the use of machine learning methods with respect to anomalies for detecting an excessive number of outgoing edges.

From the information security perspective such nodes with anomalies can be interpreted as in points of the product, and the study of program execution security must start with them.

Algorithmically, the algorithm has the following code – Algorithm 5 (using the first approach (i.e. based on the extreme value).

Algorithm 5: DetectingTheAnomaliesOfOutgoingInteractions

```
Input:
Model – Model of a program interaction
OptionsForAlgs – Options for algorithms
Output:
AnomalyNodes – Nodes with anomaly outcome relations
begin
    List<Node> AnomalyNodes;
    foreach Node in Model.Nodes do
        if Node.OutcomeRelations.Count > MAX_OUTCOME_RELATIONS then
            AnomalyNodes.Add(Node);
        end
    end
    return AnomalyNode;
end
```

The algorithm at the input takes the model, built by the algorithm of the main branch of the method (*VisualizationMethod*), as well as some of the options related to its work. At the output the nodes which have an anomalously high number of interactions with other programs are retrieved. The body of the algorithm consists of visiting the nodes of the graphs of the model and checking whether the number of outgoing connections is exceeded or not. All the nodes satisfying this condition are included in the retrieved structure.

5.4 Detecting the anomalies of incoming interactions

The algorithm is aimed at detecting nodes-programs, with which the others interact a lot.

The principle of the algorithm is based on the same approaches as Algorithm 5 – the comparison with the peak value of the incoming ones and the detection of anomalies in a similar manner.

From the information security perspective, the nodes are of special interest for the Expert, because their vulnerability can put the work of all other nodes-programs at risk (using them for information exchange).

Algorithmically, the algorithm has the following code – Algorithm 6.

Algorithm 6: DetectingTheAnomaliesOfIncomingInteractions

```
Input:
Model – Model of a program interaction
OptionsForAlgs – Options for algorithms
Output:
AnomalyNodes – Nodes with anomaly income relations
begin
    List<Node> AnomalyNodes;
    foreach Node in Model.Nodes do
        if Node.IncomeRelations.Count > MAX_INCOME_RELATIONS then
            AnomalyNodes.Add(Node);
        end
    end
    return AnomalyNode;
end
```

The algorithm at the input takes the model built by the algorithm of the main branch of the method (*VisualizationMethod*), as well as some of the options related to its work. At the output all the nodes with which other programs have an anomalously high number of interactions are retrieved. The body of the algorithm consists of visiting all the nodes of the graphs of the model and checking whether the number of incoming connections is exceeded or not. All the nodes satisfying this condition are included in the retrieved structure.

5.5 Determining the characteristics by external interactions

The algorithm is aimed at compiling a set of characteristics for each node in terms of assigning the nodes interacting with it (including external programs and, in particular, system libraries of the OS). For instance, if one program interacts with the libraries of logging and work with the network, and the second one – with the libraries of logging and authentication, then we can claim that the first one can hypothetically write down the network exchange data in the log, (for example, the user's messages), and the second one – the data of the account (for example, the username and the password). In this case the Expert's analysis vector, obviously, will be different.

The principle of the algorithm's operation is based on selecting the categories of programs (network, authentication, etc.), defining the degree of including nodes-programs to these categories by interactions with the corresponding programs, adjustments of these degrees in case of indirect interactions, etc.

From the information security perspective, this information will provide the Expert with the opportunity to determine the vector of the characteristics of each node-program, which will allow us to restore functional and goal-oriented connections in the product [21].

Algorithmically, the algorithm has the following code – Algorithm 7.

Algorithm 7: DeterminingTheCharacteristicsByExternalInteractions

```
Input:
Model – Model of a program interaction
OptionsForAlgs – Options for algorithms
Output:
NodesMetric – Hash container with nodes and metric of interacting programs
begin
    Hash<Node, List<Category>> NodesMetric;
    foreach Node in Model.Nodes do
        if Node.Program.IsPartOfProduct then
            foreach NodeTo in Nodes.OutcomeNodes do
                if !NodeTo.Program.IsPartOfProduct then
                    CategoryId = GetCategoryId(NodeTo.Program);
                    NodesMetric[Node][CategoryId] ++;
                end
            end
        end
    end
    return NodesMetric;
end
```

The algorithm at the input takes the model, built by the algorithm of the main branch of the method (*VisualizationMethod*), as well as some of the options related to its work. At the output the hash is returned, in which the key is the node of the model graph, and the value – the list of external programs (for example, the system of libraries) for different categories, with which the program interacts in this node. There is a list of numbers in the hash value, and each of these numbers equals the program counter of a certain category. The algorithm body consists of visiting all the nodes of the model graph, with which program products are associated. For each such node-program the programs are visited not from the product, with which it is connected in the graph. After that, the category of non-product programs is determined, according to which the counter is incremented, which is connected with the product program. So, as a result, a certain array of counters turns out to be connected with each program from the product. This array presents a certain program metrics and is included in the retrieved hash.

5.6 Determining the cycles and their quantity

The algorithm is aimed at revealing the cyclic character of program interaction. In some cases (mainly, for direct import) it can be a sign of incorrect product design – due to incorrect division of features into programs [18].

The working principle of the algorithm is based on using standard techniques of detecting cycles in the graphs, using depth-first traversal and coloring the nodes.

From the information security perspective, such an anomalous topology of interactions can be a sign not only of certain “blunders” made by developers, but also of the possibility of the intruder’s implementing the malevolent code. The last fact may be caused by the intruder (not being a developer) having no deep knowledge about the architecture of the product and not always being able to implement his code correctly. The number of cycles in this case will be a signal of the degree of disrupting architecture.

Algorithmically, the algorithm has the following code – Algorithm 8.

Algorithm 8: DeterminingTheCyclesAndTheirQuantity

```
Input:
Model – Model of a program interaction
OptionsForAlgs – Options for algorithms
Output:
HasLoops – Boolean flag of loops at program interaction
Loops – List of loops in model of a program interaction
begin
    Loops = DetectLoops(Model.Graph);
    HasLoops = (Loops.Count != 0);
    return (HasLoops, Loops);
end
```

The algorithm at the input takes the model, built by the algorithm of the main branch of the method (*VisualizationMethod*), as well as some of the options related to its work. At the output the flag indicating the presence of cycles returns to the graph of the model, as well as to the nodes of these cycles. The algorithm of defining cycles in the graph is a standard one and is realized by the double coloring of the nodes – during traversal in the direct and opposite directions. The flag of the presence of the cycles and their composition are included in the retrieved structure.

5.7 Determination of trusted routes of interactions

The algorithm is aimed at determining the characteristics of the paths of interactions (routes), passing through the programs of various degrees of trust (similar to [25]).

The working principle of the algorithm is based on searching for all possible paths from one node to the other one with the subsequent check of the trustworthiness degree of their nodes [3].

From the information security perspective, the algorithm can be used as follows. Based on his analysis, the Expert can determine (or consider from the very beginning) some of the nodes-programs as secure, i.e. trusted [28]. The others, unknown and unstudied programs will initially be considered potentially dangerous – i.e. untrusted. Thus, we can follow the path of the interaction of one program with the other one – for example, the user's window with the network library. If all interaction routes pass only through trusted programs, the potential threat of the account's leakage will be reduced. Otherwise, if some paths go through untrusted programs, the degree of the current threat can be assessed. This information will allow the Expert to find the most critical spots in the product, requiring his paramount attention.

Algorithmically, the algorithm has the following code – Algorithm 9.

The algorithm at the input takes the model, built by the algorithm of the main branch of the method (*VisualizationMethod*), as well as some of the options related to its work – the pair of nodes, between which trusted paths have to be found. At the output the number of paths between the two nodes of the model graph, going through the trusted nodes, is retrieved. The body of the algorithm consists of visiting all the model graph nodes and determining the fact of trusting the programs associated with them (for example, using the database or the results of statistical analysis of vulnerabilities [30]). Using that, subsequently, the search for the trusted paths into the graph is conducted (for example, the Dijkstra's algorithm) between the two nodes obtained from the options of the algorithm. The number of trusted paths is written down in the retrieved structure.

Algorithm 9: DeterminationOfTrustedRoutesOfInteractions

```
Input:
Model – Model of a program interaction
OptionsForAlgs – Options for algorithms (first and last program for analysis of trust path)
Output:
NumOfPath – Number of trusted paths
begin
    Hash<Node, Bool> NodesTrust;
    foreach Node in Model.Nodes do
        IsTrust = GetTrust(Node.Program);
        NodesTrust[Node] = IsTrust;
    end
    TrustedPath = FindTrustedPath(Model.Graph, NodesTrust, OptionsForAlgs.NodesPair);
    NumOfPath = NumOfPath.Count;
    return NumOfPath;
end
```

5.8 Classification of programs in terms of maliciousness

The algorithm is aimed at including the product programs to the classes pointing at the malicious nature of their work [44, 5, 15].

The working principle of the algorithm is based on using machine learning with regard to classification (for example, similar to [9]). For instance, the pairs of interacting programs can be used as features, as well as the number (or in some cases the composition – as called and calling functions) of interactions between them. The database of known malevolent programs, implemented in various products can be used as the training sample, or, alternatively, the Expert's experience obtained in the course of his analysis.

Algorithmically, the algorithm has the following code – Algorithm 10.

The algorithm at the input takes the model, built by the algorithm of the main branch of the method (*VisualizationMethod*), as well as some of the options related to its work – training data for machine learning. At the output the class, into which this product was included, is retrieved. The algorithm body consists of nested traversal of the nodes of the model graph and obtaining the list of pairs of programs. Within the pair the first program is a part of the product, while the second one – is not (for example, as a system library). Following that, a machine learning object (i.e. the model is created), which is trained on the data from the options of the algorithm. Finally, the current product is classified according to the list of the pairs of programs. The resulting class (in the simplest case – licensed or malevolent software) is included in the retrieved structure.

From the information security perspective, the algorithm enables us to include all the programs to certain classes denoting the proximity to this or that malevolent sample.

5.9 Comparison of algorithms

Each algorithm is designed to solve its own problem, while having certain properties. Let us compare the proposed algorithms using the following five criteria for evaluating universal properties:

- Criterion 1 – the applicability of machine learning methods, which means following modern trends and advances in the field of intelligent systems, which have advantages over the rules developed

Algorithm 10: TheClassificationOfProgramsInTermsOfMaliciousness

```
Input:
Model – Model of a program interaction
OptionsForAlgs – Options for algorithms (input and output data for Machine Learning)
Output:
Class – Class of software product (by machine learning)
begin
    List<Program, Program> Test;
    foreach Node in Model.Nodes do
        if Node.Program.IsPartOfProduct then
            foreach NodeTo in Nodes.OutcomeNodes do
                if !NodeTo.Program.IsPartOfProduct then
                    Test.Add(Node.Program, NodeTo.Program);
                end
            end
        end
    end
    MachineLearning.Teach(Options.MLClsInput, Options.MLClsOutput);
    Class = MachineLearning.Classification(Test);
    return Class;
end
```

manually by experts based on their own experience: ”+” – the need for application, ”+/–” – possible use, ”–” – non-use.

- Criterion 2 – the ability to directly search for malicious code, which corresponds to the extent to which the algorithm allows you to search for such code without additional post-processing: ”+” – direct search, ”+/–” – indirect search (i.e. determination of some of the features of such a code), ”–” to get meta information that will help in the search.

- Criterion 3 – the level of automation corresponding to the degree of the Expert’s involvement in setting and managing the algorithm’s operation: ”+” – for full-fledged work without the participation of the Expert, ”+/–” – for the need for some adjustment of the parameters, ”–” – for the need to adjust the key points of work.

- Criterion 4 – simplicity of implementation, the opposite of the complexity of bringing the algorithm to a full-fledged solution: ”+” – if there are ready-made components for work, ”+/–” – if it is necessary to refine existing components, ”–” – if it is necessary to conduct additional research and development of components from scratch.

- Criterion 5 – comprehensibility of the application from the point of view of an Expert, the opposite of his required qualifications: ”+” – with sufficient general knowledge in information security, ”–” – if necessary to have some experience in the analysis of interactions of the software in terms of information security, ”–” – with the need to be a security specialist for complex software.

The values of all criteria are chosen so that the algorithm with the ”+” value outperforms the other algorithm with the ”–” value. This will allow you to enter the following scores for meeting the criteria:

1 point – for ”+”, 0.5 point – for ”+/–” and 0 points – for ”–”. Thus, it will be possible to summarize the points awarded for each algorithm, thereby making their quantitative comparison.

Let us give a comparative analysis of the algorithms from the point of view of the introduced criteria (Table 2).

Algorithms	**Applicability**	**Direct search of malware**	**Work automation**	**Ease of implementation**	**Comprehensibility for an Expert**	**Points**
Algorithm 3	+	+/–	–	+/–	+/–	2.5
Algorithm 4	–	–	+	+/–	+/–	2.0
Algorithm 5	+/–	+/–	+/–	+	–	2.5
Algorithm 6	+/–	+/–	+/–	+	–	2.5
Algorithm 7	+/–	+	+/–	–	+	3.0
Algorithm 8	–	–	+	+	–	2.0
Algorithm 9	–	+/–	–	+/–	+/–	1.5
Algorithm 10	+	+	+/–	–	–	2.5
Total points	3.5	4	4	4.5	2.5	18.5

Table 2: Comparative analysis of algorithms according to the entered criteria

The results of the analysis allow us to assert (see the column ”Points” in Table 2) that all algorithms are assigned points in the range from 1.5 to 3.0, while the majority of them have scores 2.0 (2 algorithms) and 2.5 (4 algorithms), and the rest (by 1 algorithm) are 0.5 points from the minimum and maximum boundaries of the main ones. Therefore, it can be argued that, in general, the algorithms are fairly equivalent. A deeper analysis of the particular values of their criteria suggests that each of the algorithms has its own strengths (”+”) and weaknesses (”–”) sides.

A similar situation with a similar distribution is observed for the total scores of programs for each of the criteria (see the line ”Total scores” in Table 2) – the values for the criteria range from 2.5 to 4.5, which for all algorithms and criteria gives an average value $\frac{18.5}{(8\times5)} \sim 0.5$ = ”+/–”. Consequently, the joint (i.e. integrated) application of the algorithms will theoretically have averaged values according to the criteria, which is undoubtedly a good result.

6 Discussion

As it was, to a certain degree, mentioned before, the visualization method (and, consequently, the Prototype) has a number of disadvantages.

Firstly, the visualization of big products significantly reduces the opportunities of the Analysis Expert. However, the proposed analysis algorithms are aimed at simplifying this type of analysis – by automatization and selecting new meta-information. Moreover, other visualization types, in addition to 2D-graph can be used – for example, with the help of smart scaling, specialized tools of navigation in the graph, 3D-models [43], augmented reality, etc.

Secondly, the Prototype and the obtained examples of visualization have been provided only for PE-files, excluding the other program types. However, the aim of our article was to describe the very approach to the analysis of software products, as well as the development of algorithms, making the Expert’s work easier. For other types of programs this picture will not have any quantitative differences.

Thirdly, for the full-scale work of the method the support of much more complicated types of interactions is needed, requiring syntactic and semantic analysis of the executable code. In some cases, the

full or partial de-compilation of the latter will be required. However, the authors are interested in this problem, too, and the ways to solve it will be described in their future articles.

And, finally, the use of the algorithms is not always a sign of an information security breach in a software product. Nevertheless, this will all the same increase the efficiency of detecting threats, because it will make the Expert's work easier, and the Expert, as a rule, has limited personal time, psychic and mental resources. The first fact follows from the idea that the updates of the product can appear more often than when the Expert's analysis is finished (because a deep analysis of the machine code might be needed, and this takes several months). The second fact – from the general limited availability of information security experts with sufficient qualifications and experience.

7 Conclusion

This article is devoted to the method of visualizing the interaction between various software types, which will enable the Expert to audit software products. This is quite an important information security task. Special attention is given to the development of intelligent algorithms of interactions analysis, which will simplify the Expert's work significantly.

The main idea of the method lies in the analysis of statistical connections between programs and building the corresponding model, which suits both for the application of algorithms and for the final visualization.

Our future research will be devoted to the following topics. Firstly, the extension of the analysis of algorithms and their practical realization in the Prototype. Secondly, the support of new types of programs and interactions. Thirdly, the assessment of the efficiency of the method and the Prototype, as well as their practical application during the analysis of the OS in general and software products, in particular.

Acknowledgement

This research is being supported by the grant of RSF #21-71-20078 in St. Petersburg Federal Research Center of the Russian Academy of Sciences (SPC RAS).

References

[1] H. AlBreiki and Q. Mahmoud. Evaluation of static analysis tools for software security. In *Proc. of the 10th International Conference on Innovations in Information Technology (IIT'14), Al Ain, United Arab Emirates*, pages 93–98. IEEE, November 2014.

[2] M. Almorsy, J. Grundy, and A. Ibrahim. Automated software architecture security risk analysis using formalized signatures. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13), San Francisco, California, USA*, pages 662–671. IEEE, May 2013.

[3] S. Alqahtani. Enhancing trust – software vulnerability analysis framework. In *Proc. of the IEEE International Conference on Software Testing, Verification and Validation (ICST'17), Tokyo, Japan*, pages 563–564. IEEE, March 2017.

[4] A. Amine, B. Mohammed, and L. Jean-Louis. Generating control flow graph from java card byte code. In *Proc. of the 3th IEEE International Colloquium in Information Science and Technology (CIST'14), Tetouan, Morocco*, pages 206–212. IEEE, October 2014.

[5] V. Atluri. Malware classification of portable executables using tree-based ensemble machine learning. In *Proc. of the SoutheastCon (SoutheastCon'19), Huntsville, Alabama, USA*, pages 1–6. IEEE, April 2019.

[6] R. Bearden and D. Lo. Automated microsoft office macro malware detection using machine learning. In *Proc. of the 2017 IEEE International Conference on Big Data (Big Data'17), Boston, Massachusetts, USA*, pages 4448–4452. IEEE, December 2017.

[7] S. Bieliauskas and A. Schreiber. A conversational user interface for software visualization. In *Proc. of the 2017 IEEE Working Conference on Software Visualization (VISSOFT'17), Shanghai, China*, pages 139–143. IEEE, September 2017.

[8] M. Buinevich and K. Izrailov. Method and utility for recovering code algorithms of telecommunication devices for vulnerability search. In *Proc. of the 16th International Conference on Advanced Communication Technology (ICACT'14), Pyeongchang, Republic of Korea*, pages 172–176. IEEE, February 2014.

[9] M. Buinevich and K. Izrailov. Identification of processor's architecture of executable code based on machine learning. part 1. frequency byte model. *Proceedings of Telecommunication Universities*, 6(1):77–85, March 2020.

[10] M. Buinevich, K. Izrailov, and G. Ganov. Intellectual method of programs interactions visualization for information security audit of the operating system. In *Proc. of the 12th Majorov International Conference on Software Engineering and Computer Systems (MICSECS'20), Saint Petersburg, Russia*, volume 1, pages 1–12. CEUR Workshop Proceedings, December 2020.

[11] M. Buinevich, K. Izrailov, and A. Vladyko. The life cycle of vulnerabilities in the representations of software for telecommunication devices. In *Proc. of the 18th International Conference on Advanced Communication Technology (ICACT'16), PyeongChang, Republic of Korea*, pages 430–435. IEEE, January – February 2016.

[12] M. Buinevich, K. Izrailov, and A. Vladyko. Metric of vulnerability at the base of the life cycle of software representations. In *Proc. of the 20th International Conference on Advanced Communication Technology (ICACT'18), Chuncheon, Republic of Korea*, pages 1–8. IEEE, February 2018.

[13] Z. Chen, L. Chen, and B. Xu. Hybrid information flow analysis for python bytecode. In *Proc. of the 11th Web Information System and Application Conference (WISA'14), Tianjin, China*, pages 95–100. IEEE, September 2014.

[14] K. Dhambri, H. Sahraoui, and P. Poulin. Visual detection of design anomalies. In *Proc. of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08), Athens, Greece*, pages 279–283. IEEE, April 2008.

[15] A. Elkhawas and N. Abdelbaki. Malware detection using opcode trigram sequence with svm. In *Proc. of the 26th International Conference on Software, Telecommunications and Computer Networks (SoftCOM'18), Split, Croatia*, pages 1–6. IEEE, September 2018.

[16] P. Ferrara, A. Cortesi, and F. Spoto. Cil to java-bytecode translation for static analysis leveraging. In *Proc. of the IEEE/ACM 6th International FME Workshop on Formal Methods in Software Engineering (FormaliSE'18), Gothenburg, Sweden*, pages 40–49. IEEE, May – June 2018.

[17] F. Fittkau, A. Krause, and W. Hasselbring. Exploring software cities in virtual reality. In *Proc. of the IEEE 3rd Working Conference on Software Visualization (VISSOFT'15), Bremen, Germany*, pages 130–134. IEEE, September 2015.

[18] M. Goldstein and I. Segall. Automatic and continuous software architecture validation. In *Proc. of the IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE'15), Florence, Italy*, volume 2, pages 59–68. IEEE, May 2015.

[19] E. Guimaraes, A. Garcia, E. Figueiredo, and Y. Cai. Prioritizing software anomalies with software metrics and architecture blueprints. In *Proc. of the 5th International Workshop on Modeling in Software Engineering (MiSE'13), San Francisco, California, USA*, pages 82–88. IEEE, May 2013.

[20] M. Ijaz, M. Durad, and M. Ismail. Static and dynamic malware analysis using machine learning. In *Proc. of the 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST'19), Islamabad, Pakistan*, pages 687–691. IEEE, January 2019.

[21] M. Jang, H. Kim, and Y. Yun. Detection of dll inserted by windows malicious code. In *Proc. of the 2007 International Conference on Convergence Information Technology (ICCIT'07), Gwangju, Republic of Korea*, pages 1059–1064. IEEE, November 2007.

[22] H. Jeong, J. Baik, and K. Kang. Functional level hot-patching platform for executable and linkable format binaries. In *Proc. of the 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC'17),*

Banff, Alberta, pages 489–494. IEEE, December 2017.

[23] K. Karppinen, M. Lindvall, and L. Yonkwa. Detecting security vulnerabilities with software architecture analysis tools. In *Proc. of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08), Lillehammer, Norway*, pages 262–268. IEEE, April 2008.

[24] S. Kim, J. Park, K. Lee, I. You, and K. Yim. A brief survey on rootkit techniques in malicious codes. *Journal of Internet Services and Information Security (JISIS)*, 3(4):134–147, November 2012.

[25] I. Kotenko and E. Doynikova. Security assessment of computer networks based on attack graphs and security events. In *Proc. of the 2nd Information and Communication Technology - EurAsia Conference (ICT-EurAsia'14), Bali, Indonesia*, volume 8407 of *Lecture Notes in Computer Science*, pages 462–471. Springer Berlin Heidelberg, April 2014.

[26] I. Kotenko, I. Saenko, and A. Branitskiy. Applying big data processing and machine learning methods for mobile internet of things security monitoring. *Journal of Internet Services and Information Security (JISIS)*, 8(3):54–63, August 2018.

[27] I. Kotenko, I. Saenko, and A. Kushnerevich. Parallel big data processing system for security monitoring in internet of things networks. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 8(4):60–74, December 2017.

[28] A. Krasov and S. Shterenberg. Methods for building a trusted environment in unix operating systems based on the implementation of a digital watermark. In *Proc. of the 12th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT'20), Brno, Czech Republic*, pages 253–257. IEEE, October 2020.

[29] K. Lee, H. Yeuk, Y. Choi, S. Pho, I. You, and K. Yim. Safe authentication protocol for secure usb memories. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 1(1):46–55, June 2010.

[30] L. Merino, J. Fuchs, M. Blumenschein, C. Anslow, M. Ghafari, O. Nierstrasz, M. Behrisch, and D. A. Keim. On the impact of the medium in the effectiveness of 3d software visualizations. In *Proc. of the 2017 IEEE Working Conference on Software Visualization (VISSOFT'17), Shanghai, China*, pages 11–21. IEEE, September 2017.

[31] L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz. Cityvr: Gameful software visualization. In *Proc. of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME'17), Shanghai, China*, pages 633–637. IEEE, September 2017.

[32] K. Mikhailovich, M. Valerievna, P. Andreevich, U. Alexandrovich, and K. Vladimirovich. Guidelines for using machine learning technology to ensure information security. In *Proc. of the 12th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT'20), Brno, Czech Republic*, pages 285–290. IEEE, October 2020.

[33] S. Moon, B. Park, and R. Kim. Sw visualization framework for safe software. In *Proc. of the 2016 International Conference on Platform Technology and Service (PlatCon'16), Jeju Island, Republic of Korea*, pages 1–5. IEEE, February 2016.

[34] F. Morsali and M. Keyvanpour. Search-based software module clustering techniques: A review article. In *Proc. of the IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI'17), Tehran, Iran*, pages 0977–0983. IEEE, December 2017.

[35] A. Radwan. Machine learning techniques to detect maliciousness of portable executable files. In *Proc. of the 2019 International Conference on Promising Electronic Technologies (ICPET'19), Gaza City, Palestine*, pages 86–90. IEEE, October 2019.

[36] S. Sarkar, A. Kak, and N. Nagaraja. Metrics for analyzing module interactions in large software systems. In *Proc. of the 12th Asia-Pacific Software Engineering Conference (APSEC'05), Taipei, Taiwan*, page 8. IEEE, December 2005.

[37] A. Schreiber and M. Brüggemann. Interactive visualization of software components with virtual reality headsets. In *Proc. of the 2017 IEEE Working Conference on Software Visualization (VISSOFT'17), Shanghai, China*, pages 119–123. IEEE, September 2017.

[38] A. Shukl and M. Garg. A list based approach to solve graph coloring problem. In *Proc. of the International Conference on System Modeling Advancement in Research Trends (SMART'18), Moradabad, India*, pages

265–267. IEEE, November 2018.

[39] H. Shukla, S. Patil, D. Solanki, L. Singh, M. Swarnkar, and H. K. Thakkar. On the design of supervised binary classifiers for malware detection using portable executable files. In *Proc. of the IEEE 9th International Conference on Advanced Computing (IACC'19), Tiruchirappalli, India*, pages 141–146. IEEE, December 2019.

[40] A. Smith. Cluster-based tangible programming. In *Proc. of the 4th International Conference on Digital Information and Communication Technology and its Applications (DICTAP'14), Bangkok, Thailand*, pages 405–410. IEEE, May 2014.

[41] B. Tahtaci and B. Canbay. Android malware detection using machine learning. In *Proc. of the 2020 Innovations in Intelligent Systems and Applications Conference (ASYU'20), Istanbul, Turkey*, pages 1–6. IEEE, October 2020.

[42] A. Teyseyre and M. Campo. An overview of 3d software visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(1):87–105, July 2008.

[43] J. Vincur, P. Navrat, and I. Polasek. Vr city: Software analysis in virtual reality environment. In *Proc. of the IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C'17), Prague, Czech*, pages 509–516. IEEE, July 2017.

[44] Q. Wei, S. Xiao, and L. Dongbao. Malware classification system based on machine learning. In *Proc. of the 2019 Chinese Control And Decision Conference (CCDC'19), Nanchang, China*, pages 647–652. IEEE, June 2019.

[45] C. Woody. Security engineering risk analysis (sera). In *Proc. of the IEEE/ACM 3rd International Workshop on Software Engineering Research and Industrial Practice (SER&IP'16), Austin, Texas, USA*, pages 23–24. IEEE, May 2016.

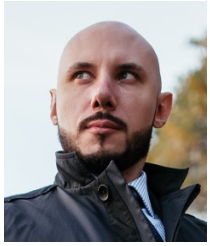
[46] J. Xue, C. Hu, K. Wang, R. Ma, and Z. Fan. Design of software security detection system based on system call sequences. In *Proc. of the 2nd International Conference on Software Engineering and Data Mining (SEDM'10), Chengdu, China*, pages 97–100. IEEE, June 2010.

Author Biography

Mikhail Buinevich graduated as a military electronics engineer in 1980. He served in the navy and state information security agencies. He has extensive experience in practical and design work in this area. He taught at various universities, and prepared over 10 PhD. His research interests are cybersecurity, Internet of things, vehicle-to-everything, reverse engineering, and machine learning. He has over 150 scientific papers, including patents. She is the editor-in-chief of the scientific journal "Proceedings of Telecommunication Universities", as well as a guest and topic editor for the journals of the MDPI publishing house. Currently, Dr. Buinevich is a professor at St. Petersburg University of State Fire Service of Emercom of Russia and The Bonch-Bruevich St. Petersburg State University of Telecommunications.

Konstantin Izrailov graduated from St. Petersburg State Polytechnic University, Physical and Mechanical Department in 1996. He received PhD degree from the Bonch-Bruevich St. Petersburg State University of Telecommunications in 2017. Currently, he is an associate professor at the St. Petersburg State University of Telecommunications and Senior Researcher of the St. Petersburg Federal Research Center of the Russian Academy of Sciences. He has about 70 published articles and 5 patents. He is also the Deputy Editor-in-Chief of the university journal. His scientific interests include information security, search of vulnerabilities in machine code, reverse engineering, telecommunication devices, machine learning, intelligent algorithms, Internet of things, UEFI/BIOS, Smart City.

Igor Kotenko graduated with honors from St. Petersburg Academy of Space Engineering and St. Petersburg Signal Academy. He obtained the Ph.D. degree in 1990 and the National degree of Doctor of Engineering Science in 1999. He is Professor of computer science and Head of the Laboratory of Computer Security Problems of the St. Petersburg Federal Research Center of the Russian Academy of Sciences. He is the author of more than 500 refereed publications, including 14 textbooks and monographs. Igor Kotenko has a high experience in the research on computer network security and participated in several projects on developing new security technologies. For example, he was a project leader in the research projects from the US Air Force research department, via its EOARD (European Office of Aerospace Research and Development) branch, EU FP7 and FP6 Projects, HP, Intel, F-Secure, etc. The research results of Igor Kotenko were tested and implemented in more than 50 Russian research and development projects.



Pavel Kurta graduated from the St. Petersburg Baltic State Technical University in 2006. He also underwent retraining under the Presidential Management Retraining Program at the Polytechnic University in St. Petersburg in 2017. He studied at the Bonch-Bruyevich St. Petersburg University of Telecommunications in graduate school from 2017 to 2020. His research interests include system analysis, psychoanalysis and psychology, finance, software development, and the issues of user interaction systems. He is actively published in domestic and foreign scientific journals.