Approach to analyzing executable code based on the software architecture recovery

D.S. Kononov <dspr2@yandex.ru>
Federal State Unitary Enterprise «18 CSRI», Ministry of Defence of RF,
4 Svobodny prospect, Moscow, Russia, 111123

Abstract. The article discusses a new approach to obtaining additional information about the software module under study based on the preliminary software architecture recovery during the executable code analysis. As a result, it is possible to reduce the requirements for the resources spent by limiting the field of research, rational choice of priorities, and abstraction from secondary elements. The paper demonstrates the feasibility of restoring the software architecture in a two-step process: first, the separate components are isolated, and then their purposes and relationships are determined. An automated method for decomposing a software module is proposed, which allows allocating components corresponding to static libraries, classes, and their groups. This method is based on the functions clustering by the distances between them in the address space and on the call graph. A description of the implementation of the developed method as a plug-in for the IDA disassembler is given.

Keywords: executable code analysis; software architecture; clustering; call graph; distance between functions; software module; decomposition

DOI: 10.15514/ISPRAS-2018-30(5)-4

For citation: Kononov D.S. Approach to analyzing executable code based on the software architecture recovery. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 5, 2018, pp. 75-88. DOI: 10.15514/ISPRAS-2018-30(5)-4

1. Introduction

The need to analyze the executable code is widely encountered in practice when addressing issues related to the protection of intellectual property, the search for software backdoors and vulnerabilities, the analysis of computer viruses, certification of compilers, and software development. It should be noted that despite all the achievements in this area, the task in question is still far from being solved. This is due to the fundamental limitations expressed in the extreme complexity of formalizing and automating the analysis of executable code (Каушан, Маркин, Падарян, & Тихонов, 2013). If, when searching for vulnerabilities, there are still effective automatic methods (fuzzing, symbolic execution), then, when restoring the executable code algorithm, the result of the study is largely determined by the quality of expert analysis. In this case, automation is limited to local signature or statistical methods that facilitate the search for constants or a special set of

instructions in specific algorithms. Particularly acute shortcomings of the modern scientific and methodological apparatus appear when it is impossible to use dynamic analysis methods. Taking into account the volume of the software modules under study, which in the case of embedded systems reach tens and hundreds of thousands of functions, the solution of practical tasks in this area requires significant investments of both material and time resources and the availability of substantial human capital.

Thus, it becomes necessary to provide an expert with information about a software module, important for achieving a positive result. As practice shows, first of all, an expert needs to understand the logic of the software module (Quist & Liebrock, 2009), since it allows targeted search through "reverse engineering" (Streekmann, 2012, стр. 27)p. 27]. In this case, the model of the expert's work changes: instead of the direct restoration of the algorithms implemented in the program code, the assumptions regarding their implementations are confirmed and specific parameters are determined.

One of the important components of the "reverse engineering" approach is the restoration of the software architecture, information about which is lost during compilation. Software architecture (Streekmann, 2012, стр. 9)p. 9] can be described in the framework of a hierarchical model of the structure of complex systems (Косяков & Свит, 2014).

2. Hierarchical model of software

The development of complex software requires high-quality software architecture with well-defined systems and subsystems that solve a specific problem and have weak coupling (Макконнелл, 2010, стр. 96). Modern non-specialized programming languages clearly support such programming paradigm (Microsoft Corp., 2006). When examining a software module, it is possible to break it up into components that correspond to the initially programmed systems and subsystems (for example, using source code or debugging information (Ebert, Riediger, & Winter, 2008), (Bohnet & Dollner, 2006)). For definiteness, one can designate such selected subsets of a software module as components, and the process of breaking up a software module – decomposition. According to the model used, the set of components is hierarchical, in which the components located at the upper level consist of a combination of the underlying ones. As part of the research, the components classification is proposed, presented in Table 1.

Table 1. Classification of components

No.	Name	Description
1	Software	A component that fully incorporates the software under investigation. Always present in a single instance.
2	File	A separate file in the corresponding file system included in the software. For embedded systems without a file system, all firmware is considered as a single file.

3	Embedded operating system	For systems with multiple processors, there can be several embedded operating systems running in parallel within the same address space.
4	Static library	For software modules (including embedded operating systems) the presence of built-in libraries is typical. For example, OpenSSL encryption libraries, various drivers, libraries of standard functions (memcpy, strlen), etc. Components of a similar size (500-3000 functions) that have a weak connection (for example, only using API) with the rest of the code also belong to this level.
5	Class group	This level corresponds to a group of classes in the object-oriented programming terminology. Typical size from 100 to 1000 functions. Differ from level 4 in a greater connectivity with the rest of the code. For example, various classes that implement the same interface (plug-ins), network protocol handler, etc.
6	Class	This level corresponds to a class in the object-oriented programming terminology or an object (compiled) module in the C language. They have a size of up to 100 functions. Differ from level 5 in a greater connectivity with the rest of the code. For example, the implementation of a circular buffer, hash tables, etc.
7	Function	Currently, the task of the allocation of automating functions from executable code is solved and implemented in modern disassemblers at a sufficient level.
8	Logical block in a function	The part of the function consisting of basic units designed to solve a subtask. For example, inline functions, condition checking, loops, etc.
9	Basic unit	A sequence of instructions without transitions automatically isolated by modern disassemblers (cycle body, the condition being checked, etc.).
10	Logically isolated sequence of instructions	Part of the basic unit, designed to solve some subtasks. For example, loading data from memory, untwisted cycle, etc.
11	Instruction	Executable machine instruction. Automatically isolated by modern disassemblers.
12	Instruction argument	The executable machine instruction argument. Automatically isolated by modern disassemblers.

In the modern scientific and methodological apparatus for analyzing executable code, the information only about levels 1, 2, 3, 7, 9, 11, 12 is used due to the absence of additional debugging data (Meng & Miller, 2016). The existing significant gap between levels 3 and 7 makes it necessary to analyze software modules consisting of tens and hundreds of thousands of functions using methods

that have exponential computational complexity (fuzzing, character execution, etc.) (Каушан, Маркин, Падарян, & Тихонов, 2013). To overcome the existing limitations, it is necessary to take into account levels 4, 5, 6 (Streekmann, 2012), which will significantly reduce the requirement for the resources needed to conduct the software module investigation (Quist & Liebrock, 2009). Thus, existing methods with high computational complexity can be scaled for software modules with a volume of more than 10,000 functions due to their decomposition into components with a characteristic size falling within the range of effective application of the corresponding methods. For embedded systems, this process is, in fact, analogous to isolating dynamic libraries and programs from the general-purpose operating system (OS) and examining them separately. At the same time, to implement the considered approach, it is necessary that the separated integral parts have a specific isolated functionality, that is, they would be components according to the terminology adopted in the paper.

There are a large number of software architecture definitions (Clements, Bachmann, & Bass, 2010, стр. 27), (Ian, 2011, стр. 20). The conceptual apparatus of the research is based on the IEEE 1471 standard (ANSI/IEEE Standart 1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems, p. 9), p. 9]: software architecture is the fundamental organization of the system, embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution. Thus, from the definition, it follows that the restoration of the disassembled software module architecture (Streekmann, 2012, стр. 30)p. 30] should be carried out in two stages. The first is the decomposition of the disassembled software module into levels 3-6 components, and the second is the determination of the functionality of the selected components and the restoration of their relationships with each other and with the environment.

3. Analysis of the executable code, taking into account the software architecture

In the modern scientific and methodological apparatus, the main element of the research is the function, which leads to the need to analyze and restore the algorithms of a large number of interrelated functions to determine their common purpose. In contrast, the preliminary decomposition of a software module allows determining the role of a specific component in the architecture by analyzing only a few of its functions (in some cases just one) or the data and strings used in it. As a result, based on the described approach, a reasoned conclusion is made on the assignment of hundreds and thousands of functions that form the corresponding component by analyzing a small amount of data. Furthermore, additional information about the purpose of the components is provided by an analysis of their relationships.

Knowledge of the software architecture makes it possible to rationally prioritize the research within the framework of solving a specific practical problem. For example,

if it is necessary to restore the network interaction protocol of the botnet, then the emphasis in the study should be placed on the appropriate handler. One of the features of this approach to restoring the executable code algorithm is the ability to limit the study of functions from non-priority components to the conceptual level. As an example, one can cite the situation when the algorithm of the bootloader is investigated and the component of interaction with flash memory is isolated. In this case, one should not restore the entire algorithm for writing or reading flash memory, but logically assign the values "write" or "read" to the component functions called from the bootloader. In addition, the joint analysis of a single software module by several experts is significantly simplified due to the rational differentiation of the studied areas into separate components.

Information about the software module architecture is also required when conducting dynamic analysis. Thus, the lack of information about the components used significantly complicates the analysis of execution routes and slicing (Падарян, Гетьман, & д.р., 2014). Even when examining programs for the Windows OS family under the x86 architecture, it is difficult to draw an unequivocal conclusion about the algorithm being performed and its purpose without separating the called functions by the system API. In the case of embedded OS («firmware»), this problem is only aggravated.

In the framework of «fuzzing», it is impossible to correctly emphasize its direction without knowing the architecture of the software module. The cases of work only within one component from the study of their entirety are indistinguishable. The availability of information about the software architecture makes it possible to rationally select the area of study, excluding components that are not interesting in the current context. This leads to the possibility of multiple reductions in computational costs. For example, by isolating the component of working with strings or with memory, one can prevent loop traversal in the functions of copying memory or comparing strings, replacing them with appropriate heuristics.

Despite all the advantages described, in modern scientific and methodological apparatus for analyzing executable code in the absence of debugging data, there are no effective automated methods not only for restoring the software architecture but also for decomposing a software module into components. As a result, such an approach is not used in practice in the overwhelming majority of cases, since the time and cost of resources do not pay off in the current realities. This situation significantly limits the ability to analyze executable code.

4. Method of the software module decomposition

As the applied methods for decomposing a software module into components, apart from the expert one, one can single out various modifications of the task of finding *strongly connected components* (Новиков, 2009, стр. 283) of the call graph and various imaging techniques. However, due to limited disassembling capabilities (Meng & Miller, 2016), low call graph density, and the presence of widely used functions (for example, working with strings and memory) that are called from

almost anywhere, there are low informative results that do not allow the software module decomposition (Streekmann, 2012, стр. 52, 96). Dynamic analysis methods simplify this process insignificantly (Quist & Liebrock, 2009), but they themselves are not applicable in the general case and work for relatively small amounts of code due to the coverage problem (Каушан, Маркин, Падарян, & Тихонов, 2013), (Падарян, Гетьман, & д.р., 2014). In this regard, similar approaches to the analysis of executable code are practically not used, although they are quite widespread for the source code (Bohnet & Dollner, 2006), (Kienle & Muller, 2010).

At the same time, the experience of software research has shown that often interrelated functions are located nearby in the address space. This arrangement is explained both by the optimization for the hardware architecture and by the simplification of compiler development. For example, with paging memory, a speed gain occurs when finding jointly called functions within a single page (Eagle, 2011, стр. 114). On the other hand, the simplest implementation of the linker involves the sequential addition of object modules (Bryant & O'Hallaron, 2016, стр. 672), and shuffling the functions between them implies some optimization. As a rule, an object module corresponds to a separate source code file (a class in object-oriented programming) and, therefore, is a component by definition in software with a well-developed architecture. Thus, to carry out the decomposition of the software module, it is proposed to perform clustering of functions based on distances both in the address space and on the call graph. It should be noted, however, that the interrelated functions are absolutely not obliged to be located near each other in the address space, but these cases are associated either with a significant level of optimization or with the use of some protection measures (for example, small granular randomization of address space allocation during compilation (Нурмухаметов, Курмангалеев, Каушан, & С.С., 2014)).

As the distance between two functions in the address space, it is proposed to use the number of positions enclosed between them in the list of functions sorted by starting address. Such a choice is explained by the need to eliminate the dependence of the distance on the size of the functions and the data placement order. However, it should be taken into account that there is a certain selected size of a component of a certain level (for example, 1000-3000 functions in the case of static libraries), and, at the same time, it is necessary to consider the interaction of all functions in the software module under study. Based on these prerequisites, in order to obtain the final distance in the address space, an increasing step function was chosen corresponding to the estimated sizes of the components at various levels (see Table 1).

$$d(f_i, f_j) = l(j - i)$$
$$l(x) = \begin{cases} 0, x = r_0 = 0 \\ k_1, 0 < x < r_1 \\ k_2, r_1 \le x < r_2 \\ \dots \\ k_K, r_{K-1} \le x < \infty \end{cases} \quad (1)$$
$$k_{i+1} > k_i; r_{i+1} > r_i$$
$$k_i, r_i \in \mathbb{N}$$

where f_i is a function with a sequence number i in the list of functions, sorted in order of increasing starting addresses; k_i – step function coefficient for the i-th range; r_i – limit of the i-th range; K – the number of ranges in the step function.

Then each edge of the call graph is assigned a weight equal to the distance $d(f_i, f_j)$ between the functions in the address space (1). As a result, the distance between functions on the call graph is defined as the minimum sum of edge weights that form the path from one function to another. It is necessary to clarify that the call graph, in this case, is considered as an undirected graph, that is, there is a path in the graph from the calling function to the called one, and vice versa. It should also be borne in mind that with an arbitrary choice of the step function, it is possible that the distance calculated from the call graph will be less than the corresponding value of the step function. The simplest example of this kind is

$$l(x) = \begin{cases} 0, x = 0 \\ 1, x = 1 \\ +\infty, x > 1 \end{cases} \quad (2)$$

To eliminate this inconsistency, it is necessary to ensure that the selected step function satisfies the condition: the coefficient value for each interval of the step function must be less than or equal to the sum of the coefficient values for the previous interval and the minimum possible edge weight (i.e., the coefficient of the first interval). Indeed, consider the first point of any interval and draw an edge to it from the previous point (located in the previous interval, respectively). Then if the difference between adjacent intervals is greater than the minimum coefficient of the step function, then the length of the edge from the origin to the selected point will be greater than the length of the path through the immediately preceding point. The formally described condition can be expressed by the formula (3).

$$k_{i+1} \le k_i + k_1 \quad (3)$$

Since the weight of the edge in this problem is non-negative, it is possible to use the Dijkstra algorithm to find the distances between all the functions (Кормен, Лейзерсон, Ривест, & Штайн, 2013, стр. 595)p. 595]. Then the computational complexity of finding the distances from the current function to all the others will be $O(n^2 + m)$, where n is the number of nodes (functions) on the call graph, and m is the number of edges (calls). Given that clustering requires the calculation of the distance matrix between all functions, then the total computational complexity will be $O(n^3 + nm)$. Using the binary heap in Dijkstra's algorithm can reduce the

computational complexity to $O(n^2 \log n + nm \log n)$. At the same time, it is necessary to consider that $m = O(n^2)$. However, in practice, the call graphs of real programs are strongly sparse: the density of call graphs for all checked software modules with volumes from a few hundred to tens of thousands of functions tends to 0. The latter is explained by the decrease in connectivity between subsystems with increasing software scale. The calculated values of the ratio m/n (the average number of edges per node) for the studied software modules did not exceed 4 and tends to decrease with an increase in the module volume. Consequently, in the analysis of the executable code, the relationship for the call graph is satisfied $m = O(n)$ and the evaluation can be used for the computational complexity of constructing the distance matrix $O(n^2 \log n)$.

In the framework of the experiments, it was found that the parameters of the step function can be specified considering the expected sizes of the components within fairly wide limits. Thus, the proposed method for decomposing a software module is robust. Additionally, this conclusion is confirmed by the fact that the experiments were conducted under the conditions of the existing limitations of modern means of disassembling (IDA software) to restore the call graph. As a result, one can conclude that the information about the original software architecture is stored in the executable code and can be restored.

5. Practical implementation

Currently, interactive clustering is implemented based on the creation of a heat map for the distance matrix. In the distance matrix, functions are sorted in ascending order of their starting addresses. In this case, no additional computational costs are required apart from converting the calculated matrix into a graphic image in the BMP format. An example of such an image ("code map") is shown in Fig. 1; the darker the point, the smaller the distance between the functions.

Interactive clustering is performed by manually selecting rectangular blocks visually different from adjacent areas. As a result, a hierarchical structure is formed on the "code map" consisting of a number of square blocks located on the diagonal, which either do not intersect or are nested in another block. This structure corresponds to the software architecture of the module under study, and the diagonal blocks themselves corresponds directly with the components of different levels. Blocks outside the diagonal determine the degree of interaction between the components. Also, for additional confirmation of the decision on the correctness of the components selection and the initial assessment of their assignment, strings and other data, which are used in the functions from the block selected on the "code map", are automatically displayed on request for the operator.

The following optimizations are added during implementation:

1) individual disconnected components of the call graph are excluded if the number of nodes is less than the threshold (the recommended value is 20);

2) individual disconnected call graph components having a diameter (Новиков, 2009, стр. 249)p. 249] less than the threshold are excluded (the recommended value of 3, excluding graphs with the star topology);
3) excluded functions that are called only from a single function and do not call anything;
4) springboard functions (mediating long-distance calls) are excluded from the distance matrix by signature, but are taken into account when constructing routes;
5) stub functions and imported functions are excluded.

The program for calculating the distance matrix and interactive clustering is implemented as a plug-in for the IDA disassembler. The minimum input data is the call graph and start addresses of functions, which allows analyzing the executable modules for unsupported IDA processors upon independent receipt of the specified data. Information about the selected components is stored in the IDB file and is used to display functions in the form of a tree-like list, similar to that used when displaying projects in modern integrated software development environments.

The PC with average computing capabilities was used as a test bench: dual-core processor with a frequency of 3.1 GHz (Core i3 2100), 8 GB RAM, SSD drive. In the study of software modules of up to 10,000 functions, the calculation and construction of the "code map" takes place within a minute. Such delays are insignificant in the context of the study of the program code for the operator. The applied step distance function is given by the formula 4:

$$l(x) = \begin{cases} 0, x = 0 \\ 1, 0 < x < 100 \\ 2, 100 \le x < 400 \\ 3, 400 \le x < 800 \\ 4, 800 \le x < 1600 \\ 5, 1600 \le x \end{cases} \quad (4)$$

The experiments performed using the example of Nmap software version 7.10 x86 for Windows OS[1] (Nmap.org, 2016) showed that the selected components in the executable code correspond to specific subsystems and classes in the source code (fig. 1). In addition, the dependence of the isolation degree of the components on their level and on the size of the software module was confirmed, which is fully consistent with the need to improve the quality of the software architecture while increasing the size and complexity of the project. In turn, the high quality of the latter is provided mainly by strengthening the cohesion of the components and weakening the coupling between them.

It should be noted that in the process of decomposition, the specific features of the software module (including those introduced by the compiler) are revealed, the information on which allows simplifying and automating the study of the executable

[1] nmap.exe module disassembled in IDA Pro 7.0 environment contains 3436 functions, 3082 functions were allocated after the use of heuristics.

code. Due to the additional analysis, functions of the main cycles, standard service functions of working with memory and strings, error handling functions, springboard functions, designed to link the high-level components to each other are determined.

6. Areas for further research

For the full implementation of the approach to analyzing the executable code considered in the article, it is necessary to develop automatic methods for restoring the software architecture, making it possible to explore the entire existing range of sizes of software modules. In addition, these methods should be universal both in terms of hardware architecture and the technologies, languages, and programming paradigms used. To achieve these properties, it is required to work out the issues of the restoration of components interconnections and effective hierarchical clustering, including the case of random allocation of address space.

In the near future, the proposed method for decomposing a software module will be developed. It is planned to implement automatic clustering methods; to take into account the relationship graph of functions based on the using data in addition to the call graph; to perform software modules classification based on the characteristics that affect the decomposition process (hardware architecture, programming paradigms, code size, etc.); to optimize the parameters of the step function of the distance for the extracted classes of modules.

It should be noted that the result is influenced by the quality of the call graph recovery and, accordingly, the improvement of this indicator is also one of the directions of the described approach development.

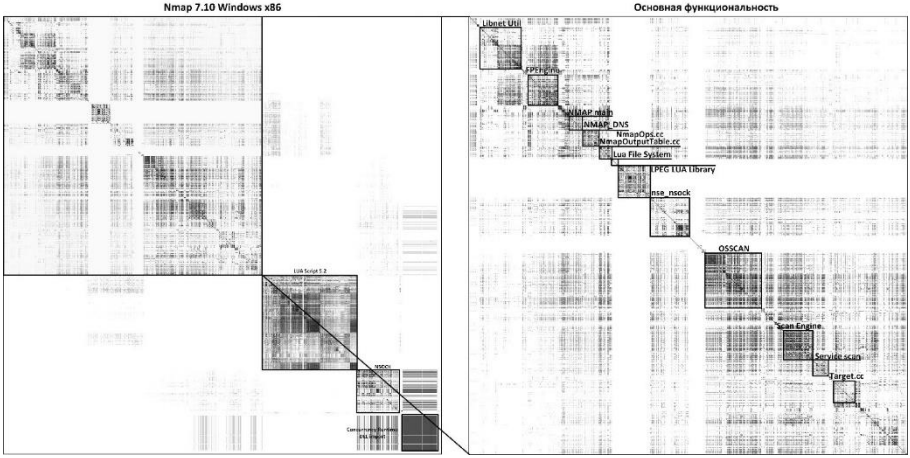

Fig. 1 – Code map of the Nmap.exe software module version 7.10 x86 for Windows OS after contrast correction. The components corresponding to the classes and their groups in the source code are partially labeled. Small parts are not displayed due to scale limitations

7. Conclusion

Knowledge of the software architecture makes it possible to significantly reduce the requirements for consumed resources during the analysis of executable code by limiting the field of research, rational choice of priorities, abstraction from secondary elements, and joint analysis. As a result, the software module under study is divided into separate components with a characteristic size of several thousand functions, which, in fact, leads to a decrease in the dimension of the original problem. Moreover, there is an additional way of expansion of the obtained intermediate results of the analysis for the entire software module. To achieve the indicated advantages, it is necessary to restore the software architecture of the executable code.

It is proposed to carry out this process in two stages: 1) decomposition of the disassembled software module into separate components; 2) the definition of the functionality of the selected components and their relationships. To perform the first stage, an automated method has been developed that allows selecting components corresponding to static libraries, classes, and their groups. This method is based on the functions clustering by the distances between them in the address space and on the call graph. Currently, interactive heat map clustering for the distance matrix is implemented as a plug-in for the IDA disassembler. The conducted experiments confirmed the possibility of restoring the software architecture only by the software module, which made it possible to demonstrate in practice the advantages of the approach to the analysis of executable code considered in the article.

References

[1]. Kaushan V.V., Markin Yu.V., Padaryan V.A., Tikhonov A.Yu. Methods for Finding Errors in a Binary Code. Technical Report. ISP RAS, Moscow, 2013 (in Russian).

[2]. Quist D.A., Liebrock L.M. Visualizing Compiled Executables for Malware Analysis. Proc. of the International Workshop on Visualization for Cyber Security (VisSec09), 2009, pp. 27-32.

[3]. Streekmann N. Clustering-Based Support for Software Architecture Restructuring. Springer, 2012, 241 p.

[4]. Kosyakov A., Svit U. Systems Engineering. Principles and Practice. 2nd ed. Moscow: DMK Press, 2014, 624 p. (in Russian).

[5]. McConnell S. Code Complete. Workshop. 2nd ed. Moscow: Russian edition, 2010, 896 p. (in Russian).

[6]. Microsoft Corp. ECMA-334 C# Language Specification. Ecma International. 2006. Available at: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf, accessed 13.10.2017.

[7]. Ebert J., Riediger V., Winter A. Graph Technology in Reverse Engineering. The TGraph Approach, Proc. of the 10th Workshop Sowtware Reengineering (WSR 2008), vol. 126, 2008, pp. 67-81.

[8]. Meng X., Miller B.P. Binary Code Is Not Easy. Proc. of the 25th International Symposium on Software Testing and Analysis (ISSTA16), Saarbrucken, Germany, 2016, pp. 24-35.
[9]. Clements P., Bachmann F., Bass L. et al. Documenting Software Architectures: Views and Beyond. 2nd ed. Addison-Wesley Professional, 2010, 517 p.
[10]. Ian G. Essential Software Architecture, 2nd ed. Springer, 2011, 242 p.
[11]. ANSI/IEEE Standard 1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems.
[12]. Padaryan V.A., Getman A.I. et al. Methods and Software Supporting the Combined Analysis of a Binary Code. Programming and Computer Software, vol. 40, issue 5, 2014, pp. 276–287.
[13]. Novikov F.A. Discrete Mathematics for Programmers: Textbook for Universities, 3rd ed. Piter, 2009, 384 p. (in Russian).
[14]. Bohnet J., Dollner J. Visual Exploration of Function Call Graphs for Feature Location in Complex Software Systems. Proc. of the 2006 ACM Symposium on Software Visualization, 2006, pp. 95-104.
[15]. Kienle H.M., Muller H.A. Rigi – An Environment for Software Reverse Engineering, Exploration, Visualization and Redocumentation. Science of Computer Programming, vol. 75, issue 4, 2010, pp. 247-263.
[16]. Eagle C. IDA Pro Book, 2nd ed. No Starch Press, 2011, 672 p.
[17]. Nurmukhametov A.R., Zhabotinsky E.A., Kurmangaleev S.F., Gaisaryan S.S., Vishnyakov A.V. Fine-grained address space layout randomization on program load. . Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017, pp. 163-182 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-9
[18]. Bryant R.E., O'Hallaron D.R. Computer Systems: A Programmer's Perspective. 3rd ed. Pearson, 2016, 1084 p.
[19]. Nurmukhametov A.R., Kurmangaleev S.F., Kaushan V.V., Gaisaryan S.S. Compiler protection techniques against software vulnerabilities exploitation. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 3, 2014, pp. 113-126 (in Russian). DOI: 10.15514/ISPRAS-2014-26(3)-6
[20]. Kormen T. Kh., Leyzerson Ch.I., Rivest R.L., Stein K. Algorithms: Construction and Analysis, 3rd ed. Williams LLC, 2013, 1328 p. (in Russian).
[21]. Nmap: the Network Mapper, 2016. Available at: https://nmap.org/dist/nmap-7.10-win32.zip, accessed 21.08.2018.

Подход к анализу исполняемого кода на основе восстановления программной архитектуры

Д.С. Кононов <dspr2@yandex.ru>
ФГУП «18 ЦНИИ» МО РФ,
111123, Россия, г. Москва, Свободный проспект, д. 4.

Аннотация. В статье рассматриваются новый подход к получению дополнительной информации об исследуемом программном модуле на основе предварительного восстановления программной архитектуры в ходе анализа исполняемого кода. В результате появляется возможность сократить требования к затрачиваемым ресурсам за счёт ограничения области исследования, рационального выбора приоритетов, абстрагирования от второстепенных элементов. В работе демонстрируется

осуществимость восстановления программной архитектуры в рамках двухэтапного процесса: вначале проводится выделение обособленных компонентов, а затем определяются их назначения и взаимоотношения. Предлагается автоматизированный метод декомпозиции программного модуля, позволяющий выделять компоненты, соответствующие статическим библиотекам, классам и их группам. Данный метод базируется на кластеризации функций по расстояниям между ними в адресном пространстве и на графе вызовов. Приведено описание реализации разработанного метода в виде плагина для дизассемблера IDA.

Ключевые слова: анализ исполняемого кода; программная архитектура; кластеризация; граф вызовов; расстояние между функциями; программный модуль; декомпозиция.

DOI: 10.15514/ISPRAS-2018-30(5)-4

Для цитирования: Кононов Д.С. Подход к анализу исполняемого кода на основе восстановления программной архитектуры. Труды ИСП РАН, том 30, вып. 5, 2018 г., стр. 75-88 (на английском языке).. DOI: 10.15514/ISPRAS-2018-30(5)-4

Список литературы

[1]. Каушан В.В., Маркин Ю.В., Падарян В.А., Тихонов А.Ю. Методы поиска ошибок в бинарном коде, технический отчет, ИСП РАН, Москва, 2013.
[2]. Quist D.A., Liebrock L.M. Visualizing Compiled Executables for Malware Analysis. Proc. of the International Workshop on Visualization for Cyber Security (VisSec09), 2009, pp. 27-32.
[3]. Streekmann N. Clustering-Based Support for Software Architecture Restructuring. Springer, 2012, 241 p.
[4]. Косяков А., Свит У. Системная инженерия. Принципы и практика. 2-е изд. Москва: ДМК Пресс, 2014. 624 с.
[5]. Макконнелл С. Совершенный код. Мастер-класс. 2-е изд. Москва: Издательство «Русская редакция», 2010. 896 с.
[6]. Microsoft Corp. ECMA-334 C# Language Specification. Ecma International. 2006. URL: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf (дата обращения: 13.Октябрь.2017).
[7]. Ebert J., Riediger V., Winter A. Graph Technology in Reverse Engineering. The TGraph Approach, Proc. of the 10th Workshop Sowtware Reengineering (WSR 2008), vol. 126, 2008, pp. 67-81.
[8]. Meng X., Miller B.P. Binary Code Is Not Easy. Proc. of the 25th International Symposium on Software Testing and Analysis (ISSTA16), Saarbrucken, Germany, 2016, pp. 24-35.
[9]. Clements P., Bachmann F., Bass L. et al. Documenting Software Architectures: Views and Beyond. 2nd ed. Addison-Wesley Professional, 2010, 517 p.
[10]. Ian G. Essential Software Architecture, 2nd ed. Springer, 2011, 242 p.
[11]. ANSI/IEEE Standard 1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems.
[12]. Падарян В.А., Гетьман А.И. и др. Методы и программные средства поддерживающие комбинированный анализ бинарного кода. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 251-276. DOI: 10.15514/ISPRAS-2014-26(1)-8

[13]. Новиков Ф.А. Дискретная математика для программистов: Учебник для вузов. 3-е изд. СПб: Питер, 2009. 384 с.
[14]. Bohnet J., Dollner J. Visual Exploration of Function Call Graphs for Feature Location in Complex Software Systems. Proc. of the 2006 ACM Symposium on Software Visualization, 2006, pp. 95-104.
[15]. Kienle H.M., Muller H.A. Rigi – An Environment for Software Reverse Engineering, Exploration, Visualization and Redocumentation. Science of Computer Programming, vol. 75, issue 4, 2010, pp. 247-263.
[16]. Eagle C. IDA Pro Book, 2nd ed. No Starch Press, 2011, 672 p.
[17]. Нурмухаметов А.Р., Жаботинский Е.А., Курмангалеев Ш.Ф., Гайсарян С.С., Вишняков А.В. Мелкогранулярная рандомизация адресного пространства программы при запуске. Труды ИСП РАН. том 29, вып. 6, стр. 163-182. DOI: 10.15514/ISPRAS-2017-29(6)-9
[18]. Bryant R.E., O'Hallaron D.R. Computer Systems: A Programmer's Perspective. 3rd ed. Pearson, 2016, 1084 p.
[19]. Нурмухаметов А.Р., Курмангалеев Ш.Ф., Каушан В.В., С.С. Г. Применение компиляторных преобразований для противодействия эксплуатации уязвимостей программного обеспечения. Труды ИСП РАН, том 26, вып. 3, стр. 113-126. DOI: 10.15514/ISPRAS-2014-26(3)-6
[20]. Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К. Алгоритмы: построение и анализ. 3-е изд. Москва: ООО «И.Д. Вильямс», 2013. 1328 с.
[21]. Nmap.org. Nmap: the Network Mapper 2016. URL: https://nmap.org/dist/nmap-7.10-win32.zip (дата обращения: 21.08.2018).