Reengineering Modern Industrial Software to Find Vulnerabilities Based on Genetic Algorithms

Konstantin Izrailov
Computer Security Problems Laboratory
St. Petersburg Federal Research Center of the Russian Academy of Sciences
Saint-Petersburg, Russia
konstantin.izrailov@mail.ru

Igor Kotenko
Computer Security Problems Laboratory
St. Petersburg Federal Research Center of the Russian Academy of Sciences
Saint-Petersburg, Russia
ivkote@comsec.spb.ru

Mikhail Buinevich
Department of Applied Mathematics and Information Technologies Security
Saint-Petersburg University of State Fire Service of EMERCOM of Russia
Saint-Petersburg, Russia
bmv1958@yandex.ru

2025 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM) | 979-8-3315-1219-4/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICIEAM65163.2025.11028410

***Abstract*—Modern industry is built, among other things, on software, the presence of vulnerabilities in which is a significant problem. It is more rational to search for vulnerabilities in those representations of the program (source code, algorithms, architecture, etc.) on the basis of which they were developed. However, as a rule, there is difficult-to-analyze machine code available. Obtaining higher-level representations is possible by reverse engineering, carried out in various ways, such as expert, algorithmic, intelligent, enumeration and logging. The qualitative comparison of these representations is given. The current study is devoted to a new method of reverse engineering based on the use of genetic algorithms. The course of the research and the following main scientific results are briefly described: the methodology of reverse engineering of a software system, the model of the life cycle of a program with multi-level vulnerabilities, the concept of genetic de-evolution of program representations, scientific, methodological and algorithmic instrumentation for genetic decompilation, an architectural block for conducting genetic de-evolution of representations with functionality for searching for multi-level vulnerabilities. All results are novel, as well as theoretically and practically significant.**

Keywords—software, vulnerability, information security, reverse engineering, genetic algorithms

I. Introduction

The industrial engineering (IE) area also includes, among other things, ensuring the information security of the data circulating in the system, thereby preventing the implementation of threats to confidentiality, integrity and availability. This requirement is updated by the introduction of cyber-physical systems into industry, in which information is closely related to physical processes that affect both the continuity of processes and human safety. The main means of data transmitting, storing and processing are programs (as an integral part of the software), the presence of vulnerabilities in the code of which leads to the corresponding threats. Thus, the identification and neutralization of vulnerabilities is the most urgent task of the subject area, in the interests of solving which the course and results of the current research will be described below.

One of the current scientific contradictions (i.e. problem) in the field of software security is the following. On the one hand, it is more rational to search for vulnerabilities in the program code according to the representation of the program (source code, algorithms, architecture, etc.) in which they were embedded. On the other hand, for analysis, as a rule, only a low-level executable representation of the program is available – machine code.

Thus, it is necessary to transform machine code into higher-level representations, which, among other things, are suitable for manual analysis by information security experts; this process is classically called reverse engineering. Thus, an intentional distortion of the program architecture by an intruder (for example, an insider in the IT development department) at the algorithm level will have a rather weak reflection, and the analysis of the source or even machine code is highly unlikely to detect this vulnerability. And it is precisely obtaining previous representations of the program from its current (usually machine) that will allow solving the specified scientific problem.

II. Related Work

There are several classical approaches to obtaining higher-level representations of a program from low-level ones, used in the interests of subsequent search for vulnerabilities in them:

(1) the very first approach can be considered a labor-intensive manual analysis of one representation of the program with systematization of the collected information and restoration of another [1]. Thus, even now, experts, analyzing processor instructions (for example, in the IDA Pro product) "assume" the source code from which the machine code of the program could be obtained;

(2) a certain set of decompilation tools with embedded algorithms has been created [2], which automatically transform machine code structures into similar source code structures (using many internal representations of the program, such as an abstract syntax tree, subroutine call graphs, data flow graphs, etc.);

(3) the active introduction of artificial intelligence into all spheres of life has opened up prospects for its application for reverse engineering [3].

(4) as the authors' research has shown, in a number of cases the task of reverse engineering an executable program can be

The work was partially funded by the budget project FFZF-2025-0016.

979-8-3315-1219-4/25/$31.00 ©2025 IEEE

2025 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM) | 979-8-3315-1219-4/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICIEAM65163.2025.11028410

solved by "smartly" enumerating copies of the source code until one is found that exactly compiles into the specified machine code.

(5) a special place among the approaches is occupied by studying the behavior of the program, for example, by intercepting called functions, sent and received messages, or embedding logging [4].

Comparison of each approach from the standpoint of efficiency allows us to obtain the following relative assessments regarding its classical indicators: performance (P), operativeness (O) and resource safekeeping (RS), presented in Table I; the following designations (and their scores) are used: “L” – low (1), “M” – medium (2), “H” – high (3), “L–“ – ultra-low (0); “*” – mark about the authors’ approach, given below; the last column provides the sum of scores for the criteria for each method.

TABLE I. COMPARISON OF THE PERFORMANCE INDICATORS TO APPROACHES TO OBTAINING HIGH-LEVEL REPRESENTATIONS

No.	P	O	RS	Sum.
(1)	H	L	L	5
(2)	M	H	M	7
(3)	M	M	M	6
(4)*	H	L–	L	4
(5)	H	M	H	6

And since (see Table I) to date none of the approaches can be considered satisfactory (since each of them has its own strengths and weaknesses), it is necessary to create a fundamentally new solutions for reverse engineering of programs. A review of scientific publications on the topic of the main problems solved using machine learning models and methods (i.e. classification, regression, clustering, anomaly detection, dimensionality reduction) showed the possibility of their application for typical stages of IE information systems analysis (i.e. information collection, its preparation, processing and visualization) [5]. Other intelligent tools, such as genetic algorithms [6] and genetic programming [7], can be applied in a similar way.

III. METHODOLOGY OF REVERSE SOFTWARE ENGINEERING SYSTEM FOR THE IDENTIFYING VULNERABILITIES PURPOSE

Based on the above, a scientific task was set as the development of theoretical provisions for reverse engineering of programs in the interests of searching for vulnerabilities using artificial intelligence.

Since, as a rule, in IE systems, the program code is contained on physical devices (and not in the form of separate programs, as, for example, in Windows or Linux operating systems), then in order to search for vulnerabilities in the code, it is initially necessary to extract it for subsequent study. In this case, such code can be an entire IE software system with many programs interacting in a complex manner. For this purpose, a corresponding methodology for reverse engineering of a software system (hereinafter referred – Methodology) was created, consisting of the stages of preparation, static and dynamic analysis, and documentation. At the output, the Methodology made it possible to obtain both a list of found vulnerabilities and detailed information about the IE software system for additional search. This Methodology is the first main scientific result of the current authors' research. Almost every step of the Methodology (which constitutes its stages) can be performed by existing or prospective tools [8–10], which justifies the operability of the entire Methodology. Nevertheless, some of the most important steps have not been fully implemented. These are precisely the transformations of program representations, and the current study is devoted to the creation of their theoretical basis.

IV. MODEL OF THE LIFE CYCLE OF A PROGRAM WITH DIFFERENT LEVELS OF VULNERABILITIES

The development of theoretical provisions of reverse engineering of programs from the position of obtaining higher-level representations of IE programs requires understanding the general scheme of transition between these representations; both in the forward direction – i.e. when creating a program, and in the opposite direction – during reverse engineering. In the interests of this, a model of the program life cycle (hereinafter – Model) was obtained, reflecting, among other things, its potential vulnerabilities [11].

This Model is the second main scientific result of the current author's research. The Model allowed us to introduce a new classification of vulnerabilities based on their structural level; as a result, all vulnerabilities can be divided into the following classes: conceptual (in the program concept), architectural, high-level (for example, in architecture), mid-level (for example, in algorithms), low-level (for example, in source and machine code), atomic (for example, in bytecode).

Based on the obtained second main scientific result, as well as its deep study, the principle of de-evolution of program representations was formulated, which consists in the gradual transformation of each current representation into the previous one; in this case, "de-evolution" is understood as a process that is the reverse of the evolution of program development. As a result, it is possible to obtain all previous program representations from the machine code and carry out a highly effective search for vulnerabilities in them.

V. THE GENETIC DE-EVOLUTION CONCEPT OF PROGRAM PEPRESENTATIONS TO VULNERABILITIES SEARCHING

Before directly creating the principles of de-evolution of program representations in IE systems, a number of experiments were conducted on its rougher version, indicated as the 4th approach. For this purpose, a prototype of the method of smart enumeration of source code instances was implemented and studied to obtain one that would exactly compile into a given machine code, i.e. decompilation. Since the implementation of the method is quite technically complex, we will describe only its essence.

Within the framework of smart enumeration, the source code of the program is specified as a path along the syntax rule graph (SRG), corresponding to the programming language of the source code. At the same time, the syntax itself can be specified in a formal form traditional for the field of compiler development [12].

An example of such a graph for a group of expressions with equating variables with different numbers is shown in Fig. 1. Unique node identifiers are indicated in brackets after "#" (they will be used below), and the dotted lines indicate alternative syntax branches for parsing the code.

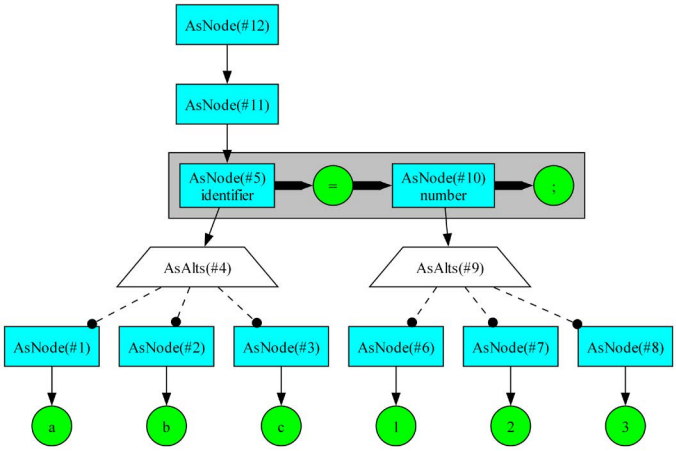

Fig. 1. Example of a syntax rules graph for equating different numbers to variables.

Although the SRG (see Fig. 1) is intuitive and does not require detailed explanations, we will nevertheless give an example of displaying the following source code on it: the expression "b = 3;" will correspond to a path along the graph through the following nodes: #12 → #11 → #5 → #4 → #2 → 'b' → '=' → #10 → #9 → #8 → '3' → ';' . In this case, when traversing the graph to generate code in nodes #4 and #9, one of the alternative branches to the child nodes is selected, which makes the total number of combinations of the source code 3 × 3 = 9. The essence of the smart enumeration method itself is to traverse the SRG along all alternative branches, generate a huge set of source code, compile it into machine code and compare it with the specified one. The identity of two such copies will mean solving the reverse engineering problem.

And although this method is not suitable for directly obtaining the source code for any, no matter how complex, machine code due to its resource and time costs, nevertheless, even it has shown its efficiency in a number of cases. Thus, to obtain the machine code of a non-trivial mathematical expression "z = x * (y – (x / y))" highly qualified experts need 3-5 minutes, and the method (for a limited SRG) does not require much more – 10 minutes.

Certain successes achieved in creating and testing the smart enumeration method allowed us to assume the fundamental possibility of using such approaches for reverse engineering, but through their high-quality optimization. For this, the too trivial enumeration was reduced to solving the optimization problem of selecting the constructions of the previous representation of the program in order to obtain such an instance of it that would be as close as possible (in the limit – coincided) with the program in the current representation. From the standpoint of reverse engineering of the executable program code, this means that it is necessary to select the constructions of the programming language of the source code in such a way that after compilation it would produce the corresponding machine code. The solution to this problem was found by using genetic algorithms with characteristics sufficiently close to the subject area: the genes of an individual correspond to the constructions of the programming language, the chromosome – to the copy of the source code itself, the fitness function shows the proximity of two copies of the machine code (compiled and specified), the formation of new copies of the source code based on the previous ones is carried out due to the operations of selection (selection of those that are compiled into machine code close to the specified one), crossing (mixing the constructions of two source codes) and mutations (random change of program constructions); etc.

The absence of dependencies on representation syntaxes (programming languages, processor instructions) in the algorithms of genetic algorithms allows us to extend the solution not only to de-evolution of machine code into the source code, but also to obtaining any previous representation from the current one. All of the above allows us to talk about the creation of an entire concept (hereinafter referred to as the Concept) of genetic deevolution of representations (hereinafter referred to as GDE).

This Concept is the third main scientific result of the current author's research. The GDE process itself can be described as follows (using the source and machine code as an example): 1) creation of an initial random population of source code instances; 2) compilation of the population into machine code instances; 3) comparison of the obtained instances with the given machine code – a fitness function is applied; 4) selection of those instances that are the closest (after compilation to the given machine code); 5) mixing of the selected source codes – the selection operation; 6) random change of source code structures – the mutation operation; 7) forming a new generation; 8) repeating the scheme from 2). In this case, if the fitness function shows a complete match of the compiled source code with the specified machine code, this will mean a successful solution to the optimization problem, followed by an exit from the process.

A hypothetical experiment on the application of the Concept showed its validity. For example, selecting an expression from two binary operations ("+" and "-") over three operands ("x", "y" and "z") lasted 163 generations, although a complete enumeration of all 6 × 6 × 6 × 6 = 7776 options with their compilation would have taken a significant amount of time. Note that SRG was not used in this experiment, which would have further reduced the number of generations; however, this will be taken into account below.

In order to qualitatively increase the efficiency of vulnerability search in the IE software system (which is the target application of the work), the described Concept can be applied as follows. Despite the fact that, as a rule, a program in the form of executable (machine code) is available for analysis, it is more rational to search for vulnerabilities in higher-level representations, which can be obtained using the Concept. In this case, following the Model, the vulnerability embedded in the program is completely preserved when switching between representations, changing only its form (for example, from a

set of machine instructions to the corresponding set of programming language constructs).

Thus, to search for them, it is necessary to obtain each of the previous representations using the Concept and search for vulnerabilities in them using context-adapted methods [13]. This entire process of sequential restoration of previous representations was called genetic reverse-engineering (hereinafter – GRE).

Based on the operating principles of genetic algorithms and the specifics of their application in the interests of reverse engineering, it is possible to form a necessary instrumentation for the implementation of the Concept. It consists of a method based on a genetic algorithm, a program model in some representation (close, but more "advanced" relative to that used in the smart enumeration method), a number of algorithms and a metric for comparing two code instances in one representation (for calculating the fitness function). And although this instrumentation can be obtained practically invariant from program representations, nevertheless, they will be further considered in the aspect of decompilation, which was logically called genetic - Genetic Decompilation Concept (GDC).

VI. Instrumentation for Genetic Decompilation of Program Representations

For direct practical verification (and not more theoretical, as was done earlier) of the Concept, a scientific-methodical and algorithmic instrumentation (hereinafter – Instrumentation) was developed for GDE of a program from machine code to its source code, which can be used in the interests of ensuring the security of software and cyber-physical devices IE. Note that further, in fact, instead of a machine representation, the corresponding assembler representation is used, since the transformation between them is a well-solvable problem: evolution is carried out by assembling, and de-evolution is carried out by disassembling. The scheme of such GDE partially repeats the solution of the optimization problem within the framework of the Concept and is the following from the point of view of the Instrumentation.

Instrument 1. The core of the scheme is a method for decompiling machine (or assembler) code based on a genetic algorithm modified by adding crosses and mutations of two aspects of the source code – the external form (for example, variable names) and internal content (for example, logical constructions). This method coordinates the work of all other instruments.

Instrument 2. The source code itself, as in the smart enumeration method, is described as a path along the SRG corresponding to the syntax of the programming language (the most traditional one was taken in the study – the C language). At the same time, to "compact" the record and adapt it to the specifics of genetic algorithms, the code record was specified in the form of a chromosome consisting of genes, each of which corresponds to the choice of one of the alternatives. Thus, for the SRG in Fig. 1, the expression "b = 3" will consist of two alternative genes – [2, 3], since when traversing the graph, it is necessary to select the 2nd and 3rd child nodes for the parent nodes #4 and #9.

Instrument 3. To process SRG in the interests of performing operations of crossing and mutation of copies of the source code, as well as a number of other actions, the corresponding support is required, implemented in the form of a set of algorithms.

Instrument 4. The work of the genetic algorithm begins with the formation of the first population, which in the general case can be random; for this, an algorithm was created that forms a random path along the SRG and generates a corresponding copy of the source code.

Instrument 5. Since the Concept uses chromosomes of variable length, their initial "successful" choice will qualitatively affect the convergence of the solution to the optimization problem. For this, a corresponding forecasting algorithm was created using statistical data regarding the correspondence of the size of the source code and the size of the machine code sections compiled from it (collected from the source code of functions in the C programming language from the ExeBench dataset).

Instrument 6. One of the rather complex subtasks that needs to be solved using the Instrumentation is the selection of constant values using crossover and mutation operations (the total number of variations of which is over 4 billion). However, there is a certain statistical probability of the occurrence of certain values, collected by analyzing the source code of functions (from the ExeBench dataset) and used by the corresponding prediction algorithm.

Instrument 7. To select instances of the source code, it is necessary to compare the corresponding assembler code, for which purpose a corresponding author's metric was created, the essence of which is as follows. First, the assembler code is represented as a list of text lines. Second, the proximity to the same positions of the characters of each line is estimated. Third, the proximity to the same positions of the lines of each text is estimated (taking into account their differences in characters). Fourth, the proximity is "enhanced" when each character and line is located closer to the beginning of their sequences and lists. And, fifthly, the normalization of the proximity of both individual lines and all texts to the range [0…1] is performed, which allows the metric values to be used in other, more complex, mathematical assessments.

The described Instrumentation is the fourth main scientific result of the current author's research.

VII. Software Prototype and Experiment

To test the Instrumentation, a corresponding software prototype of the GDC was created in Python (the modules of which implemented each of the instrument), allowing for a given syntax to obtain the source code of a program from its machine code using modified genetic algorithms.

In an experiment with the prototype, the GDC of the previously given mathematical expression "z = x*(y – (x/y))" was produced (for the corresponding syntax of the programming language), the assembler code of which for the x86 architecture processor had the following form:

```
mov eax, DWORD PTR _x$[ebp]  ; move the value of variable "x"
                             ; to register EAX
cdq                          ; convert the signed
                             ; double word from EAX
                             ; to a signed quad word in EDX:EAX
idiv DWORD PTR _y$[ebp]      ; signed division of register EAX
                             ; by variable "y"
mov ecx, DWORD PTR _y$[ebp]  ; move the value of variable "y"
                             ; to register ECX
sub ecx, eax                 ; subtraction from register ECX
                             ; to register EAX
imul ecx, DWORD PTR _x$[ebp] ; signed multiplication of ECX
                             ; by variable "x"
mov DWORD PTR _z$[ebp], ecx  ; move the value of register ECX
                             ; to variable "z"
```

The prototype settings in terms of genetic algorithms were as follows: the frequency of crossover and mutation in form was 50%, and in content – 10%. A personal computer ("Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz") was used as a hardware platform. The prototype restored the correct source code in 30 seconds of operation (i.e. 6-10 times faster than experts), while making about 1000 compiler calls (which is the most time-consuming operation). Thus, we can talk not only about the general operability of the prototype, but also about the possibility of its practical application in limited conditions.

VIII. The Genetic De-evolution Architecture

The common scheme of the GRE is shown in Fig. 2

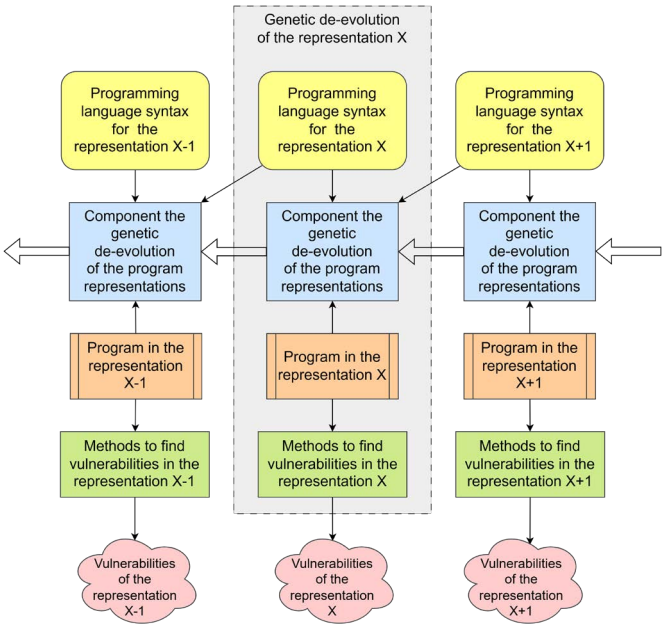

Fig. 2. Scheme of the cascade system of genetic reverse engineering.

Following the scheme in Fig. 2, the GRE process is built from similar components according to the specified "pipe-line" principle. In this case, in addition to the reconstructed program representations, vulnerabilities present (i.e. specified in that basis) in each of the representations are identified. The architecture of each component (hereinafter referred to as the Architecture) of such a cascade GDE corresponds to the generalized scheme of the Instrumentation given earlier and does not require more detail. It is only worth emphasizing that each component allows transmitting the program representation it has reconstructed to the input of the next component, uses a common syntax record (or final SRG) with the neighboring one, is controlled by an expert using a single mechanism (for example, by specifying specialized settings of a modified genetic algorithm), calls the corresponding means of direct program transformation (for example, a source code compiler to obtain its machine analog, the identification of the version and settings of which is generally considered to be a solved problem [14]), allows connecting external vulnerability search tools, etc. This Architecture is the fifth main scientific result of the current author's research.

The Component Architecture entry in analytical form formally justifies the possibility of constructing all GRE schemes from them. For example, a separate component (Component) has the following entry:

$$R_{i-1} = Component(R_i, PC_i), \quad (1)$$

where R_i – i-th representation, PC_i – its parameters; the entire GRE process can be represented as follows:

$$R_1 = Component(R_2, PC_2) \circ \ldots \circ Component(R_n, PC_n), \quad (2)$$

where «°» – the operator of the "pipe-line" connection organization.

Detection of vulnerabilities (V_i) n the i-th representation by some method ($Find_i$) has the following analytical record:

$$V_i = Find_i(R_i, PF_i), \quad (3)$$

where PF_i – parameters of the search method.

Based on GDE, a vulnerability search method was proposed, developed and implemented, based on signature analysis of the chromosome corresponding to the restored program instance. The signature itself was a set of a node on the SRG and a part of the chromosome in the form of a subsequence of alternative selection, the coincidence with which means the presence of a vulnerability in the program. Naturally, the signature record can be expanded by using more complex constructions, such as regular expressions (on graphs), templates, etc. The search itself, following the already applied tradition, was called signature-genetic in the study. This vulnerability search was implemented as a corresponding prototype, and experiments showed its operability. An important difference of the search is its independence from the syntax of representations, since a specific SRG can be selected based on the expert's tasks; in particular, so as to most accurately display specific vulnerabilities.

IX. Novelty, Theoretical and Practical Significance

Each of the results has its own novelty and consists of the following: (1) the Methodology was obtained for the first time, the Model differs from similar ones by a high level of abstraction, a single set of elements and support for multiple software engineering scenarios; (2) the Concept is based on an approach, which is opposite to the classical one, since it

transforms program instances in the current representation to the subsequent one; (3) the elements of the Instrumentation individually are also either new or differ from alternative characteristics that take into account the specifics of GDE; (4) the Architecture, in turn, is both a software projection of the author's Concept and contains a new method of signature-genetic vulnerability search.

The theoretical significance of the Model lies in the generalization and systematization of the steps of classical software reverse engineering, presented in a graphical form and a system of analytical records. The Methodology allows one to build a "path" (as a set of interrelated methods and tools) for conducting reverse engineering for a large class of problems.

The Model introduces the concepts of program representations and establishes interrelated transformations between them; also, a new classification of vulnerabilities by the structural level of their location in the code is distinguished. From the standpoint of practical significance, the Model allows one to build a route for de-evolution of representations with the reflection of vulnerabilities in them.

The concept is intended for the theoretical justification of the possibility of reverse engineering a program by de-evolution of its representations, based on genetic algorithms. The concept is the basis for the practical implementation of the corresponding methods, their software architectures and implementations.

One part of the Instrumentation has theoretical significance – a syntactically optimal model of the source code, the relationship between the sizes of the source and machine code of programs, the probability of the appearance of constant values in the source code, the formula for the proximity of two assembler codes (presented as text) are obtained; anothrer part has practical significance – the ability to create solutions in the GDC part and others, based on working with the source code model, the generation of syntactically correct programs.

X. Conslusion

The paper presented the progress of the authors' research devoted to software reengineering for the purpose of searching for vulnerabilities in IE software systems based on genetic algorithms.

Five main scientific results obtained, prerequisites for their creation and their relationship with subsequent ones are described. From the standpoint of efficiency (see Table I), the proposed approach to reverse engineering has the following indicators: performance – high (since, for example, the resulting code will be guaranteed to compile into the specified machine code), operativeness – medium or high (which can be ensured by optimizing the implementation, parallelization and other techniques), resource safekeeping – high (since expert participation is practically not required, and GRE algorithms do not consume significant resources). From this standpoint, the method has certain advantages over the others (the sum of its points is 8.5), competing only with decompilers. However, its important advantage is the initial ability to support arbitrary syntax of the programming language and program execution processor, which none of the methods have.

All the results obtained are certainly novel, as well as theoretical and practical significance for the field of IE. The continuation of the work should be the testing of the final GRE system in practice.

References

[1] P. Reiter, H. J. Tay, W. Weimer, A. Doupé, R. Wang, and S. Forrest, “Automatically mitigating vulnerabilities in binary programs via partially recompilable decompilation,” IEEE Transactions on Dependable and Secure Computing, doi: 10.1109/TDSC.2024.3482413.

[2] N. Mauthe, U. Kargén, and N. Shahmehri, “A large-scale empirical study of android app decompilation,” in 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 2021, pp. 400–410, doi: 10.1109/SANER50967.2021.00044.

[3] D. S. Katz, J. Ruchti, and E. Schulte, “Using recurrent neural networks for decompilation,” in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso, Italy, 2018, pp. 346–356.

[4] K. Xue, Q. Han, S. Han, Z. Shi, and Y. Qiao, “A review of software testing process log parsing and mining,” in 2024 IEEE International Conference on Software Services Engineering (SSE), Shenzhen, China, 2024, pp. 334–343, doi: 10.1109/SSE62657.2024.00055.

[5] I. Kotenko, K. Izrailov, and M. Buinevich, “Static analysis of information systems for IoT cyber security: A survey of machine learning approaches," Sensors, vol. 22, iss. 4, 2022, art. no. 1335, doi: 10.3390/s22041335.

[6] A. A. Mundade and T. M. Pattewar, “Comparison study of optimized test suite generation using Genetic and Memetic algorithm,” in 2015 International Conference on Pervasive Computing (ICPC), Pune, India, 2015, pp. 1–5, doi: 10.1109/PERVASIVE.2015.7087175.

[7] K. Tanemura, Y. Sasaki, S. Takahashi, Y. Tokuni, H. Manabe, and R. Miyadera, “Application of genetic programming to unsolved mathematical problems II,” in 2023 IEEE 12th Global Conference on Consumer Electronics (GCCE), Nara, Japan, 2023, pp. 608–609.

[8] M. Buinevich, K. Izrailov, and G. Ganov, “Intellectual method of program interactions visualisation in Unix-like systems for information security purposes,” in IEEE 12th Majorov International Conference on Software Engineering and Computer Systems, Saint-Petersburg, Russia, 2020, pp. 59–71.

[9] R. Tamilkodi, V. B. Sankar, S. Madhu, L. Revathi, V. S. Guna Srikar, and E. Ajay, “Exploring IoT device vulnerabilities through malware analysis and reverse engineering," in 2024 5th International Conference on Data Intelligence and Cognitive Informatics (ICDICI), Tirunelveli, India, 2024, pp. 279–283, doi: 10.1109/ICDICI62993.2024.10810929.

[10] H. Sparks and K. Ghosh, “NUVER: Network based vulnerability visualizer,” in 2023 IEEE 30th Annual Software Technology Conference (STC), MD, USA, 2023, pp. 16–19.

[11] I. Kotenko, K. Izrailov, M. Buinevich, I. Saenko, and R. Shorey, “Modeling the development of energy network software, taking into account the detection and elimination of vulnerabilities,” Energies, vol. 16, iss. 13, 2023, art. no. 5111, doi: 10.3390/en16135111.

[12] A. Latif, F. Azam, M. W. Anwar, and A. Zafar, “Comparison of leading language parsers – ANTLR, JavaCC, SableCC, Tree-sitter, Yacc, Bison,” in 2023 13th International Conference on Software Technology and Engineering (ICSTE), Osaka, Japan, 2023, pp. 7–13.

[13] R. Xu, X. Mao, and W. Xiao, “Accelerating high-precision vulnerability detection in C programs with parallel graph summarization,” in 2023 6th International Conference on Software Engineering and Computer Science (CSECS), Chengdu, China, 2023, pp. 1–6.

[14] Z. Pan, Y. Yan, L. Yu, and T. Wang, “Identification of binary file compilation information,” in 2022 IEEE 5th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), Chongqing, China, 2022, pp. 1141–1150.