# GREMC: Genetic Reverse-Engineering of Machine Code to Search Vulnerabilities in Software for Industry 4.0. Predicting the Size of the Decompiling Source Code

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

2024 International Russian Smart Industry Conference (SmartIndustryCon) | 979-8-3503-9504-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/SMARTINDUSTRYCON61328.2024.10515515

***Abstract*—The article is devoted to the problem of security of cyber-physical systems as part of production according to the Industry 4.0 concept. For this purpose, the author's approach of "Genetic Reverse Engineering of Machine Code" (GREMC) is proposed. The essence of this approach lies in the use of artificial intelligence in the field of genetic algorithms to restore the source code of software executed in the form of machine code on cyber-physical devices for Industry 4.0. The resulting code can then be analysed for vulnerabilities by an expert. One issue that arises during genetic reverse engineering is predicting the size of the source code based on its machine representation (i.e., in an object or executable file). The article is devoted to solving this problem for functions in the C programming language. To do this, a method for obtaining a relationship between the sizes of the source and machine code of individual functions is described, which consists of steps such as loading a dataset with C-functions, isolating the function code in it, preprocessing them, compiling them into machine code, calculating the required sizes, building dependencies between each source code size and the corresponding machine code sizes, generating the final table and determining the dependency formula. An experiment is carried out using a software prototype that implements the method; ExeBench with 83 thousand C-functions is taken as a dataset. Justifications are given regarding the appearance of abnormal machine code sizes and their impact on the dependence formula.**

***Keywords*—machine code, source code, reverse-engineering, decompilation, size dependence**

## I. Introduction

The security of the functioning of cyber-physical systems is one of the main problems of the modern world. This is due to the close connection between physical mechanisms and their software control. This connection is realised by software; however, both accidental and malicious vulnerabilities within it can lead not only to the implementation of information threats (violations of the triad – confidentiality, integrity and availability), but also to incidents in the real (physical) world - material damage and human casualties. The problem of secure software acquires particular relevance in production built according to the concept of Industry 4.0 since it is based on the use of cyber-physical systems.

Software (hereinafter – SW) used in production usually consists of programs in binary form, which, in turn, consist of machine instructions for direct execution on the CPU – i.e. machine code (hereinafter – MC). To detect vulnerabilities (with subsequent neutralisation) in the MC, there exists a particular set of methods consisting of both automatic software tools and manual techniques.

However, automatic tools usually allow you to detect only well-formalized and relatively "simple" vulnerabilities, whereas manual ones require enormous amounts of time and human resources. As a result, the process of searching for vulnerabilities in the MC can be assessed as having below-average efficiency.

The situation is aggravated by the fact that a targeted cyber attack through SW on production complexes is characterised by introducing hard-to-detect vulnerabilities into program code (for example, obfuscation of control logic [1]) and the deliberate exploitation of existing ones. Moreover, such vulnerabilities, in terms of the degree of "injection" into the program code, are often not low-level (for example, buffer overflow errors) but medium-level (for example, incorrect password verification), high-level (for instance, unauthorised interaction between architecture modules via a backdoor) or even conceptual (for example, the absence of encryption algorithms or weak algorithms being in place).

One approach to searching for vulnerabilities (especially those with a medium and high penetration level) is *decompilation* [2, 3] – i.e. the process of obtaining the corresponding pseudo-source code (and subsequently, software algorithms and architecture) from the MC. However, existing decompilation tools (the most famous of which are the Hex-Rays plugin for IDA Pro [4] and Ghidra [5]) use built-in



979-8-3503-9504-4/24/$31.00 ©2024 IEEE

complex algorithms for converting MC constructs and their combinations into similar elements in the source code (hereinafter – SC). The author's extensive experience in this area of reverse engineering has shown that such tools often restore uncompilable or incomplete code, produce code that is not always readable, or lead to software exceptions during operation. An alternative approach developed by the author is "Genetic Reverse Engineering of Machine Code" (hereinafter – GREMC), based on the use of artificial intelligence in the field of genetic algorithm usage for decompilation (the approach will be described in more detail below). The main goal of GREMC is to iteratively solve the optimisation problem of selecting SC variants closest to the MC after compilation, which must then be decompiled; the choice of CPU architecture and compiler is considered a partially solved problem [6]. The end product of GREMC is not pseudo-source code that only reflects the logic of SW but code in an actual programming language that compiles exactly into the MC under investigation. Such an SC can either be analysed by an expert for inherent vulnerabilities or modified and reassembled into an MC where this vulnerability is absent. Thus, the use of GREMC for SW will significantly improve the overall security of industrial solutions based on the Industry 4.0 concept.

The main goal of GREMC is to determine a specific variant of SC using "smart" enumeration of the sequence of elements that compose it, which is compiled into an MC identical to the one being decompiled. Consequently, one of the tasks in this approach is determining the size of the SC. Otherwise, it would be necessary to select not only the elements of the sequence but also its size, which would qualitatively complicate the implementation of the approach and theoretically make the problem unsolvable (since the size of the source code can increase almost indefinitely). Naturally, it is advisable to compose SC not from individual symbols but from constructs of its programming language (for the C/C++ language, these will be named function headers [7], auxiliary or meaningful named variables [8], block boundary elements "{" and "}", "+" or "–" operations, "if" or "else" operators, etc.).

Thus, the task of the current study can be formulated as follows:

*"Creation of a method and software tool for predicting the size of a source code from its corresponding machine code."*

The details of the formulation and solution of the problem will be described later in the article, and its relevance is also justified by the fact that there are no methods or means close to its solution.

Section 2 provides a brief overview of works related to the problem of the current study. Section 3 describes the author's GREMC approach. Section 4 describes the method and its software implementation for solving the research problem. Section 5 conducts an experiment based on the method and analyses the results. Section 6 discusses the shortcomings of the solutions adopted in the investigation and provides ways to eliminate them. Section 7 represents the article's final part, summarising the research and indicating its novelty, theoretical and practical significance.

## II. Related Works

Let us review works related to the study's central problem – determining the relationship between SC and MC sizes.

The work [9] provides a method for converting MC into algorithms without the need to reconstruct SC. The application of the method indicates the detection of malicious code, particularly those with deliberately "obfuscated" SC. The method uses control flow graphs to represent the program as algorithms.

In [10], the authors present the BinDeep tool, designed to compare SC fragments with functionally similar fragments in MC. The solution's practical significance lies in searching for code clones, particularly those containing malicious code.

In [11], a recurrent neural network is used to decompile MC fragments. A noted feature of the proposed approach is its independence from the programming language. The neural network modules are trained on SC templates. The solution can be used for manual analysis of SW when its SC is missing.

The work [12] is also devoted to decompilation, but in terms of restoring functions built into MC, optimised during compilation from SC. The author's method is based on supervised machine learning and can be combined with the Ghidra product. The application of the method is the analysis of SW corresponding to MC to detect malicious code and theft of intellectual property.

The article [13] is devoted to predicting the size of SC, however, not according to MC, but according to a specific software engineering process. To do this, a 6-step method is proposed that considers the following features of SW: unadjusted weights of actors and use cases, unadjusted points of use case, technical complexity and environmental factors, adjusted points of use case, and man-hours. Thus, it is possible to predict the SC size of large projects formally.

A brief review of the works showed an almost complete absence of solutions (by solution, here we mean methods and their software implementations) applicable to the current research problem.

## III. Genetic Reverse-Engineering

The classic approach to decompilation can be called reverse engineering since, according to the SW life cycle [14], it carries out a transformation from the current representation (i.e. MC or similar assembly code) to the previous one (i.e. SC). From this point of view, the GREMC approach proposed by the author can be called "pseudo-direct" since it seeks to select a SC representation that, when compiled, would be transformed into the MC under investigation. Thus, GREMC solves an optimisation problem [15] in which the parameter is a sequence of SC constructs (symbols, tokens, abstract syntax nodes, templates or other entities), which, when compiled, would produce an MC that is closest to the required one (ideally, to the original). Obtaining identical MCs means solving the optimisation problem and detecting SCs – i.e. reaching a global extreme. Theoretically, this problem could be solved by a complete search of all SC constructions, but this process will consume an unacceptable amount of resources. Let

us give a very rough example of calculations for this process. Let's say that a particular SC consists of 1000 characters (i.e. a relatively small isolated function), each of which can take 50 values (as combinations of letters of the English alphabet, numbers, signs and spaces). The total number of SC options will be $50^{1000}$. And taking into account the fact that the approximate compilation time of one SC can take 1 second, the time of a complete search will take almost infinity.

To solve this problem, it is proposed to apply genetic algorithms [16] in the following step-by-step manner. The essence of genetic algorithms lies in creating a population of individuals, the features of which (structure, parameters, properties, etc.) are specified by a chromosome consisting of a set of genes.

In the first stage of the genetic algorithm, an initial population of individuals is created, the genes of which can be randomly assigned. Thus, after this stage, many random individuals will be generated.

In the second stage, the selection of individuals most adapted to the environment occurs using the so-called Fitness function. This function numerically determines the "survivability" of each individual, allowing you to select the most successful. Thus, after this stage, the population consists of its "best" representatives (from the Fitness function perspective). It is evident that the fitness of individuals is determined precisely by their genes. If any individual has a Fitness function with a given value (for example, the maximum possible), the problem is considered solved, and the algorithm ends.

In the third stage, individuals are crossed, which consists of mixing their genes and obtaining other individuals (to replenish the population reduced in the second stage). Thus, after this stage, new individuals possess genes from the best representatives of the population.

At the fourth stage, individual genes are mutated, introducing some "noise" into the individual's chromosomes and, consequently, their adaptability. This stage is necessary to exit local extrema when solving an optimisation problem [17].

Then, execution is repeated from the second stage.

Let us present the correspondence between the terms of genetic algorithms and the associated GREMC concepts:

1) Individual – some instance of SC, subject to compilation into MC;

2) Population of individuals – a set of SC instances obtained in the process of work;

3) Chromosome – sequence of SC constructs;

4) Gene – a separate SC construct that makes up the entire chromosome (at the moment, from the author's point of view, the most suitable construct is a C programming language token);

5) Selection – selection of SC instances, compiled into MC, closest to the one under investigation (i.e., subject to reverse engineering);

6) Crossing – mixing the structures of two SC instances to obtain a new SC instance;

7) Mutation – a random change in the design of an SC instance (i.e., replacing one token with another);

8) Fitness function – a function that calculates the proximity of two MCs (the first was obtained by compilation from an SC instance, and the second is the one under investigation).

In these terms, the GREMC operating algorithm is as follows. First, many random SC instances are created. Secondly, all SC instances are compiled into some MC. Thirdly, using the Fitness function, the proximity of their MC and the one under investigation is assessed, and those closest to the desired SC are selected. If an SC that compiles exactly into the desired MC is obtained, then the problem is considered solved. Fifth, new SC instances are created from old tokens. Sixth, random tokens in SC are changed to random ones. Seventh, the process is repeated from the moment the SC set is compiled.

Also, tokens mean individual programming language elements, such as keywords, variables, etc. For example, the function of summing numbers "int sum (int x, int y) { return x + y; }" using tokens can be written as the following list: [ "int", "sum", "(", "int", "x", "," , "int", "y", ")" , "{" , "return", "x", "+" , "y", ";" , "}" ].

Based on the GREMC idea, the essential issue remains the choice of chromosome length – i.e. SC instance token lengths. Despite the existence of genetic algorithms with chromosomes of variable length [18], one solution to this issue could be to predict the size of the SC based on the size of the MC instance. It is precisely this dependence that this investigation is devoted to, the progress and results of which will be directly presented below.

## IV. Method and Prototype

The following method (hereinafter – Method) was developed to obtain the dependence of the SC size in tokens on the MC size in bytes. Its essence is obtaining a large number of SC functions in the C programming language, compiling them into MC, calculating sizes, collecting statistics and determining the final dependency. The C programming language was chosen for its popularity in developing SW cyber-physical devices, particularly production and technological devices for Industry 4.0 (due to high execution performance, for example). Also, Microsoft Visual Studio Community 2019 (hereinafter – MSVSC2019) included in the product was taken as a compiler. The initial dataset was data from the ExeBench project, containing a set of executable functions in the C language for use in machine learning [19]. The author is sincerely grateful to the team of this project for the enormous work done and high-quality results!

Let us further describe the steps of the proposed Method.

*Step 1. Loading dataset with C-functions*

Structures are loaded in JSON format containing functions in the C programming language from the ExeBench project.

*Step 2. Extracting SC C-functions*

From the loaded JSON structures, SC C-functions are extracted and assigned unique names (for unambiguous identification), which are added to a single internal storage.

*Step 3. Preprocessing SC C-functions*

SC C-functions are preprocessed as follows:

- the keywords “inline” and “__inline__” are removed before the function signature since otherwise the compiler will not generate an MC for the function (based on the assignment of the keywords);
- the “static” keyword is removed before the function signature since otherwise, an MC will not be generated for the compiler function (based on the purpose of the keyword);
- “NULL” is replaced with “((void *)0)” since this macro is not built into the compiler syntax;
- SC fragments “__attribute__((…))” are removed since they are not standard for the C language and are not supported by many compilers;
- functions with SC containing assembly inserts (defined by the keyword “__asm__”) are removed since it is only necessary to find the dependency between SC and MC;
- optionally, functions with SC containing operations with floating point types (“double” and “float”) are removed since they can lead to an abnormally large MC size (which will be discussed further);
- optionally, functions with SC containing initialisation of complex variables (arrays and strings) in the function body are removed since they can lead to an abnormally large MC size (which will be discussed further).

*Step 4. Compiling SC C-Functions*

The SC of each function is copied into a C file (“file.c”), which is compiled (with the “cl.exe” utility) without optimisation (the “/Od” switch) to produce only object files (the “/c” switch) containing the MC (key “/Fo”); For debugging purposes, an assembler file is also generated (key “/Fa”). The resulting line is:

```
> cl.exe file.c /c /Od /Fafile.asm /Fofile.obj
```

”.

*Step 5. Calculating SC and MC sizes*

The code size is calculated both in the source (text) and the machine (binary) representation. The first size is calculated by dividing the SC into tokens and counting their number. The object file with MC obtained during compilation is used to calculate the second size. Since the object file contains other information (determined by its header) besides the CPU instructions themselves, it needs to be parsed, the section with the code must be selected, and its size needs to be obtained.

All received sizes are entered into the internal storage. In case of an error, its text is also saved, and the function is marked as uncompilable. Then, for each SC size, the corresponding MC size's minimum, maximum and average value is calculated; additionally, the number of elements in these lists is stored in the storage.

*Step 6. Outputting the dependency table*

The dependencies between the sizes of the SC and the MC compiled from it are output in tabular form for subsequent visualisation. In the Method, this table is intended to be loaded into Microsoft Excel for semi-automatic analysis.

*Step 7. Defining the dependency formula*

The formula for the relationship between the SC and MC sizes is determined. To do this, the Method uses tools built into Microsoft Excel to construct trends according to the predefined law (in the scatter charts settings).

The method was automated using a developed prototype software tool (hereinafter referred to as the Prototype), which automatically performs all steps except the 7th. The prototype was realised using the Python 3.10 programming language. It used the following libraries: json – for working with files in JSON format (i.e. containing C-functions), subprocess – for launching external processes (i.e. the “cl. Exe”); nltk – to divide the SC function text into a list of tokens; coff – for parsing the resulting object files of The Common Object File Format (abbr. COFF) and extracting sections with MC from them; signal – to intercept the “Ctrl+C” press to shut down the work correctly. Also, the Prototype automatically scans a given directory for the presence of JSON files with C-functions, allows you to load and update the internal storage with functions from new JSON files, makes intermediate saves of the internal storage to an external file, controls the omission of functions with floating point types and initialisation of complex variables, displays a log of its work on the console.

## V. Experiment

Let us further describe the experiment conducted using the Method and the Prototype.

The following initial data and parameters were taken in the experiment:

- the dataset with C-functions was downloaded from the Internet address https://huggingface.co/datasets/jordiae/exebench/tree/main;
- C-functions with floating-point types and initialisation of complex variables were skipped;
- to calculate the dependence, we took an SC of no more than 300 tokens in size since, for a larger size, instances of C-functions were either absent or in single quantities;
- the MC size was considered as the average value of the set of all MCs (for a particular SC size);
- to determine the formula for the dependence between the sizes of SC and MC, a power trend was used (according to Microsoft Excel terminology).

An experiment with the Method in the form of a log of the work of the Prototype is presented below; the sign "..." marks the omission of lines similar to the previous ones.

```
(2024.01.24 20:07:56) Loading dataset from
'Dataset\real_test\data_0_time1678114487_default.jsonl' -
> OK (2132 items, {'WithRealError': 2})

(2024.01.24 20:07:57) Loading dataset from
'Dataset\valid_real\data_0_time1677999491_default.jsonl'
-> OK (2131 items, {'WithRealError': 2})

...

(2024.01.24 20:09:59) Loading dataset from
'Dataset\train_synth_simple_io\data_0_time1677909063_defa
ult.jsonl' -> OK (4818 items, {'WithRealError': 5182})

(2024.01.24 20:10:03) Loading dataset from
'Dataset\train_synth_simple_io\data_0_time1677914260_defa
ult.jsonl' -> OK (4786 items, {'WithRealError': 5214})

(2024.01.24 20:10:07) Dataset preparing (219077) -> OK
({'Pass with asm': 478, 'Skip double/float': 13569, 'Skip
array init': 2438, 'Skip by position': 0})

(2024.01.24 20:10:15) Dataset compiling (202592 items)
...

(2024.01.24 20:10:15)   0) Compile function
'[data_0_time1678114487_default:0] num2str()' ->
Succeeded

(2024.01.24 20:10:15)   1) Compile function
'[data_0_time1678114487_default:2] R90deg()' ->
Succeeded

...

(2024.01.24 22:39:55)   202590) Compile function
'[data_0_time1677914260_default:9993] icosd()' ->
Succeeded

(2024.01.24 22:39:55)   202591) Compile function
'[data_0_time1677914260_default:9996]
get_current_frame()' -> Failed

(2024.01.24 22:39:55)   Saving to 'a_dataset_3.json' ->
OK

(2024.01.24 22:39:57) -> OK ({'All': 202592, 'Skipped':
0, 'Compiled': 202592, 'Succeeded': 83450, 'Failed':
119142})
```

According to the log, the Prototype process took 2 hours, 32 minutes and 1 second. At the same time, 219077 C-functions were loaded, of which 202592 copies were prepared for compilation. Of all the C-functions, 83,450 instances were successfully compiled, and 119,142 resulted in various errors. Thus, to build the dependence, approximately 83.5 thousand ratios of the number of tokens in SC and the corresponding MC sizes were obtained.

As a result of applying the Prototype and selecting SC with a size of no more than 256 tokens, 82282 instances were used to track the dependence, which is $\frac{82282}{83450} = 98.6\%$ of their total number and is a reasonably representative sample for evaluation. The distribution of the number of C-function instances versus their size is presented in Fig. 1, which also shows the distribution of Zipf (with a selected value for the first SC value of 1 token) – an empirical pattern of the frequency distribution of natural language words.

Following from the fact that (see Fig. 1) the distribution of the number of SCs of a certain length is close to Zipf's law, we can assume the correctness and "naturalness" of the taken sample of C-functions; the insufficient size of the original dataset can explain present discrepancies.

When forming a direct dependence of the SC on the MC size, it was taken into account that object files (with the extension "*.obj") had the COFF format after compiling C-functions in MSVSC2019, and CPU instructions were contained in sections with the service name ".text$mn". Thus, the calculated MC size was significantly different from the size of the object file itself.

The resulting graph of MC size versus SC (with linear trend) is presented in Fig. 2.

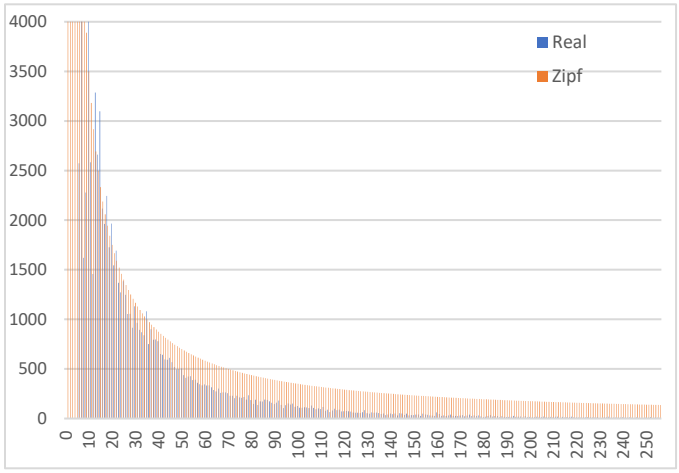


Fig. 1. Distribution of the number of C-function instances depending on their size (in tokens).

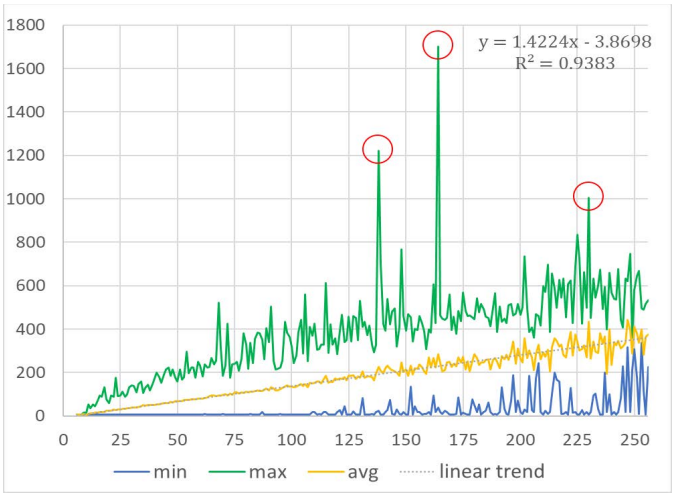


Fig. 2. Dependence of MC size (in bytes) on SC size (in tokens).

Thus, the formula for the dependence of the SC size (SC_Size) on the MC size (MC_Size) has the following average form (see the upper right part of Fig. 2):

$$SC_{Size} = 1.4224 \times MC_{Size} - 3.8698,$$

in this case, the reliability of the approximation is relatively high – $R^2 = 0.9383$.

A chart of the ratio of the MC size to the SC size (with a power-law trend) is presented in Fig. 3.

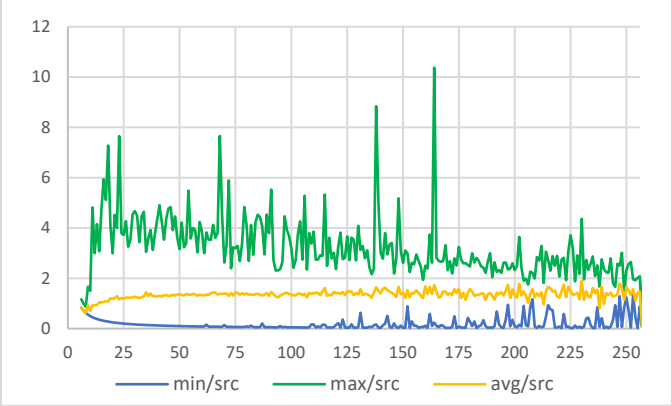


Fig. 3. Ratio of MC size (in bytes) to SC size (in tokens).

The graph in Fig. 3 repeats the relationships in Fig. 2 but presents them in a different form.

Analysis of the dependence graphs (see Fig. 2 and 3) allowed us to identify several anomalous size ratios (in the form of peaks marked with red circles) and to analyse the SC that caused them:

1) $SC_{Size} = 138, MC_{Size} = 1220$ – many complex nested function calls with checking return values;
2) $SC_{Size} = 164$ , $MC_{Size} = 1701$ – many complex function calls with floating-point arguments;
3) $SC_{Size} = 230, MC_{Size} = 1004$ – many function calls with arguments in the form of results of arithmetic and logical operations.

Thus, all anomalous relationships arise due to the compilation of SC, which is not typical (i.e., averaged) since it contains floating point values (generating a specific MC) or SC in a "compressed" coding style (when programming language constructs are written in one line without using temporary variables). Such C-functions are specific, poorly reflect the general SC coding style and may not be considered.

## VI. Discussion

A specific dataset (i.e. ExeBench) was selected as the SC source for C-functions for the following reasons. Firstly, there aren't a lot of datasets containing individual functions that can be compiled without a software environment (for example, include files). For example, if you take any large project with many functions, then although it will be compiled as a whole, individual functions will "pull" others along, including library ones. Secondly, the ExeBench dataset already has a fairly good structure since it consists of JSON files with SC functions, assembly code for several compilers (except for the one used as part of MSVSC2019), compilation errors and additional meta-information. And thirdly, the C-functions of the dataset do not relate to any specific area of SW but are averaged for software engineering.

Initially, the ExeBench dataset contained assembly code obtained by GCC compilers with various optimisations for a number of CPUs (for example, x86 and ARM), from which the corresponding object files could be generated. However, the compiler included in MSVSC2019 is a full-fledged tool for obtaining object code [20], including the one operating in devices for Industry 4.0. At the same time, the theoretical relationship between the size of SC in one programming language and MC for a specific CPU will not highly depend on the choice of a specific compilation tool.

As indicated, the SC size can be understood as various metrics – the number of text characters (for example, characters in the string "int x = 0;"), lexical objects (for example, a list of tokens – TOK_INT, TOK_IDENT("x"), TOK_ASSIGN, TOK_CONST("0"), TOK_SEMICOLON), syntactic constructions (for example, a subgraph in Abstract Syntax Tree [21] – AS_DECLARATION (AS_ASSIGN(AS_IDENT("x"), AS_CONST("0"))), etc. However, counting the number of SC tokens compared to the number of characters will increase the performance of the genetic algorithm since generating SCs from lexically correct pieces of text (rather than a random sequence of text) will be more likely to result in a compliable result. Using more abstract entities (subgraphs of syntactic constructs corresponding to formal syntax language) is possible but more complex; this will be explored by the author in the future.

In the experiment, when plotting the relationship between SC and MC sizes, some abnormally high MC sizes were discovered in object files, which was expected because of the features of SC coding and the specifics of the generated MC. Based on the fact that the number of such anomalies (only 3) is insignificant compared to the total number of examined specimens (82282), and the value of the anomalous size is one, which exceeds the average by no more than ten times (see Fig. 2), then they do not have a significant effect on the dependence formula. For the same reason, the calculation skipped SC, which used the double and float types, and also included dynamic initialisation of arrays in the body of the functions.

The resulting formula for the dependence of the SC size on the MC size has a relatively simple linear form

$$SC_{Size} = A \times MC_{Size} + B,$$

where A and B are coefficients. However, this is understandable and realistic since an increase in structures in SC logically leads to approximately the same increase in instructions in MC.

## VII. Conclusion

The paper presents the author's alternative approach to decompiling MC with obtaining SC, analysing which for vulnerabilities can significantly improve production security for Industry 4.0 [22]. The essence of the approach (abbreviated GREMC) is using artificial intelligence in terms of genetic algorithms to iteratively approximate the SC to a representation

that would be compiled into the desired MC. One of the objectives of GREMC is to determine the size of the original SC, which is the focus of this investigation.

The main result of the current work is a Method and Prototype that allows the relationship between the SC and MC sizes of individual functions to be determined using statistical data (the ExeBench dataset is used for this). Also, a direct formula for the dependence of sizes was obtained: $SC_{Size} = 1.4224 \times MC_{Size} - 3.8698$.

The theoretical significance of the investigation lies in establishing a direct relationship between the size of SC and MC for individual standard functions. The practical significance lies in the possibility of selecting the parameters of various algorithms (including genetic ones within the GREMC framework), which need to predict the sizes of SW representations during the inverse transformation between SC and MC.

The continuation of the work should be obtaining dependencies between the sizes of SC and MC C-functions for various operating modes of compilers and CPU instructions, refining the dependency formula based on the characteristics of MC, as well as selecting more complex entities for constructing SC.

## REFERENCES


[1] N. Zubair, A. Ayub, H. Yoo, and I. Ahmed, "Control logic obfuscation attack in industrial control systems," IEEE International Conference on Cyber Security and Resilience (CSR), Rhodes, Greece, pp. 227–232, August 2022, doi: 10.1109/CSR54599.2022.9850326.

[2] D. S. Katz, J. Ruchti, and E. Schulte, "Using recurrent neural networks for decompilation," IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso, Italy, pp. 346–356, April 2018, doi: 10.1109/SANER.2018.8330222.

[3] A. Fokin, K. Troshina, and A. Chernov, "Reconstruction of class hierarchies for decompilation of C++ programs," 14th European Conference on Software Maintenance and Reengineering, Madrid, Spain, pp. 240–243, February 2011, doi: 10.1109/CSMR.2010.43.

[4] M. Buinevich, K. Izrailov, and A. Vladyko, "Testing of utilities for finding vulnerabilities in the machine code of telecommunication devices," 19th International Conference on Advanced Communication Technology (ICACT), PyeongChang, Korea (South), pp. 408–414, March 2017, doi: 10.23919/ICACT.2017.7890122.

[5] S. Poudyal and D. Dasgupta, "AI-powered ransomware detection framework," IEEE Symposium Series on Computational Intelligence (SSCI), Canberra, ACT, Australia, pp. 1154–1161, January 2021, doi: 10.1109/SSCI47803.2020.9308387.

[6] I. Kotenko, K. Izrailov, and M. Buinevich, "The method and software tool for identification of the machine code architecture in cyberphysical devices," Journal of Sensor and Actuator Networks, vol. 12., iss. 1., pp. 11, January 2023, doi: 10.3390/jsan12010011.

[7] B. Xia, Y. Ge, R. Yang, J. Yin, J. Pang, and C. Tang, "BContext2Name: Naming functions in stripped binaries with multi-label learning and neural networks," IEEE 10th International Conference on Cyber Security and Cloud Computing (CSCloud)/IEEE 9th International Conference on Edge Computing and Scalable Cloud (EdgeCom), Xiangtan, Hunan, China, pp. 167–172, August 2023, doi: 10.1109/CSCloud-EdgeCom58631.2023.00037.

[8] A. Jaffe, J. Lacomis, E. J. Schwartz, C. Le Goues, and B. Vasilescu, "Meaningful variable names for decompiled code: a machine translation approach," 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC), Gothenburg, Sweden, pp. 20–2010, January 2020.

[9] M. Shudrak and V. Zolotarev, "The new technique of decompilation and its application in information security," Sixth UKSim/AMSS European Symposium on Computer Modeling and Simulation, Malta, Malta, pp. 115–120, January 2013, doi: 10.1109/EMS.2012.20.

[10] S. Alrabaee, K. -K. R. Choo, M. Qbea'h, and M. Khasawneh, "BinDeep: Binary to source code matching using deep learning," IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Shenyang, China, pp. 1100–1107, March 2022, doi: 10.1109/TrustCom53373.2021.00150.

[11] D. S. Katz, J. Ruchti, and E. Schulte, "Using recurrent neural networks for decompilation," IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso, Italy, pp. 346–356, April 2018, doi: 10.1109/SANER.2018.8330222.

[12] T. Ahmed, P. Devanbu, and A. A. Sawant, "Learning to find usages of library functions in optimized binaries," IEEE Transactions on Software Engineering, vol. 48, no. 10, pp. 3862–3876, August 2021, doi: 10.1109/TSE.2021.3106572.

[13] M. Badri, L. Badri, W. Flageol, and F. Toure, "Source code size prediction using use case metrics: an empirical comparison with use case points," Innovations in Systems and Software Engineering, vol. 13, pp. 143–159, September 2016, doi: 10.1007/s11334-016-0285-7.

[14] 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, pp. 5111, July 2023, doi: 10.3390/en16135111.

[15] H. J. Kaleybar, M. Davoodi, M. Brenna, and D. Zaninelli, "Applications of genetic algorithm and its variants in rail vehicle systems: a bibliometric analysis and comprehensive review," IEEE Access, vol. 11, pp. 68972–68993, July 2023, doi: 10.1109/ACCESS.2023.3292790.

[16] C. -Y. Yu and C. -Y. Huang, "Utilizing multi-objective evolutionary algorithms to optimize open source software release management," IEEE Access, vol. 11, pp. 112248–112262, October 2023, doi: 10.1109/ACCESS.2023.3323615.

[17] L. Jiacheng and L. Lei, "A hybrid genetic algorithm based on information entropy and game theory," IEEE Access, vol. 8, pp. 36602–-36611, February 2020, doi: 10.1109/ACCESS.2020.2971060.

[18] Z. Bin, G. Zhichun, and H. Qiangqiang, "A genetic clustering method based on variable length string," 2nd International Conference on Safety Produce Informatization (IICSPI), Chongqing, China, pp. 460–464, 19 May 2020, doi: 10.1109/IICSPI48186.2019.9095977.

[19] J. Armengol-Estapé, J. Woodruff, A. Brauckmann, J. W. de S. Magalhães, and M. F. P. O'Boyle, "ExeBench: An ML-scale dataset of executable C functions," 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS 2022). Association for Computing Machinery, New York, NY, USA, pp. 50–59, June 2022, doi: 10.1145/3520312.353486.

[20] M. Pashinska-Gadzheva, "Comparison of compiler efficiency with SSE and AVX instructions," International Conference Automatics and Informatics (ICAI), Varna, Bulgaria, pp. 56–59, November 2022, doi: 10.1109/ICAI55857.2022.9960080.

[21] G. Si, Y. Zhang, M. Li, and S. Jing, "Malicious code utilization chain detection scheme based on Abstract Syntax Tree," IEEE 6th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), Beijing, China, pp. 1108–1111, November 2022, doi: 10.1109/IAEAC54830.2022.9929773.

[22] C. H. Li, and H. K. Lau, "A critical review of product safety in industry 4.0 applications," IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), Singapore, pp. 1661–1665, February 2018, doi: 10.1109/IEEM.2017.8290175.