International Journal of Scientific Trends- (IJST)
ISSN: 2980-4299
Volume 4, Issue 4, April - 2025
Website: https://scientifictrends.org/index.php/ijst
Open Access, Peer Reviewed, Scientific Journal

Increasing The Efficiency of Damaged File Detection Tools Based on The Use of Hidden Markov Models

1 E. J. Qilichev
1Tashkent State Agrarian University, Tashkent, Uzbekistan

2 I. E. Isroilov
1Tashkent State Agrarian University, Tashkent, Uzbekistan

Abstract

This paper reviews a new algorithm developed to improve the performance of file detection based on hidden Markov models (HMM). Hidden Markov models are an effective method for detecting various types of disturbances and damages using time series and probabilities. The algorithm uses learning-based technologies to quickly and accurately detect file corruption.

The article is based on recovery of damaged files, detection and analysis of changes in their structure. Monitoring file corruption processes through hidden Markov models increases the possibility of correctly predicting errors. The algorithm's performance is more efficient in more complex structures compared to simple statistics, it quickly detects file integration violations and creates a recovery mechanism.

The main part of the article provides a detailed explanation of the mechanisms for explaining and predicting possible file corruption or invalidation using HMM. The new algorithm is designed to improve security and speed up the recovery of damaged files.

Keywords: Heuristic model, statistical model, meta-, polymorphism, artificial neural networks, infection, Snort system, heuristic analysis, imitation, behavioral analysis, signature search, Anomaly detection.

Introduction

One of the main reasons for the increase in the number and variety of attacks on information systems and the spread of malicious software, including programs that gain unauthorized access to protected data, is the existing methods of protection against the introduction of malicious programs. Programs included in information systems often cannot provide the necessary protection. There are two main aspects to this problem. First of all, the most common methods of protection against the spread of malware are based on the recognition of previously known signatures, and therefore cannot fundamentally resist new malware in the initial period of its

existence, as well as malware that has been modified using meta- or polymorphism methods. The same applies to somewhat sophisticated methods of software analysis based on the search for correspondence between the studied information objects and specialized models: malicious programs without images can be successfully used to carry out attacks on information systems. It is suitable for heuristic and statistical models used in defense tools, as well as models based on artificial neural networks.

II METHODS AND MATERIALS

Currently, there are few special tools for detecting infected data: their detection is performed by antivirus programs, firewalls and intrusion detection systems as part of general protection against viruses. These tools are divided into infection prevention tools and secure code development tools. As a means of preventing damage, the most popular anti-malware software tools are analyzed: Snort intrusion detection system and Kaspersky Internet Security. Malicious attachment detection methods used in damage prevention tools can be divided into two groups: static and dynamic analysis methods [1]. The first group of methods is based on the analysis that does not require the operation of the studied object. The second group is about monitoring the program during the work process. A more complete classification of methods for detecting harmful additives is presented in Figure 1.

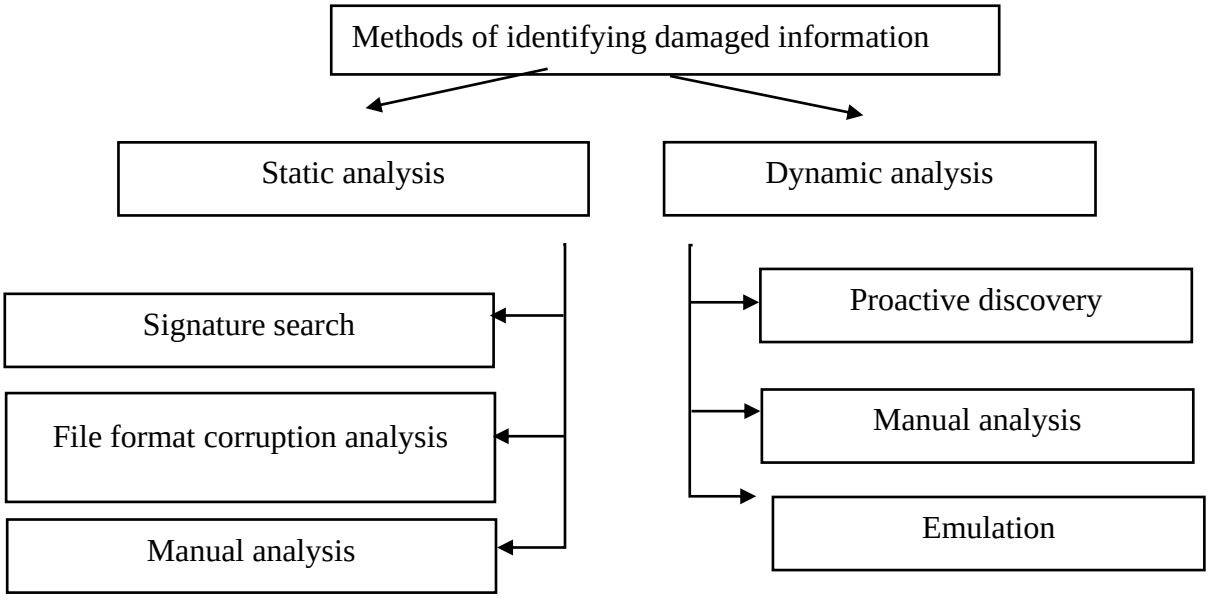

Figure 1. Methods for detecting harmful additives.

In terms of information security, exploits occur due to weaknesses in application and system software. The presence of this type of vulnerability allows an attacker to exploit the vulnerability. Vulnerabilities that could lead to arbitrary code execution include:

1. Buffer overflow vulnerabilities:
   1.1. Stack Overflow.
   1.2. The ball overflows.

2. "format line" vulnerabilities.

3. Integer weaknesses.

Figure 2 provides a diagram illustrating the lifecycle of a software vulnerability and an exploit that exploits it [2].

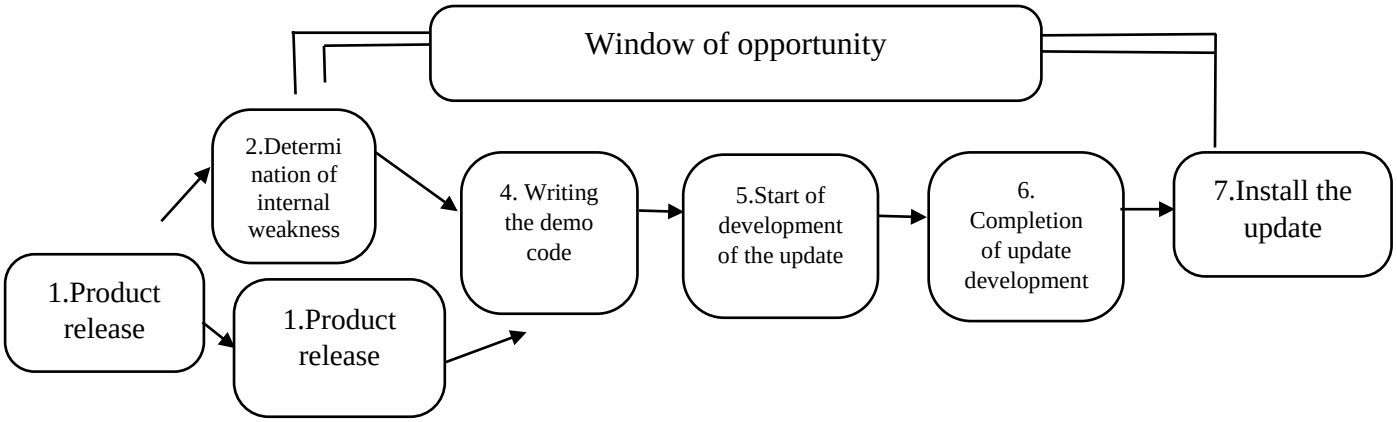

Figure 2. The life cycle of vulnerability.

Between stages 2 and 7 of the vulnerability lifecycles, it is important for an attacker to successfully develop an exploit that they use to launch a discovered vulnerability, creating user-agent threats to the IP. This time period is called the "window of opportunity." The characteristic length of the "window of opportunity" is different for different software companies and can be small or significant depending on how the company organizes the software support process. For example, vulnerabilities in Microsoft software have not been updated for anywhere from a week to six months. Those who are most inclined to use the Internet, therefore, users are isolated from the source of updates.

Based on the analysis of the most popular and effective usage detection systems, the classification of the methods of detecting the damaged data is explained in detail (Jiaru Song, Guihe Qin, Yanhua Liang, Jie Yan, Minghui Sun,, 2024).

Currently, there are the following methods for detecting a corrupted file in real time:

- heuristic analysis;
- emulation;
- behavioral analysis;
- signature search;
- anomaly detection.

Despite the development of emulation methods and heuristic methods, modern antivirus systems and complexes for detection of exploits, as can be seen from the statistics of actual detections, mainly use the last two technologies [4]. However, these technologies do not fully cover the full range of malware in the exploit category and cannot detect new exploits that use a new file format unknown to antivirus software and new malicious code. The widespread use of "obfuscation" - obfuscation of the code - in the development of different versions of exploits that use a certain vulnerability, contributes to the decrease in the effectiveness of these technologies. Taking into account the above and from the point of view of the goal of this dissertation (increasing the effectiveness of tools and methods for detecting damaged files), it is interesting to develop an effective specialized method for detecting malicious attachments in files. Unfeasible, based on

principles other than the above methods. For this purpose, it was decided to develop a method for identifying malicious attachments of inexecutable format files based on a model that takes into account the structural features of exploits and does not use the concepts of exploit signatures and anomalies in the structure.

Dedicated to the development of a model of infected files in non-executable format that identifies the structure and properties of exploits. A mathematical model of the exploit loader is presented, the presence of which allows correcting the fact of exploit injection. Based on the Markov chain of executable instructions included in it, a method of building a language model of the operational loader has been developed. As part of the study of the problem of putting a sequence of instructions in the body of a file in a non-executable format, which appears when searching for a loader, a theorem is formulated and establishes the relationship between the probability of convergence of the two. That is: based on the research of the developed language model of the sequence of iso-assembler in a certain number of steps with the probability of different values of the difference between the initial positions of the disassembly, it was concluded that natural language text classification methods can be effectively used in the task of separating code and data, which allows to determine the exploit loader from the sequence of common data in the file. A method for detecting exploits based on the detection of the presence of an executable loader in the body of a file in a non-executable format has been developed [5].

Usually, an infected file is presented as a corrupted file and an embedded malicious part - shellcode. Malware requires a properly executed bootloader to work. Thus, there is a descriptive model of malicious attachments in files of non-executable format, which fully represents the object studied in the work:

1. There is a container file in some format.
2. Changes to the container file format will cause a software error.
3. The malicious application in the container file is represented by its body and loader.
    a) An additional body loader is an actual executable sequence of bytes.
    b) An attachment body is a sequence of bytes that may contain infected data [6].

As a characteristic sign of a corrupted file in an unexecutable format, the presence of a portion of the data that is a valid sequence of bytes in the file in an unexecutable format can be used. It is this feature that forms the basis of the following works. Thus, the task of detecting malicious attachments is reduced to the task of detecting a bootloader, which is performed on a file in a non-executable format. To formalize it, the following mathematical model of attachment loader is built [7].

$B* = \{BL = b1, b2, \ldots, bL | bi \in N0, bi \leq 255 \leq i \leq L, L \in N0\}$ - be the set of ordered sequences of bytes. $B \in B*$ for we define a subjective view.

$$len: B* \to N : B \in B*, BL = b1, b2, \ldots, bL;\ len(B) = L; \quad (1)$$

its value is the number of bytes in the sequence B. Set of all possible command statements

$$O \subset B* \ \|O\| = N, o \in O, min\ len(o) > 0; \quad (2)$$

We mean the set of operator parameters. $P \subset B*: \|P\| = M,$

where M - the number of possible parameters.

Let K be the set of correctly executed instructions. At the same time, $K \subset RxO$, ya'ni $K = \{(o, p) \mid p \in P, o \in O\}$. E, $IA - 32$ be the limit for the minimum length of the byte sequence for

the exploit file loader, determined based on the processor's executable instruction standard. $X \in B, len(X) = F$ let the input file be an ordered sequence of X bytes [8].

where F is the length of the file. According to the rule, we introduce a non-commutative addition operation for B*:

$$B1, B2 \in \text{B}*, \text{B1}, + \text{B2}, = b1l, \dots, b1L, b2l, \dots, b1R;\ len(\text{B1}) = L, len(B2) = R; \quad (3)$$

$f((o,p)) = o + p$. about the rule f : $K \to B*$ function, then the exploit loader will take the form of the command chain Y from the X source file, that is

$$\exists\{kj\} \in K: Y = \sum(k\,j), Y \subset X, and\ len(Y) > \text{E}; \quad (4)$$

It should be noted that the presented model works with executable instructions, and the parsed file is a sequence of bytes or machine code. To transfer from machine codes to assembly language mnemonics, it is necessary to separate them into parts. The disassembly is based on static analysis of the byte sequence [9]. The sequence of bytes that make up the body of a non-executable file can also be thought of as a sequence of executable instructions that do not need to be correctly translated into an executable instruction mnemonic. However, applying the disassembly technique to the body of an executable file allows you to efficiently search for the shellcode in it. Thus, the general algorithm for detection of exploits can be described as a sequence of disassembling parts of the analyzed file and checking the results of the disassembly for compliance with the correctly executed loader model. Finding an executable loader in a file in a non-executable format is equivalent to solving the problem of dividing undefined sections of the program into code and data [10]. Figure 3. shows a schematic view of the problem of dividing ambiguous parts of programs into code and data. A similar problem occurs during disassembly and decompilation and is solved by appropriate software tools (Dynlnst, OllyDbg, IDA, Hiew, etc.). However, there is an important difference between the task of identifying a loader and the problem of clearly delimiting ambiguous sections of programs: when solving it, it is not necessary to clearly define the boundaries of code and data, it is enough to determine the existence of an executable loader. Research and development of ways to solve the problem of dividing ambiguous sections of programs into code and data have been considered by various researchers. The most successful results were presented by Getman A.M., Gaysaryan S.S., Avetisyan A.I., as well as Natan Rozenblum and Karen Hunt. Methods and classes of methods for solving the gap-filling problem can be divided into signature and heuristic methods, which include the use of function boundary templates based on the use of standard block heuristics and probabilistic methods. From the point of view of the task of detecting damaged files, the most interesting is the method proposed by I.Rozenblum using conditional random fields.

One of the main problems in using almost any gap-filling separation method is the so-called instruction stacking effect, described by I.Rozenblum and K.Hunt. This effect is due to the ambiguity of interpreting a sequence of bytes as a sequence of instructions, some of which are not 1 byte in length.

It should be noted that this effect is observed only in the calculation instructions of the complex instruction set of the processor architecture (Intel $IA-32$, as well as $IA-64$).

In his work, I.Rozenblum published the self-alignment theorem, which is used in practice to speed up data processing, and in fact, the disassembly process continues along the disassembled byte sequence, which means that the effect of loading instructions is equalized.

International Journal of Scientific Trends- (IJST)
ISSN: 2980-4299
Volume 4, Issue 4, April - 2025
Website: https://scientifictrends.org/index.php/ijst
Open Access, Peer Reviewed, Scientific Journal

Allows you to calculate the expected number of instructions, after disassembly, the traces of two disassembled sequences will merge based on their initial mismatch in bytes.

If the disassembled sequence B starts $0 < d < K$ bytes after the sequence A, then A and B are lined up, then the expected number of disassembled instructions in A is:

$$\sum_{j=1}^{K-1} N_{\text{dj}}; \quad (5)$$

Here $N = (Ndj)$ is the index size matrix.

However, when the formula is used in practice, there is a risk of losing instructions before convergence occurs. In this work, a lemma and a theorem on the application of the convergence effect were formulated and proved.

The n-step convergence probability vector of two disjoint sequences with different relative displacements of the starting points of disassembly can be calculated as $S_n = Q^{n-1} * R$, where: $R = (Q_{0,0}, Q_{1,0}, \dots, Q_{k,0})^T$ is the convergence probability vector in one step, $Q_{i,j}$ is the probability of transition from state *i* to a different state *j*.

The probability of convergence of two disassembly sequences in no more than n steps (without accepting the initial position) $Pn = (Q^{n-1}xR)xV^T, Agar\ V = (V_0, V_1, \dots V_k)^T$ are the probabilities of the initial differences in the positions of the sequences.

IV RESULT

The degree quantile a for the number of steps until the two chains converge is calculated as the minimum n with $P_n > a$. This result makes it possible to evaluate the applicability of the effect of instruction convergence in practice and to determine whether it is necessary to use the full partitioning method to find a real sequence of instructions with $O(N^2)$ complexity. Some instructions are good enough to ignore until they quickly approach $O(N)$.

With a sufficiently large length of the disassembled sequence, the probability that all chains starting within the length of the first instruction will converge completely is close to 1.

The conclusion from this investigation is that during disassembly, it is possible to skip the necessary instructions of the malicious add-on loader. Figure 3 shows a case where it is possible to skip fragmented instructions due to a sufficiently large number of convergence instructions.

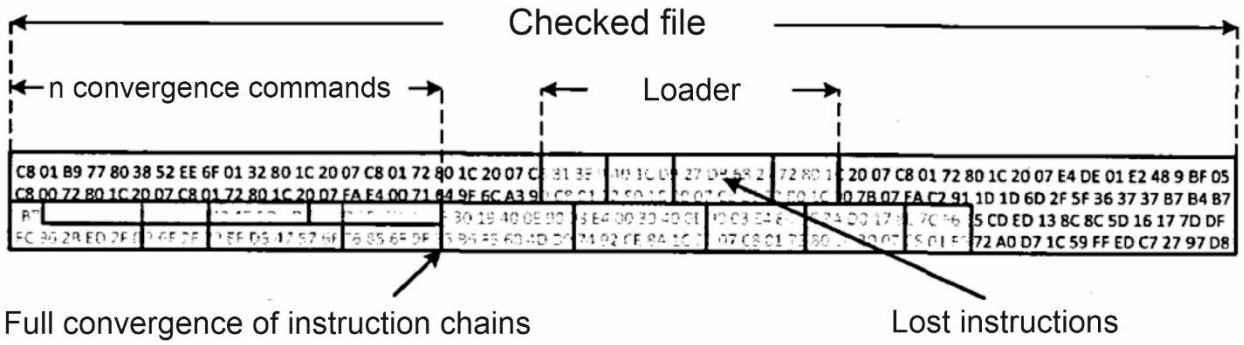

Figure 3. The ability to skip instructions.

Therefore, it is important to consider the possibility of complete convergence and avoid the possibility of missing instructions when searching for a malicious attachment loader by means of disassembly.

The difference between a corrupted file and an original file is that the internal integrity of the original data is broken and there is a slight change in the original file format. The consequence of

International Journal of Scientific Trends- (IJST)
ISSN: 2980-4299
Volume 4, Issue 4, April - 2025
Website: https://scientifictrends.org/index.php/ijst
Open Access, Peer Reviewed, Scientific Journal

this change is the incorrect operation of the software that processes the file and the violation of the execution of the command sequence by the attacker. The user agent is connected to the IP using malicious plugins. The structure of the damaged file is shown in Figure 4.

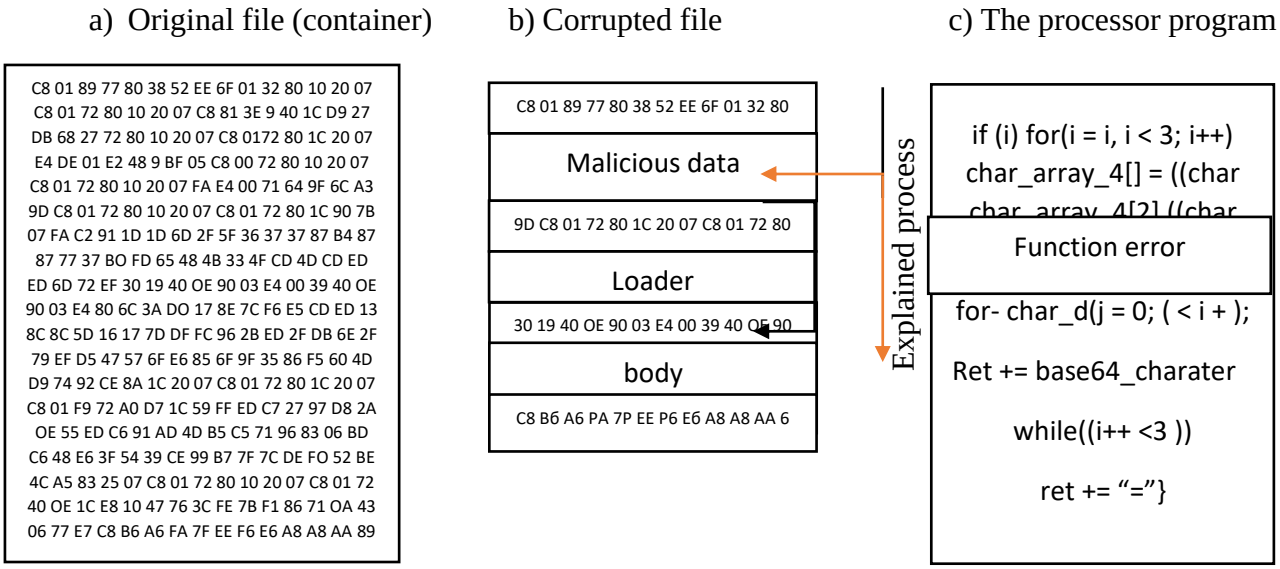

Figure 4. Structure of a damaged file.

Figure 4. An analysis of how the malicious code is added to the file and how it leads to an attack.
File = a set of contiguous bytes, $F = \{b_1, b_2, b_3, \ldots, b_n\}$; where $b_i \in \{0,1\}^8$, i.e., each byte is an 8-bit (1-byte) value.
Malicious file code:
F_o — original (harmless) file;
L — loader bytes;
M — malicious code (malicious payload);
k — position where malicious code is placed.
Then the infected file, $F_z = F_o[1{:}\,k-1] \parallel L \parallel M \parallel F_o[k{:}\,n]$; ‖ symbol - "concatenation" (concat) operator, that is, the infected file will be in the following order:
1. The beginning of the original file (from 1 to $k-1$);
2. The loader (L);
3. The malicious code (M);
4. The rest of the original file (from k to n).
The processor reads the file and executes it through the following function, $f(F[i]) = operatsiya(F[i]$, If the malicious code starts at $i = k$, then, $\exists i \in [k, k+|\,L\,|+|\,M\,|] \Rightarrow f(F[i]) \notin$, **is considered a security breach.**
Security violation model, If $f(F_z) \neq f(F_o) \Rightarrow$ The file is corrupted or dangerous, Or in other words:
$Integrity(Fz) = False$, If $\exists\, i \in [k, k+|\,L\,|+|\,M\,|] : f(F_z[i]) \notin safe\ domain$
safe domain — a set of safe instructions accepted by the processor.

International Journal of Scientific Trends- (IJST)
ISSN: 2980-4299
Volume 4, Issue 4, April - 2025
Website: https://scientifictrends.org/index.php/ijst
Open Access, Peer Reviewed, Scientific Journal

Table 1. Block view of the corrupted file.

Part	Byte spacing	Description
F_o[1:k-1]	1 from $k-1$ to	Original file
L (Loader)	k from $k+l-1$ to	Loading code
M (Malicious)	$k+l$ from $k+l+m-1$ to	Harmful code
$F\_o[k:n]$	$k+l+m$ from n to	Remainder of the file

A deep understanding of the attack mechanism was achieved.

Accurate mathematical formulas have been developed to detect malicious files. This model will serve as the main theoretical foundation for the development of automatic intrusion detection systems (for example, IDS - Snort) in the future.

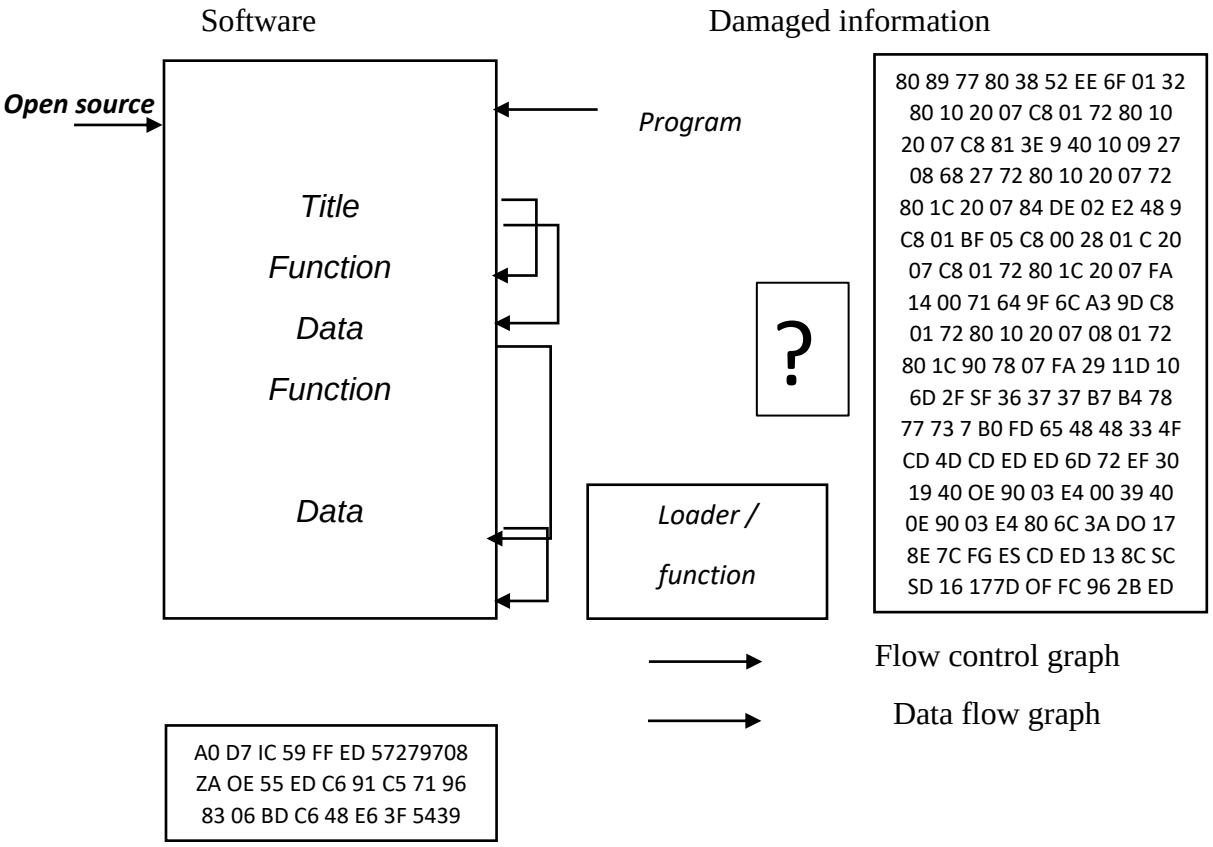

Figure 5. Problems of the gap filling problem of dividing code and data.

CONCLUSION

Among the modern information security problems, identifying malicious programs and effectively combating them is one of the urgent issues. In this study, the problem of improving the effectiveness of existing tools by using hidden Markov models (HMM) in detecting infected files was considered. Research results show that systems built on the basis of HMM study the sequences

of hidden behavior in files and show high accuracy in detecting anomalies from them. The structural structure of the files and their performance characteristics were analyzed in the detection of malicious activity using hidden Markov models. With the help of models, the possibility of accurate and efficient classification has appeared, which serves to reduce the level of false positive and false negative signal in security systems. It was also confirmed through experiments that the HMM-based approach has significant advantages over traditional signature detection methods. This approach not only detects existing malicious files, but also detects previously unknown malicious activities. In conclusion, it can be said that HMM-based approaches to detecting infected files are a promising and effective technology for strengthening information security. In the future, the integration of such models with machine learning methods will take an important place in the development of real-time detection systems.

References

[1] Marcus Botacin, Felipe Duarte Domingues, Fabrício Ceschin, Raphael Machnicki, Marco Antonio Zanata Alves, Paulo Lício de Geus, André Grégio, AntiViruses under the microscope: A hands-on perspective, ISSN 0167-4048,: https://doi.org/10.1016/j.cose.2021.102500., 2022.

[2] СПОСОБ ГГОВЫШЕШШ ЭФФЕКТИВНОСТИ СРЕДСТВ ВЫЯВЛЕНИЯ ЗАРАЖЕННЫХ ФАЙЛОВ НА ОСНОВЕ ИСНОЛЬЗОВАИИЯ СКРЫТЫХ МАРКОВСКИХ ¡МОДЕЛЕЙ, Эдел Дмитрий Александрович, Ростов-на-Дону, 2013.

[3] Jiaru Song, Guihe Qin, Yanhua Liang, Jie Yan, Minghui Sun,, DGIDS: Dynamic graph-based intrusion detection system for CAN,, ISSN 0167-4048,: https://doi.org/10.1016/j.cose.2024.104076., 2024.

[4] Carlos Henrique Macedo dos Santos, Sidney Marlon Lopes de Lima,, XAI-driven antivirus in pattern identification of citadel malware,, ISSN 1877-7503,: https://doi.org/10.1016/j.jocs.2024.102389., 2024,.

[5] Rajasekhar Chaganti, Vinayakumar Ravi, Tuan D. Pham,, Deep learning based cross architecture internet of things malware detection and classification,, ISSN 0167-4048,: https://doi.org/10.1016/j.cose.2022.102779., 2022.

[6] Timothy McIntosh, Paul Watters, A.S.M. Kayes, Alex Ng, Yi-Ping Phoebe Chen,, Enforcing situation-aware access control to build malware-resilient file systems,, ISSN 0167-739X,: https://doi.org/10.1016/j.future.2020.09.035., 2021.

[7] Manuel Navarro-García, Vanesa Guerrero, María Durban, Arturo del Cerro,, Feature and functional form selection in additive models via mixed-integer optimization,, ISSN 0305-0548: https://doi.org/10.1016/j.cor.2024.106945., 2025.

[8] Rui Liu, Xiaoli Zhang,, Generating machine-executable plans from end-user's natural-language instructions,, ISSN 0950-7051: https://doi.org/10.1016/j.knosys.2017.10.023., 2018.

[9] Muhammad Mudassar Yamin, Basel Katt,, Modeling and executing cyber security exercise scenarios in cyber ranges,, ISSN 0167-4048,: https://doi.org/10.1016/j.cose.2022.102635., 2022.

[10] Hasan H. Al-Khshali, Muhammad Ilyas,, Impact of Portable Executable Header Features on Malware Detection Accuracy,, ISSN 1546-2218,: https://doi.org/10.32604/cmc.2023.032182., 2022.