METHODS AND MODELS OF RELIABLE SOFTWARE PROTECTION SYSTEMS

The article deals with the problems in constructing a protection system of executable code. The techniques of breaking the integrity of executable code and ways to eliminate them are described. The adoption of virtual machine technology in the context of executable code protection from analysis is considered. The substantiation of the application of virtual machines as the best way to oppose the analysis of executable code is made. The protection of executable code by transferring the protected code in a virtual execution environment is considered. An efficient implementation of the method is proposed. Also, general characteristics of information systems software life cycle are specified in this paper, considered software protection applications reliability questions and use of fail-safe ensuring programming. In addition there presented some known models used for software debugging and operating.

INTRODUCTION

The information processing in computer systems and networks is the problem of to-day (and it'll be still urgent for a long time). In particular, the task of code and processed data protection is still not solved. The main objectives of the code protection program are:

- Preventing unauthorized changes in the program behavior algorithm.
- Protection of processed data against an illegitimate user.
- Ensuring of the software protection from unauthorized copying.

The information processing in computer systems and networks is the problem of to-day (and it'll be still urgent for a long time). In particular, the task of code and processed data protection is still not solved. The main objectives of the code protection program are:

- Preventing unauthorized changes in the program behavior algorithm.
- Protection of processed data against an illegitimate user.
- Ensuring of the software protection from unauthorized copying.

Ideal protection in the field of software is still not created, although there has been a war for decades between protection creators and hackers. Invariably the latter are the victors, yielding the palm to protection creators only by chance. Under the protection the authors mean a unit of executable code, which counteracts (successfully or not) the illegal use of the program. This can be a protection from copying of storage device, input of the serial registration number of the program, means that limits the maximum number of licenses in the network and so on.

Next, let us consider the methods and procedures by which the above objectives are achieved. NOTA BENE that none of these goals at the moment is reached and the corresponding scientific and technological problems are not fully resolved. Since it makes no sense to reinvent the protection against unknown threats, not knowing where, how, and when the next attack will be made. We shall consider the protection means, starting with the attack tools. In other words, we will make the analysis starting with the threat and moving to the protection means, speaking first on the side of the attacker, and then on the side of information security specialists.

1. THE URGENCY OF THE PROGRAM CODE PROTECTION

There is a huge field of activity for hackers in the field of program behavior change algorithm. The fact that the majority of program code is in the open form, i.e. have a view of either the machine code or some intermediate byte code (assembler program) executed by the virtual machine.

The main problem faced by programmers creating protection code (that portion of the code that makes the decision to open or close access to the main code, in other words, performs the functions of identification, authentication and authorization) is a problem of potential capacity of analysis and correction of definite binary code. Moreover, this vulnerability, according to the authors, is one of the most problematic in the field of software protection.

Today, more than one solution of application code encryption allowing it to be secured from disassemble and connected to the dongle, for example, Guardant, is widely known. Take the most popular method of encryption. Encode important sections of code so that they can be correctly decoded only through legally obtained key. Keyless application ceases to be operational, or work during the evaluation version. Encoding algorithm will be a test of legitimate use. Conditional jump IF “decoded_correct “THEN ...

- It’s just a formality that will eliminate an access violation when
- Performing an incorrectly decoded area application crypto remains vulnerable to attack when an intruder is buying a licensed copy and running it removes from memory dump decoded version. Consequently, immediately after the execution of the decoded area application requires encode it again, or move back to pre-stored encrypted portion. Cracker will have the only opportunity to catch the moment when in the memory the decoded portion is executed, keep working application code and go on acting in such a way, eventually collecting application into a whole piece by piece, but this is a manual hard work, especially if you provide a large number of small areas of code, with its consistent decoding. The fight against this kind of attack is the largest possible number of coded sections, replacing them in each new version of the system [1].

There are created enough of disassemblers for machine code [2], able to provide a program in a convenient form (in assembly language) for review. Thus, by changing at least one machine instruction, we obtain the following possible outcomes:

- inefficient code;
- postponed inoperative code (from a few seconds to several hours or days);
- misbehaving code;
- deliberately misbehaving code;
- The consequences of such events are obvious. And the most pitiable result we expect in the latter case. Depending on the domain software, there may be the loss of large sums of money in the calculations, allowing system access to illegitimate user, and so on - the list is endless.

2. ENSURING THE INTEGRITY OF THE CODE AND DATA SECURITY

There are several security solutions for the PC. First, use the executable packers such as UPX [3]. However, it stops only an
The main idea of the program by using a threaded code - an array of subroutine calls (Fig. 2). The implementation of threaded code, the method of storage of these calls may be different. This code can be processed by the interpreter (which was confirmed in the name of the address interpreter), or may be a simple sequence of machine instructions subroutine call. Some set of basic routines virtual machine using threaded code are implemented as subroutines written in the usual native [6].

Thus, program code execution speed is preserved, and simultaneously its security is increased. Theoretically, while performing a decryption just one instruction at "time" helps to protect the code with the probability \( p = 100\% \). In other words, the computer memory at a given time \( t \) will be one instruction \( k \). This will make the reversed engineering of the code impossible.

The above described technology can be implemented in different variations. As an example, consider the following sequence of steps:

1. As the encryption code key can act a certain license (key file), issued to the user (an unauthorized copying is limited).
2. The entire protected program code contains a security code fragments, dispersed by using a pseudorandom sequence generator that executes VMs, directly embedded in the main program.

As a result of these actions the original program code is "mixed" with the security code, which, in turn, is represented as a sequence of encrypted data, deciphering of which for performing is made according to one instruction, and as a result, disassembling of the security code is not possible. In this case, although the disassembling of machine code programs is still available to attacker, but does not give the desired result, because separation of the original code from the embedded security code becomes much time consuming task, not always with predictable results.

To achieve even higher degree of protection the partial implementation of the program in a virtual machine protection subsystem is possible, for example, take this code 5-10\% of the total program taking 1-2 % runtime - it will not lose speed, but significantly strengthen hacking, reducing it almost to nothing (according to the particular case of the empirical law of Pareto, for 80 % of the time, the processor performs 20\% of the total sold in the program commands).

To further complicating of the analysis code one can apply the so-called "method of mixing" code. Let the command of virtual machine consists of the command code and a pointer to the next command to be processed (while the command code can be encrypted using a key as the current and / or next pointer):

```
Struct command
{
    CODE * code;
    Struct command * next_command;
}
```

Thus, the virtual machine's memory will be a linked list, which will allow to: a) mix easily the executable code, and b) effectively add and delete meaningless (so-called junk) command. As a result, we obtain extremely simple, but very nontrivial polymorphic code, to analyze which hackers have to write a special converter. You can also implement the virtual machine memory in the form of a binary tree, pass the arguments through lists, instructions' executing organize as a deck.
Consider the most stable version of the defense - virtual machines with randomly generated set of instructions. Here any stable combination of commands can be replaced by a single compound instruction (this also minimizes the amount of executable code). If we go further, you can take a few random instructions, not even necessarily adjacent, and replace them with a new one, so continuing indefinitely. In the second generation of instructions, there would be a great choice for division instructions. Provided that such fragmentation will be succeeded by chance, the received instructions won't most likely coincide with the source ones, and as a result, the second generation of VM instructions will be practically unrecognizable [7].

At the same time to analyze the logic of the virtual machine is incredibly difficult. The matter is that the logical sense of random generated commands in most cases is not clear, since the number of connections between the separate commands tends to n!, Where n - number of basic instructions. For the virtual machine can handle hundreds or even thousands of commands, nor analyzing, nor disassembling of the code is possible - it all comes down to a non-standard VM algorithm of its behavior [7].

Based on the above assumptions and statements, let's make an intermediate review and present a generalized scheme of application protection (Fig. 3).

3. METHOD OF VIRTUAL MACHINE IN PROTECTION OF SOFTWARE

As you know, the perfect way to protect software does not exist, therefore, security systems developers do not seek to prevent a hacker from the possibility of security neutralization, but try to complicate this process.

Protection can solve one or a complex set of tasks such as protection from copying, illegal use, modification, etc., but despite the kind of the ultimate goal standing in front of such a product, the developers of each of them must first solve the general problem of everybody - quality protection from the study. Whatever algorithms are used for software protection, their resistance to reverse engineering determines the resistance of the entire security system as a whole.

Today in the market there are a large number of commercial protections, but many of them, including still popular ones, have been broken for a long time. Often a protection from the study brings their weak. After analyzing the intruder protection algorithms, serial keys are generated, the hardware - successfully emulated. The situation could be improved be the developing of an effective method of fixing the software protection from the study, applying which to algorithms of other protections it would be possible to raise their level of quality.

Consider the methods existing to protect software from the study:

- Entanglement - artificial complication of code in order to make it more difficult readability and debugging (code mixing, implementation of false procedures, sending extra parameters in procedures, etc.)
- Mutation - created every time you start up tables of operation corresponding, the operations themselves are replaced by synonyms.
- Compression, encryption - the original program is packed / encrypted, and produces a reverse process as you complete.
- Processors simulating - creates a virtual processor, the protected program is compiled under him, and executed on the target machine using the simulator.

There are the other methods as well as their combinations and variations. However, it's easy to see that they are all based on one simple idea: redundancy. In fact, what is obfuscated as not excessive coding program? It's the extra transitions, additional parameters, the extra instructions - keyword method "extra". The same applies to any of the above methods, and probably it would be natural to combine all of these methods in the same group "methods of redundant coding". What is the advantage of redundancy, in spite the fact that it increases the size of the program and reduces its speed? The matter is, that in all these protecting species the understanding of "human factor" is used - the harder the person understands the logic of a process, the more resources this process uses.

For example, the functionality of one simple instruction of constant downloading to register can be "smashed" into dozens or even hundreds of instructions, and to trace the connection of all used resources (registers, memory, etc.) in this sequence is difficult for a person. The encryption method from this point of view is not unique - as in the same way as in the other methods for simple instruction (or group of ones) an abundant sequence of commands is required - in this case, these are decryption operations and plus the decrypted code operations.

However, what is automatically "confused" or complicated, it can be also automatically restored to its original state - developers of obfuscation mechanisms usually develop in parallel "unsmashing" tools, and methods of mutation and encryption imply the content of the reverse mechanism in the protected code. Alone in this group of methods is worth only method of simulation of a virtual processor, which firstly leads to a high degree of involvement and irreducible result code, and secondly (by a certain approach to implementation), the protected code does not contain an explicit recovery methods of the original code. We consider this method in detail.

4. VIRTUAL PROCESSOR IN SOFTWARE PROTECTION

The essence of the method is as follows: some functions, modules, or program entirely, are compiled under a virtual processor with an unknown potential attacker command system and architecture. The execution is provided by embedded in the resulting code simulator.
Thus, the task of reengineering of protected fragments is reduced to the study of simulator architecture, simulated processor, creating a disassembler for the latter, and, finally, the analysis of disassembling code. This task is not trivial, even for those with good knowledge and experience in working with the architecture of the target machine. Same attacker has no access to the description of a virtual processor architecture, nor to information on the organization of the simulator used. Cost of breaking increases significantly.

Why, concerning a high theoretical efficiency, this method is still not widely used? Probably, it is not done for two main reasons. First, the method has features that narrow the field of its potential application - this will be discussed below. Secondly, and it's perhaps more serious reason, the complexity (and therefore the cost) of implementation of the method is very high. If we consider the fundamental possibility of information leakage about the newly created system, which immediately leads to its inefficiency and depreciation, it becomes clear why manufacturers of security software are not in a hurry to implement this method. However, it should be noted that with certain variations and limitations this method is implemented in all the latest products such as StarForce3, NeoGuard, VMProtect, etc. Apparently there will be created more and more products, and existing ones will be developed, so as emerging realizations confirm the high efficiency of the method, despite the fact they have got weaknesses.

5. IMPLEMENTATION OF THE METHOD

One disadvantage of this method is a high cost of implementation, but it can be considerably reduced. In the ground of the system of protection, that implements this method, would lie a compiler of a high-level language. There is the only necessary machine-dependent phase in any compiler - code generation, the dependency in other phases, as a rule, can be eliminated.

If the compiler was originally developed as a multi-platformed, the process of migrating to a different target platform is usually simplified. For example, this can be achieved with automatic generation of code generation according to special description of the target machine. In this case, to change the platform the developers only need to modify this description. But even if there is no own compiler one can use the freely distributed open source software, for example, GCC.

And to simplify the user work with the described system of protection, it can be provided with the mechanisms of embedding in popular development environments, such as MSVC. In this case, the scheme of such a complex would look like it is shown in Fig. 4.

The specific use of the compiler imposes a number of specific requirements for the virtual processor, nevertheless they can be easily implemented. There are not many requirements - one need only to provide access to external, according to the virtual machine memory, as well as the ability to call external functions - it is necessary for the interaction of protected and unprotected code. The rest of the architecture of a virtual processor can be completely free and the more confusing and original it’ll be, the higher level of protection will be achieved.

Accordingly, the functioning in such manner secured product would occur in the scheme in Fig. 5.

The compiler itself, apart from changing cogenerating phase, need to be finalized for the acquisition the possibilities for it:

1. Distinguish access to internal and external memory relatively to the virtual processor (including function calls).

2. Create for each protected function so-called shell running on a real processor, with a call of protected function through simulation.

![Software protection scheme](image)

Figure 4. Software protection scheme

The last opportunity should be described in details. As reflected in the scheme in Fig. 4, functioning of the protected functions will be implemented through a call to the simulation, indicating which the security functions need the interpreting. However, before that, you need to perform special code that prepares the protected function parameters - "move" them with real registers and virtual memory, the method according to the architecture of the virtual processor. All these will deal with special functions generated by our compiler - "shell". Shell, in turn, will use the special features of the simulator to access virtual registers and memory. Characteristically, our compiler will generate a shell in a high level language, which, in turn, will compile the standard compiler, used by the user to build the unprotected part of your project.

Of course, the specific implementation method of a virtual processor may be somewhat different from the described one as the scheme of protected product work. Nevertheless, the described variant is quite viable, and, furthermore, is relatively simple.

The heart of a protected product is a simulator. It will be included in any assembly of the protected product. But, we shall not consider its implementation, as special requirements to it are not presented - it should only simulate the architecture of our virtual processor operations, including access to external memory. However, it should be noted that according to the nature of its application, it is necessary to automate maximum the adjusting process to the new virtual simulator architectures.
Disadvantages of the method are consequence of its advantages:

a) The speed of the transferred code to a virtual environment (approximately 10-50, depending on the architecture of a virtual processor and simulator) is lower than the original code.

b) The volume of protected software will usually be somewhat higher than the unprotected one.

However, the last drawback is irrelevant, since the size will increase slightly, and in some cases may even decline. The first drawback is principle and imposes some obvious limitations to the use of the method.

6. RELIABILITY OF INFORMATION SYSTEMS SOFTWARE

Programs for modern information systems can accrue a significant number (hundreds, thousands, tens or hundreds of thousands) of simple commands. For many reasons at writing programs the errors may be revealed. In the environment of programmer’s humor, they say, there is no software without bugs, but there are programs with not found errors. Blunders are detected at the stage of working out the programs, but to check the program totally according to all possible modes is not usually viable, as a result there is no conviction that all errors are found. There is a statistical approach to the process of errors detecting during the programming analysis is proposed.

Such process can be characterized according the function

\[ f(t) = \frac{\int_{0}^{t} f(t)dt}{R} \]  

(1)

Where \( f(t) \) - number of identified and corrected errors per time unit in the program, which has \( R \) - number of commands

\[ \frac{f(t)}{R} = \frac{d\varepsilon_n(t)}{dt} = \frac{\varepsilon_n(t + \Delta t) - \varepsilon_n(t)}{\Delta t} \]  

(2)

Where \( \varepsilon_n(t) \) - number of identified and corrected errors during time \( t \) per one command.

Accordingly,

\[ \varepsilon_n(t) = \frac{1}{R} \int_{0}^{t} f(t)dt \]  

(3)

Function \( f(t) \) can be determined experimentally during pilot testing of program by means of fixing the detected errors number. Problem of \( f(t) \) definition is simplified if

\[ f(t) = \frac{\varepsilon_0 e^{-\frac{t}{\tau_0}}}{\tau_0} \]  

(4)

Where \( \varepsilon_0 \) and \( \tau_0 \) are parameters \( f(t) \) which are determined during working out.

Then

\[ \varepsilon_n(t) = \frac{1}{R} \int_{0}^{t} f(t)dt = \frac{\varepsilon_0 (1 - e^{-\frac{t}{\tau_0}})}{R} \]  

(5)

At \( t \to \infty \), \( \varepsilon_n(\infty) = \frac{\varepsilon_0}{R} \) or \( \varepsilon_0 = \frac{\varepsilon_n(\infty)}{R} \).

From this point it follows that \( \varepsilon_0 \) is the total number of errors in the program prior to testing. The testing process cannot last too long hence some errors will always remain in the program

\[ \varepsilon(t) = \frac{\varepsilon_0}{K} - \varepsilon_n = \frac{\varepsilon_0}{R} e^{-\frac{t}{\tau_0}} \]  

(6)

Where \( \varepsilon(t) \) - the number of errors found per one command.

If to anticipate that errors are uniformly distributed throughout the program, the occurrence probability of errors \( P(t) \) during \( \Delta t \) will be proportional to the tempo pace \( \delta \) of information system (that is the average number of commands changes per time unit) and to the number of errors left in the program, that is

\[ P(t) = \varepsilon(t) \delta \Delta t \]  

(7)

Drawing the analogy between the process of errors and failures of objects \( P(t) = \lambda \Delta t \) it may be concluded that the intensity of errors \( \varepsilon(t) \) does not depend on time \( t \) but is determined only by the interval \( \Delta t \) at which the probability of error is calculated. From this point the operation time till "failure", which is due to the error emergence in the program, will make

\[ T(t) = \frac{1}{\varepsilon(t) \delta} = \frac{R}{\varepsilon_0 \delta} e^{\frac{t}{\tau_0}} \]  

(8)

Analysis of \( T(t) \) changes can serve as a basis for the program working-out operation timing \( \tau \) choice, namely, the testing ends when the value \( T(t) \) becomes large enough.

In case when it is possible to estimate material losses \( C_n \) via occurrence of errors in the calculations, then the operation time testing \( \tau \) can be quantified as the following. If during the time \( T_p \) of program operation, it fails \( \frac{T}{T(\tau)} \) times, it will cause a total loss
The process of program testing requires a proper time of computations and other expenditures associated with it. If to mark \( C_0 \) as a per hour testing, then during time \( t \) the total time required will be \( C_0 \tau \). Accordingly, the total loss \( C \) because of errors and the time required for working testing of programs will be

\[
C = \frac{C_0 T_p}{T(\tau)} + C_0 \tau
\]

From this point

\[
\frac{dC}{d\tau} = \frac{-C_0 N p e^{-\tau_0}}{R \tau_0} e^{-\tau_0} + C_0 = 0
\]

or

\[
\tau_M = -\tau_0 \ln \frac{C_0 R \tau_0}{C_0 T_p e^{-\tau_0}}
\]

Where \( \tau_M \) - time of working out which will provide the minimum \( C \).

In this case, when it is required to eliminate an error in the program it is advisable to use the "backup". Here the particular problem is solved by several programs, each of them is developed by independent teams of programmers and in its basis lay various algorithms, furthermore, the results of programs computations are compared, and they are considered true if they match. Since the errors emergence in software is an improbable event, the occurrence of two or more of such events is practically impossible.

7. USING OF FAIL-SAFE PROGRAMS IN SOFTWARE PROTECTION

Fail-safe programs are designed usually by frequent repetition of calculations at the levels of micro-operations, operations, commands, program modules or the entire program.

To improve reliability against the failures of the entire information system repeated execution of programs is widely used at the level of program module. Its essence is that the program is split into several modules, each of them is executed twice, and the results are compared. If the results of the first and second calculations coincide, it is considered that the obtained results are true and then one may proceed to the next step of (operational) calculations. If they do not coincide, the computation is repeated until the two received results will be the same. The substantial advantage of this method is its simplicity. At drawing up the program it is required only to provide the appropriate actions, the method does not require the additional costs for hardware. The disadvantages of this method are: it takes twice more of the required time for problems solving; it is unable to detect errors caused by the failures.

The performance of information system at using the method of double execution depends on the number of modules into which the program is to be split. Indeed, the greater length of the modules determines also the fairly large probability of failures. So, instead of two, it will be required to repeat computations three or more times, which will increase the problem solution time. On the other hand, at small length of modules, most of time will be spent on comparing and recording the calculations results executed within individual program modules into the memory devices.

In this regard, there emerges a problem of finding an optimal number of modules into which a program should be split, namely in such way that the time \( T_p \) for problem solving will be minimal. We introduce the notation: \( T \) - time for solving of problem at a single execution of the program; \( t \) - duration of calculation at a single module; \( p(t) \) - the probability of no failure during time \( t \).

Then the ratio \( \frac{T}{t} \) will determine the number of modules onto which should be divided a program. We can determine the probabilities of two-, three-, or even \( i \) times execution repetition for any program module. If the failures are independent events, then the probability that a given program module will be executed twice will be equal to the probability of no failure at the first and second executions, so that

\[
p_2(t) = p^2_i(t)
\]

Subsequently, the probability \( p_i(t) \) at fixed \( t \) will be denoted as \( p_i \).

Similarly, \( g \) is the probability that in one of the two preceding calculations has occurred a failure, but at the third computation was obtained a correct result, that is

\[
p_3 = 2p_i^2(1 - p_i) = 2p_i q
\]

Where \( q = 1 - p \). In general, \( p_i \) equals to the probability that at the \( i \)-st and in one of the preceding calculations the failures were absent, and in the others, already passed, the failures were presented, that is

\[
p_i = (i - 1) p_i^2 q^{i - 2}
\]

Thus, the average number of computing will be equal

\[
A = \sum_{i=2}^{n} p_i = \sum_{i=2}^{n} i(i - 1) p_i^2 q^{i - 2}
\]

It is easy to show that \( A = \frac{2}{p_i} \). Hence we have

\[
A = \frac{2}{p_1}
\]

Time \( T_3 \) which is required to conduct comparisons and to write intermediate calculations into the memory device, depends on the type of memory storage device used, on the number of intermediate results \( k \) and on the number of steps \( t \) of the program, that is

\[
T_3 = T \frac{f(k, T \frac{t}{T})}{t}
\]

Where \( f(k, T \frac{t}{T}) \) - the average time of comparisons operations and of the recourse to the memory device for one module of program results recording. If we assume that \( f(k, T \frac{t}{T}) = \text{const} = a \) then
The system design, we may find:

\[ T_p = \frac{2T}{p(t)} + \frac{T_0}{t} = T \left( \frac{2}{p(t)} + \frac{a}{t} \right) \]  \hspace{1cm} (15)

For some types of information systems it was experimentally found that

\[ p(t) = e^{-\lambda t} \]  \hspace{1cm} (16)

Where \( \lambda \) - the intensity of failures. In this case, \( T_p \) accepts the minimum value for \( t \), which can be determined from the equation

\[ \frac{dT_p}{dt} = 2\lambda e^{\lambda t} - \frac{a}{t^2} = 0. \]  \hspace{1cm} (17)

Thus the value of \( T_p \) can determine the optimal length of the program section and corresponding to it number \( \frac{T}{t} \) of stations at which \( T_p \) will be minimal.

The cause of incorrect functioning of the information system may be also a presence in it of so-called virus software programs designed to insert undue distortions into computations, deleting files and creating conditions for the abnormal functioning of information system.

8. ESTIMATION OF SOFTWARE RELIABILITY ACCORDING TO THE RESULTS OF ADJUSTING AND NORMAL OPERATION

In the processes of software debugging, normal and research operation, it becomes achievable to use statistical data about the detected and corrected errors in order to refine the system design reliability assessments. For this purpose the reliability models, containing parameters, point estimations of which are obtained at the software commissioning and operation results processing, were developed. These models differ with their assumptions about the dependence of the intensity of errors emerging during the time of adjustment and operation. Some of these models contain specific requirements for the software modules internal structures.

Exponential model by Schumann is based on the following assumptions:
- The total number of commands in the program of machine language is invariable;
- at the beginning of tests, the number of errors is equal to some constant value, at length of corrections, the number of errors becomes smaller, and new mistakes are not made in the course of program correcting;
- Program failure rate is proportional to the number of remaining errors.

Regarding the structure of the program module there are made the following assumptions:
- module contains only one cycle operator in which are resided operators for information input, assignment operators and operators of controls in advance conditional transfer;
- Nested loops are absent, but there can be present \( k \) parallel paths, if we have the \( k - 1 \) controls conditional transfer operator.

At the given above mentioned assumptions the probability of faultless operation is given by the formula

\[ R(t, \tau) = \exp(-Ce_r(\tau)t) = e^{-\lambda t} \],

\[ \varepsilon_r(\tau) = \frac{E_0}{I} - \varepsilon_s(\tau) \],

\[ T = \frac{1}{C} \left( \frac{E_0}{I} - \varepsilon_s(\tau) \right) \]  \hspace{1cm} (18)

where \( E_0 \) - number of errors at the beginning of adjustment; \( I \) - the number of machine instructions in the module; \( \varepsilon_s(\tau) \), \( \varepsilon_r(\tau) \) - number of corrected and left errors per one command; \( T \) - mean time between failures; \( \tau \) - time of adjustment, \( C \) - coefficient of proportionality.

To assess the \( E_0 \) and \( C \), there are used the results of adjustments. Assume that among the total number of the system test programs runs the \( r \) is number of successful runs, while \( n - r \) - the number of runs that were interrupted by errors. Then the total time of \( n \) runs, intensity of errors and operating time per an error can be found by

\[ H = \sum_{i=1}^{r} T_i + \sum_{i=r+1}^{n} t_i \],

\[ \lambda = \frac{n - r}{H} \]

\[ T = \frac{1}{\lambda} = \frac{H}{n - r} \]  \hspace{1cm} (19)

Assuming that \( H = \tau_1 \) and \( H = \tau_2 \), we may find:

\[ \hat{\lambda}_1 = \frac{n_1 - r_1}{H_1} \]

\[ \hat{\lambda}_2 = \frac{n_2 - r_2}{H_2} \]

\[ \hat{T}_1 = \frac{1}{\hat{\lambda}_1}, \hat{T}_2 = \frac{1}{\hat{\lambda}_2} \]  \hspace{1cm} (20)

Where \( \hat{T}_1 \) and \( \hat{T}_2 \) - test time for one error. Substituting here (18) and solving the system of equations, we obtain parameters for the model estimations:

\[ \hat{E}_0 = \frac{I}{\gamma - 1} (\gamma \varepsilon_s(\tau_1) - \varepsilon_s(\tau_2)) \]

\[ \hat{C} = \frac{1}{\hat{T}_1} \left( \frac{\hat{E}_0}{I} - \varepsilon_s(\tau_1) \right) \]  \hspace{1cm} (21)
To compute the estimations we need, according the results of the adjustment, to learn the parameters $\hat{T}_1, \hat{T}_2, \hat{\varepsilon}_n(\tau_1)$ and $\hat{\varepsilon}_n(\tau_2)$.

Some generalization of results (19) - (21) looks as the following. Let $\hat{T}_1$ and $\hat{T}_2$ are times of system operation that correspond the adjustment time $\tau_1$ and $\tau_2$; $n_1$ and $n_2$ - number of errors detected in the periods $\tau_1$ and $\tau_2$. Then

$$T_1 = \frac{1}{n_1} \left( \frac{E_0}{I} - \hat{\varepsilon}_n(\tau_1) \right)$$

$$T_2 = \frac{1}{n_2} \left( \frac{E_0}{I} - \hat{\varepsilon}_n(\tau_2) \right)$$

Hence

$$\hat{\varepsilon}_o = \frac{I}{\gamma - 1} \left( \gamma \hat{\varepsilon}_n(\tau_1) - \hat{\varepsilon}_n(\tau_2) \right)$$

$$\hat{\varepsilon}_n(\tau_1) = \frac{n_1}{T_1} \left( \frac{E_0}{I} - \hat{\varepsilon}_n(\tau_1) \right), \quad \gamma = \frac{1}{n_1} / n_2$$

If $T_1$ and $T_2$ - solely the total time of adjustment, then $T_1 = T_1 / n_1$, $T_2 = T_2 / n_2$, and formula (22) coincides with (21).

If during the adjustment course it is made $k$ tests at intervals $(0, \tau_1), (0, \tau_2), ..., (0, \tau_k)$, where $\tau_1 < \tau_2 < ... < \tau_k$, then to determine the maximum likelihood estimation is used equation

$$\hat{\varepsilon}_n(\tau_j) = \frac{\sum_{i=1}^{j-1} n_i}{\sum_{i=1}^{j} n_i} \left( \frac{E_0}{I} - \hat{\varepsilon}_n(\tau_j) \right)$$

$$\hat{\varepsilon}_n(\tau_j) = \frac{\sum_{i=1}^{j} n_i}{\sum_{i=1}^{j-1} n_i} H_j$$

Asymptotic values of the estimations variance (for large values $n_j$) are determined according the expressions

$$\text{D}(\hat{\varepsilon}_n(\tau_j)) = \frac{1}{\sum_{i=1}^{j} n_i} \left( \frac{E_0}{I} - \hat{\varepsilon}_n(\tau_j) \right)^2$$

$$\text{D}(\hat{\varepsilon}_n(\tau_j)) = \frac{1}{\sum_{i=1}^{j-1} n_i} \left( \frac{E_0}{I} - \hat{\varepsilon}_n(\tau_j) \right)^2$$

Where $n_j$ - the number of runs of $j$-st test that ended with failures, $H_j$ - time which spent on the execution of successful and unsuccessful runs of the $j$-st test. At $k = 2$, the (23) reduces to the previous case and its solution gives the result (22).

Exponential model by Dzelinsky-Morandi is a particular case of Schumann model. According to this model, the errors emerging intensity is proportional to the number of residual errors:

$$\lambda(\Delta t_i) = K_{sm} (E_0 - i + 1)$$

Where $K_{sm}$ - coefficient of proportionality; $\Delta t_i$ - interval between the $i$-th and $(i-1)$-th errors.

Reliability of failure proof operation then is

$$R(t) = \exp(-\lambda(\Delta t)) = \exp(-K_{sm} (E_0 - i + 1))$$

$$R(t) = \exp(-K_{sm} (E_0 - i + 1))$$

At $K_{sm} = C / I$ and $\varepsilon_n(\tau) = (i - 1) / I$ formula (27) coincides with (18). In order to obtain maximally likelihood estimation for the parameters $E_0$ and $K_{sm}$ at sequential observation of $k$ errors in the time moments $t_1, t_2, ..., t_k$, we need to solve the system of equations

$$\sum_{i=1}^{k} (E_0 - i + 1) = k / (\hat{E}_0 - i + 1)$$

$$K_{sm} = \frac{k}{A} / (E_0 - \theta \cdot k + 1)$$
A fundamental new approach to creating systems of protection of executable program code has been developed. Method of software protection based on virtual machine has been proposed. A high resistance to cracking the security subsystem has been proved.

For the reliability index is taken probability $R(n)$ of fail-safe executions of $n$ program runs. For $j$-st run, the probabilities of failure are as following

$$Q_j = \sum_{i=1}^{N} p_{ji} y_i,$$

where $y_i$ - an indicator of failure at $i$-st set of data; $p_{ji}$ - the probability of $i$-st set at the $j$-st run.

Therefore

$$R(n) = \prod_{j=1}^{n} (1 - Q_j) = \exp \left( \sum_{j=1}^{n} \ln(1 - Q_j) \right).$$

If the $\Delta t_j$ - time of $j$-st run, the failure rate is then

$$\lambda(t_j) = \frac{- \ln(1 - Q_j)}{\Delta t_j},$$

$$R(n) = \exp \left( \sum_{j=1}^{n} \lambda(t_j) \Delta t_j \right),$$

$$t_j = \sum_{i=1}^{j} t_i. \quad (29)$$

Practical use of formula (29) is complicated as a result of a plurality of inputs and a large number of hardly estimated model parameters. In practice, software reliability is assessed according the results of test trials which cover a relatively small region of initial data area.

For a simplified estimation the formula is proposed

$$R(N) = \frac{1}{N} \sum_{i=1}^{N} E_i(n_i) W_i, \quad \sum_{i=1}^{N} W_i = N,$$

where $N$ - number of runs, $n_i$ - number of errors at $i$-st run, $E_i$ - indicator of absence of errors at $i$-st run.

To reduce the problem dimension, the multitudes of input sets are split into disjointed subsets $G_j$, to each of which corresponds a certain path $L_j$, $j = 1, \ldots, n$. If $L_j$ has errors, then at the test performance along the subset $G_j$ will emerge a refusal. Subsequently the probability of correct performing of single test is

$$R(1) = \prod_{j=1}^{n} p_j \varepsilon_j, \quad p_j = \sum_{\varepsilon_{ij} \leq 1} p_{ij} \varepsilon_j < 1.$$
BIBLIOGRAFIA


