FIRST ORDER LOGIC FOR PROGRAM CODE FUNCTIONAL REQUIREMENTS DESCRIPTION

Kozachok A.V.[1], Bochkov M.V. [2], Lai M.T.[3], Kochetkov E.V.[4]

Abstract. *Currently the problem of information security during designing and exploiting the objects of critical information infrastructure is paid special attention to. One of the most common approaches to providing information security, processed on the objects, is creating isolated programming environment. The environment security is determined by its invariability. However, the evolutional development of data processing systems gives rise to the necessity of implementing the new components and software in this environment under condition that security requirements are satisfied. The most important requirement consists in trust in the new programming code. The given paper is devoted to developing formal logical language of description of functional requirements for programming code, allowing to make further demands at the stage of statical analysis and to control their implementation in dynamics.*

Keywords: *formal logical language, modeling, process, malware, model checking, security automata.*

DOI: 10.21681/2311-3456-2017-3-2-7

Introduction

During the last few years in researches devoted to information security special attention is paid to the problem of providing data security of the objects of critical information infrastructure (CII). The fact is proved by the developed and submitted for consideration the federal law «Security of critical information infrastructure of the Russian Federation», published in December 6, 2016. Special emphasis is placed on providing security of CII object against computer attacks and malware impact [1]. In this article the matter at issue is protecting CII objects against malware impact.

Very often the possibility of implementing these threats is determined by the availability of CII objects access to the Internet. Current systems and means of providing information security do not provide secure protection. For example, the study carried out by AV-Comparatives Company showed that the devices used in modern antivirus facilities do not allow to achieve the level of 0,974 heuristic detection («Avast Internet Security»), 468 harmful samples of software being undetected [2].

One of the possible approaches to providing security against malware impact is applying isolated programming environment that is reliable and secure on the assumption of its invariability. However, the evolutional development of information gathering and processing systems as well as availability of CII objects access to Internet gives rise to the need for implementing new components and software that may affect its integrity and security, the most essential things being the problem of reliability of the new content and programming code.

One of the possible versions of constructing secure environment with the capability of trust in the incoming content is application of a system of secure implementation of programming code [3]. The proposed system is enlarged composition of two currently developed approaches to detecting malware impact, namely: using methods of formal model-checking [4-7] and applying security automata for monitoring real-time properties of the studied program [8–12].

At the heart of the system secure programming code performance there is an assumption that if the programming code security with a priori known functional requirements is not proved, its application is forbidden. The given research is devoted to developing formal language of description of functional requirements for programming code for its using in the developed systems of secure programming code performance

1. Overview of the studies in the field of malware detection based on model-checking

The idea of using the method of formal model checking while solving the problems of destructive malware impact consists in constructing formal (mathematical) malware model simulating its possible behavior in operational system. Competent

1 Aleksandr Kozachok, Ph.D., The Academy of the Federal Guard Service of the Russian Federation, Orel, Russia. E-mail:alex.totrin@gmail.com

2 Maxim Bochkov, Dr.Sc., Professor, Business Risk Educational Center, Saint-Petersburg, Russia. E-mail: mvboch@yandex.ru

3 Tuan Lai Minh, Ph.D., The Academy of Cryptographic Techniques, Hanoi, Vietnam, E-mail: lm.tuan.gov.vn@gmail.com

4 Evgeniy Kochetkov, The Academy of the Federal Guard Service of the Russian Federation, Orel, Russia. E-mail: mr.Koch91@mail.ru

program behavior is assigned as a specification. On the basis of this program and the model of the file performed using the method «Model checking» the decision concerning the possibility of applying the analyzed program.

For the first time this method was used to solve the problem of detecting malicious code by Kinder J. in 2005 [4]. A team of authors proposed to analyze behaviour of programs and on the basis of «Model checking» to decide about its similarity to malware behaviour. The proposed approach consists in assigning specifications for each class of destructive programs by means of formula of temporal logic. Representatives of each class had similar behaviour but differed in their binary representation, that's why they could not be detected by signature method. Each studied binary file was automatically converted into model assigned in verifier language. On the basis of this model the verifier determined whether it corresponds to one of the sets of the assigned specifications corresponding to malware families. For shortening of specifications recording the authors introduced CTPL (Computation Tree Predicate Logic) that is expansion of the well-known CTL.

The benefit of the proposed approach is the possibility of detecting malware families. The shortcoming consists in the fact that it ignores the performance of the studied program with stack as well as the need for manual assignment of behaviour specification for each class of malicious code.

In 2012 Song F., Touili T. proposed to use «Model checking» method for detecting malware taking into account its behaviour and interaction with stack [5].

To describe the malicious code behaviour model the authors introduced Stack Computation Tree Predicate Logic that allows taking the work with stack into account. Application of the given approach made it possible to enhance the accuracy of detecting malware. As a result of developing the given approach the authors proposed SLTPL (Stack Linear Tree Predicate Logic) [6].

2. Description of the system for secure code execution

The distinctive feature of the developed system consists in retrieving information about the behaviour of the program being studied either on the storing stage or at the stage of performance (fig. 1). The received information is compared with the behaviour specification in accordance with assigned functional requirements. In case of their conformity the studied program is considered to be secure.

To carry the proposed approach into effect one must solve the following immediate tasks:

– analysis (retrieving and converting information from the program being studied into the form suitable for further processing – specifications);

– synthesis (constructing model of secure programming code performance in accordance with assigned functional requirements);

– verification (algorithm, receiving specification of the analyzed program for the access and taking decision concerning its conformity with model of secure programming code performance;

– monitoring performance (implementation of program performance monitor allowing to intercept all the signals of the process interaction with the operating system, to track the conformity of the program condition of the assigned configuration and to close the special purpose program in case of necessity).

At the block 1 input the studied file being performed is put. The security of it must be checked. At the given block one checks whether there are self-identification constructions in the programming code (including packers) and mechanisms to protect code against analysis. Meeting these requirements is necessary for further investigation of the file being operated with. In case of their presence the file is considered to be dangerous and its further checking is stopped. Then one converts the file being operated

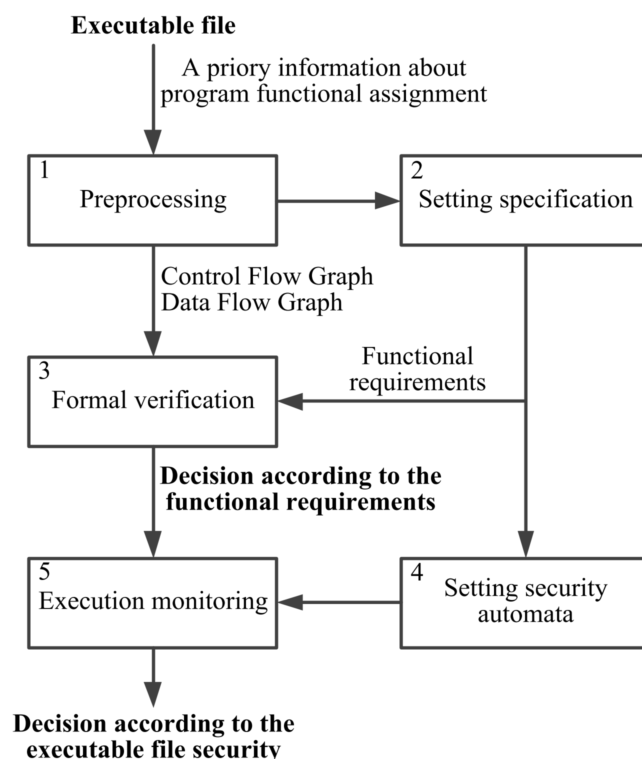

Fig. 1. *Model of secure programming code performance system*

with from binary representation into the set of assembly language instructions and data. On their basis one constructs Control Flow Graph and Data Flow Graph. At this stage one collects and systematizes a priori data concerning functional assignment of the program that are introduced into input of the block 2.

In the «Specification Assignments» Block on the basis of a priori data concerning functional assignment of the program, the list of constraints of its functional capabilities is formed, the realization of which is necessary for secure program application. The list includes functional requirements which implementation can be provided in the framework of the operation of the secure programming code performance system. They can be divided into groups according to the category of the studied program.

The output of the block is formalized functional constraints for program operation in the form of formula of temporal logic and configuration of security automata.

In block 3 the process of formal verification of the model of the performed file is executed, that is built on the basis of control flow graph and data flow graph. The purpose is to check its conformity with functional requirements of the static stage of checking by means of the «Model checking» method.

Requirements concerning correctness of secure behaviour are described in the form of specification reflecting framework of the competent program behaviour. Because of mathematical verification, decision concerning concordance, that is, conformity of possible behaviour with the required one is correct. The model checking algorithms, as a rule, are based on exhaustive attainability of a multitude of module states [13].

Thus, each state is checked if it satisfies the assigned specifications requirements. In the simplest form model checking algorithms allow to answer the question concerning the attainability of the assigned states. In this case it is necessary to determine all the forbidden states the attainability of which is not secure, and to find out if there is such a sequence of their replacement that can result in a forbidden one. If such a sequence does exist, then a decision to forbid application of the studied program is made. It should be noted that exhaustive attainability of a multitude of states is guaranteed in view of finiteness of the number of the model states [14]. In case of the program model nonconformity with security specification the performed file is considered to be dangerous and its future application is stopped. The output of the given block is the decision about secure use of the given program in accordance with the results of verification at the static checking stage.

To monitor fulfillment of functional constraints during the program operation it is suggested to use a system similar to the intrusion prevention system at the computer level. The execution monitor is switched simultaneously with the performed program and intercepts all the systems calls made by it. From the very beginning the execution monitor loads the authorized behaviour model and sets the initial state. Every call of the system function is compared with the authorized behaviour model and the transition into the new state takes place, or in case of the absence of such a transition, instruction to finish the process is given. At the heart of execution monitor security automata is put [8] that is built as a rule on the basis of automata with stack. The automata input symbols are the multitude of events of process functioning. The automata configuration determines the multitude of allowed operations for each state.

In block 4, conversion of functional program requirements into configuration of terminal automata with stack takes place. In block 5, continuous monitoring of the program functioning in the framework of the assigned secure performance model is carried out. The output of the given block is decision about safe execution of the performed file.

3. Formal logical language for functional requirements description

In the sphere of operational system (OS) performance the fundamental concepts are process and resources. According to [15] process is a container for a set of resources used during performing a copy of a program. The main kinds of OS resources are the following elements [16]:

– processing time;
– main storage
– external memory;
– input-output devices.

At the heart of the construction of the proposed secure programming code performance system is the functional process description model in operation system.

The subjects are the processes performing the action with the objects. The objects are the OS resources and processes subjected to the actions of other subjects:

– «process» (p);
– «main storage» (m);
– «external memory» (e);
– «peripheral devices» (d);
– «network subsystem» (n).

To access the resources the process performs the appropriate OS function, that is, makes a request for

performing some actions. On the basis of inside resources distribution mechanisms as well as security policy, OS makes decision concerning the access of the performed file of the given processor to the requested resources.

In the process of operation, applications possess the entire access to its virtual address space for performing operations of reading and recording.

To input and output data outside the limits of its address space of the application program, it is necessary to stimulate the corresponding OS functions, if it has the appropriate privileges for performing such operations. These OS functions are the following: reading operation, recording operation, starting/completion operation, allocation of additional memory space, its emptying etc.

Review of investigations in the field of formal verifications shows that current approaches to description of specifications for solving the problems of detecting malware are not universal, since some of them are oriented to assembly language commands, the other part being oriented to API-functions.

Thus we propose the following formal logical language to assign functional requirements with the capability of one way transition to formula of temporal logic for further verification according to the models.

In accordance with the logic definition of the first order [17] it is necessary to assign the following subsets:

$$FormSpec = Func \cup Pred \cup Var \cup Log \cup Aux.$$

At that the set of functional symbols will include the following operations:

$$Func = \{create, open, delete, read, write\},$$

where *create* is operation of creating the object, *open* is operation of opening the object, *delete* is operation of deleting (completing) of the object, *read* is operation of reading in the object, *write* is operation of recording in the object.

Set of predicative symbols includes basic predicates of temporal CTL logic [18] and security check-up predicate:

$$Pred = \{IsSecure, AX, AF, AG, AU, AR, EX, EF, EG, EU, ER, EC\},$$

where *IsSecure* is the predicate of security check-up of the current state with regard to the route of performance on the whole, A is universal quantifier showing that the given property is fulfilled for all the routes, E is existential quantifier of existence showing that the given property is for a certain route, X is unary operator showing that the given property if fulfilled at the next state of the current route, G is unary operator showing that the given property is fulfilled at every state of the current route, F is unary operator showing that the given property is fulfilled at some state in future, U is binary operator showing that the first property is fulfilled for all states of the route previous to the state where the second property is fulfilled, R is binary operator showing that the second property is fulfilled for all the states following to the state where the first property is fulfilled, C is unary operator showing that the given property is fulfilled at the current state of the current route (additionally introduced by the authors).

Set of symbols of subject variables includes the following elements:

$$Var = \{p, m, n, e, d, cat\},$$

where p,m,n,e,d are the objects subjected to actions, cat is index of the object (subject) category (table 1).

Table 1.
Categories of objects and subjects.

definitione	description
Subject «Process» (p)	
1	system process
2	privileged process
3	user process
Object «main storage» (m)	
1	system process address space
2	address space of another process
3	own process address space
Object «external memory» (e)	
1	performed files
2	system catalogues and system configuration
3	files and catalogues of other users
4	system libraries
5	own files and catalogues
Object «Peripheral devices» (d)	
1	output devices
2	input devices
Object «Network subsystem» (n)	
1	node services global networks
2	node services of local networks
3	local network services

Set of logical symbol include the following elements:

$$Log = \{\neg, \wedge, \vee, \rightarrow, \exists, \forall\},$$

where $\neg$ is symbol of logical negation, $\wedge$ is conjunction symbol, $\vee$ is disjunction symbol, $\rightarrow$ – implication symbol, $\exists$ – existential quantifier, $\forall$ – universal quantifier.

Set of subsidiary symbols include the following elements:

$$Aux = \{, ()\}.$$

4. Basis of functional requirements to provide secure program code execution

On the basis of the proposed *FormSpec* formal logical language the basic rules (formulas) of secure programming code performance were formulated for each functional symbol. Symbol * denotes object (subject) of any category from all possible members of the given class.

For the operation of creating the object:

$\neg EF\ create\ (p,*,p,*)$ – ban on creating child processes;

$EF\ create\ (p,*,m,3)$ – memory allotment only in its own process address space;

$EF\ create\ (p,*,e,5)$ – authorization for creating new files (catalogues) only in the catalogue of the current process;

$\neg EF\ create\ (p,*,n,*)$ – ban on creating network connections;

$\neg EF\ create\ (p,*,d,*)$ – ban on creating devices (drivers).

For operations of an open object:

$\neg EF\ open\ (p,*,p,*)$ – ban on opening processes;

$EF\ open\ (p,*,e,4) \vee EF\ open\ (p,*,e,5)$ – authorization for opening system libraries and files contained in current process catalogue;

$\neg EF\ open\ (p,*,d,*)$ – ban on opening devices.

For object delete operations (completion):

$EF\ delete\ (pi,*,pi,*)$ – the process may terminate its operation;

$EC\ open\ (p,*,ej,5) \wedge EF\ delete(p,*,ej,5)$ – the process may delete files created by it.

For reading from the object operation:

$\neg EF\ read(p,*,p,*)$ – ban on getting information about processes;

$EF\ read(p,*,m,3)$ – authorization for reading the address space of one's own process;

$EC\ open(p,*,ej,4) \wedge EF\ read(p,*,ej,4)$ – authorization for reading from system libraries;

$(EC\ open(p,*,ej,5) \vee EC\ create(p,*,ej,5)) \wedge EF\ read(p,*,ej,5)$ – authorization for reading files opened (created) by the process;

$\neg EF\ read(p,*,n,*)$ – ban on the work with the network

$\neg EF\ read(p,*,d,*)$ – ban on the work with devices.

For operation of recording in the object:

$EF\ write(p,*,m,3)$ – authorization for recording in the address space of one's own process;

$(EC\ open(p,*,ej,5) \vee EC\ create(p,*,ej,5)) \wedge EF\ write(p,*,ej,5)$ – authorization for reading files opened (created) by the process;

$\neg EF\ write(p,*,n,*)$ – ban on the work with the network;

$\neg EF\ write(p,*,d,*)$ – ban on the work with devices.

It should be noted that the presented basis may be considered as axiomatic one, since its execution provides security performance of programming code (IsSecure predicate performance) in perspective of protection against malware code. Constraints introduced by it concerning interaction with network and file subsystems may be overcome owing to introduction of limitations for subsequence of performed actions and isolations of possible informational contours.

5. Results and Discussion

One of the versions of the practical application of the proposed formal logical description of functional requirements to programming code is formalization of threats from «Bank of data of information threat» [19].

«Threat to changing system and global variables» by intruder may be realized at the expense of using malware that may cause to mediate destructive impact on certain programs and system as a whole. To neutralize the threat it is necessary to assign the following rule: «To disallow the 3 category processes to carry out changes of system and global variables». It is expressed in the following way:

$$\neg EF\ (create\ (p,3,e,2) \vee write\ (p,3,e,2)) \quad (1)$$

Formally the expression means (1) that 3 category processes cannot carry out creation or modification of system catalogues and configuration files. This rule can also prevent «threat of unauthorized editing register».

The essence of «threat of unauthorized copying protected information» consists in malefactor's getting the copy of protected information of another user and its further withdrawal outside the system.

To constrain the sequence of such actions it is necessary to accept the following rule:

$$\neg EF(EC\ read\ (p,*,e,3) \wedge (EF\ create\ (p,*,e,5) \vee \vee EF\ write\ (p,*,e,5) \vee EF\ write\ (p,*,d,1) \vee \vee EF\ write\ (p,*,n,1)) \quad (2)$$

The expression (2) means that process of any category is disallowed to read some information in other user's file and then to record it in files of one's own catalogue or to send it to output devices or network.

For «the threat of intercepting information that is input or output on peripheral devices» one can assign the rule limiting direct interaction of the 3 category processes and input devices:

$$\neg EF\ read(p,3,d,2) \tag{3}$$

The expression (3) inhibits direct access to reading information from input devices in a roundabout way from existing mechanisms in operating system.

A set of the given examples proves opulence and variety of possibilities to describe current threats in proposed formal logical language. Taking them into consideration while working with the system of secure code performance, will allow excluding the possibility of treat realization.

Conclusion

The proposed formal logical language of description of functional requirements enabling to describe any process behaviour without concrete definition of operations or elementary actions (at a high abstraction level) and in generalized mathematical formula to express subject-object relations of process and resources of different OS categories forms the basis of designing the system of secure programming code performance that will allow to trust the new programming code and not to affect integrity of isolated programming environment.

The orientation of further research is constructing the whole set of rules of secure programming code performance using the introduced formal logical language enabling to eliminate constrains set by axiomatic basis.

References

1. Proekt federal'nogo zakona ot 06.12.2016 № 9198-P10 «O bezopasnosti kriticheskoj informacionnoj infrastruktury Rossijskoj Federacii». URL: http://asozd2.duma.gov.ru/main.nsf/ (SpravkaNew)/OpenAgent&RN=47571-7&02.
2. Bekbosynova A.A. Testirovanie i analiz jeffektivnosti i proizvoditel'nosti antivirusov // Teorija i praktika sovremennoj nauki [The theory and practice of modern science], 2015. № 5 (5). Pp. 53-56.
3. Kozachok A.V., Kochetkov E.V. Obosnovanie vozmozhnosti primenenija verifikacii programm dlja obnaruzhenija vredonosnogo koda // Voprosy kiberbezopasnosti [Cybersecurity issues]. 2016. № 3 (16). Pp. 25-32.
4. Kinder J. et al. Detecting malicious code by model checking // International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. – Springer Berlin Heidelberg. 2005. Pp. 174-187.
5. Song F., Touili T. Efficient malware detection using model-checking // International Symposium on Formal Methods. – Springer Berlin Heidelberg. 2012. Pp. 418-433.
6. Song F., Touili T. PoMMaDe: pushdown model-checking for malware detection //Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. – ACM. 2013. Pp. 607-610.
7. Jasiul B., Szpyrka M., Śliwa J. Formal Specification of Malware Models in the Form of Colored Petri Nets // Computer Science and its Applications. – Springer Berlin Heidelberg. 2015.Pp. 475-482.
8. Schneider F.B. Enforceable security policies // ACM Transactions on Information and System Security (TISSEC). 2000. Vol. 3 .no. 1. Pp. 30-50.
9. Feng H.H. et al. Formalizing sensitivity in static analysis for intrusion detection // Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on. – IEEE. 2004. Pp. 194-208.
10. Basin D. et al. Enforceable security policies revisited // ACM Transactions on Information and System Security (TISSEC). 2013. vol. 16. no. 1. Pp. 3–8.
11. Feng H.H. et al. Anomaly detection using call stack information //Security and Privacy, 2003. Proceedings. 2003 Symposium on. – IEEE. 2003. Pp. 62-75.
12. Basin D., Klaedtke F., Zălinescu E. Algorithms for monitoring real-time properties // International Conference on Runtime Verification. – Springer Berlin Heidelberg. 2011. Pp. 260-275.
13. Klark Je., Gramberg O., Peled D. Verifikacija modelej programm: Model Checking. M.: MCNMO., 2002. 416 p.
14. Vel'der S.Je., Lukin M.A., Shalyto A.A., Jaminov B.R. Verifikacija avtomatnyh programm SPb. Nauka., 2011, 244 p.
15. Russinovich M., Solomon D. Vnutrennee ustrojstvo Microsoft Windows. 6-e izd. SPb.: Piter., 2013, 800 p.
16. Gordeev A.V. Operacionnye sistemy. Izdatel'skij dom «Piter»., 2009, 412 p.
17. Korotkov M.A., Stepanov E.O Osnovy formal'nyh logicheskih jazykov. SPb: SPb GITMO (TU), 2003, 84 p.
18. Hafer T., Thomas W. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree // International Colloquium on Automata, Languages and Programming. – Springer Berlin Heidelberg, 1987. – Pp. 269–279.
19. FSTJeK «Bank dannyh ugroz bezopasnosti informacii» URL: http://bdu.fstec.ru.