ISSN 0361-7688, Programming and Computer Software, 2017, Vol. 43, No. 5, pp. 314–319. © Pleiades Publishing, Ltd., 2017.
Original Russian Text © A.V. Kozachok, 2017, published in Trudy Instituta Sistemnogo Programmirovaniya, 2017, Vol. 1, No. 2, pp. 3–4.

Formal Logical Language to Set Requirements for Secure Code Execution

A. V. Kozachok

Academy of Federal Security Guard Service of the Russian Federation,
ul. Priborostroitel'naya 35, Orel, 302034 Russia
e-mail: a.kozachok@academ.msk.rsnet.ru

Received May 11, 2017

Abstract—Presently, a special attention is paid to the problem of information security when designing and using objects of critical information infrastructure. One of the most common approaches used to secure the information processed on these objects is the creation of an isolated program environment (sandbox). The security of the environment is determined by its invariability. However, the evolutionary development of data processing systems makes it necessary to implement new components and software in this environment on the condition that the security requirements are met. In this case, the most important requirement is trust in a new program code. This paper is devoted to developing a formal logical language to describe functional requirements for program code that allows one to impose further constraints at the stage of static analysis, as well as to control their fulfillment in dynamics.

DOI: 10.1134/S036176881705005X

1. INTRODUCTION

During the last few years, the researches devoted to information security problems pay a special attention to protection of objects of critical information infrastructure (CII). In this connection, a federal law "On the Security of the Critical Information Infrastructure of the Russian Federation" has been developed and introduced for consideration. This document is focused on protection of CII objects against computer hacking and malware [1]. In turn, this paper addresses the problem of protecting CII objects against malware.

In many cases, the threats mentioned above can materialize because CII objects have access to the Internet. Moreover, the existing information security tools and systems often fail to provide guaranteed protection. For example, a research carried out by AV-Comparatives showed that the mechanisms used in modern antivirus software make it possible to reach the heuristic detection level of 0.974 (Avast Internet Security) but fail to detect 468 malware samples [2].

One of possible approaches to protect against malicious software consists in using an isolated program environment (sandbox), which is regarded as trusted and safe as long as it preserves its invariability. However, the evolutionary development of data collection and processing systems, as well as the access of CII objects to the Internet, forces one to run new components and software in this environment, which can jeopardize its integrity and security. In this case, the most important requirement is trust in a new content and program code.

A secure environment providing confidence in the incoming content can be constructed based on secure code execution [3]. The system proposed in this paper is an extended combination of two intensively developing approaches to malware detection: model checking [4–7] and security automata for real-time monitoring of program execution [8–12].

Secure code execution is based on the assumption that, if, under a priori known functional requirements, the security of program code is not confirmed, then its execution is prohibited. This research is aimed at developing a formal language to describe functional requirements for program code in order to ensure its safe run in the developed secure execution system.

2. BRIEF SURVEY OF THE RESEARCHES IN THE FIELD OF MODEL CHECKING FOR MALICIOUS CODE DETECTION

When solving the malware detection problem, model checking is used to construct a formal (mathematical) model of a malicious program, which reflects (models) its possible behavior in the operating system. In this case, the admissible behavior of the program is described by a specification. Based on this specification and the model of the executable file, using the model checking method, a decision is made whether this program is safe to run.

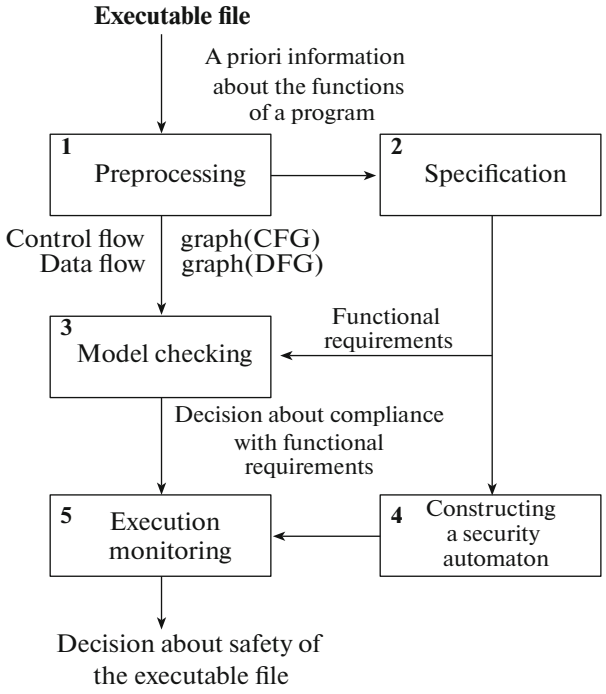

Fig. 1. Model of the secure code execution system.

Kinder was the first to use this method for malicious code detection [4]. A group of researchers proposed to analyze the behavior of programs by means of model checking so as to decide whether their behavior is malicious. The approach proposed by these researchers consisted in setting specifications, by using temporal logic formulas, for each class of malicious programs that exhibited a similar behavior but significantly differed in their binary representations and, for this reason, were misdetected by the signature method. Each binary file of a program under analysis was automatically translated into a model written in the verifier's language. Based on this model, the verifier decided whether this program fell under any of the given specifications describing families of malicious software. To shorten the notation of the specifications, the computation tree predicate logic (CTPL) was developed as an extension of the well-known CTL. The advantage of this approach is that it can detect families of malicious programs. However, this approach does not take into account the way a program works with the stack and requires setting specifications by hand for all classes of malicious code.

In 2012, Song and Touili proposed a model checking method to detect malicious programs taking into account their behavior and interaction with the stack [5]. To describe the behavioral model of malicious code, they introduced the stack computation tree predicate logic (SCTPL), which took into account operations on the stack. This approach significantly improved the accuracy of malware detection. The further development of this approach led to the stack linear tree predicate logic (SLTPL), which was proposed in [6].

3. DESCRIPTION OF THE SECURE CODE EXECUTION SYSTEM

A distinctive feature of the developed system is extraction of the information about program behavior both at the stage of storage and at the stage of execution (see Fig. 1). This information is compared with a behavior specification taking into account given functional requirements. In the case of match, the program is regarded as safe.

To implement the proposed approach, the following subproblems need to be solved:

• analysis (extraction of the corresponding information from a given program and conversion thereof into a form suitable for further processing (specification));

• synthesis (construction of a secure code execution model in accordance with given functional requirements);

• verification (running an algorithm that, based on the inputted specification of the given program, checks whether the program satisfies the secure code execution model);

• execution control (implementing a program execution monitor to intercept all interactions between the process and the operating system, check the adequacy of a program state to a given configuration, and, if required, terminate the target program).

An executable file to be analyzed is inputted to block 1. In this block, the program code is checked for the presence of self-modifying constructions (including wrappers) and mechanisms that protect the code against analysis. Compliance with these requirements is mandatory for the analysis to continue; if the file contains such constructions, then it is regarded as unsafe and the analysis terminates. Next, the executable file is converted from a binary representation into a sequence of assembler commands and data used to construct a control flow graph (CFG) and data flow graph (DFG). This block also collects and classifies a priori information about the functions of the program; this information is inputted to block 2.

In the "Specification" block, based on the information about the functions of the program, a set of constraints on its functional capabilities is generated; these constraints should be met for safe execution of the program. This set includes functional requirements that can be checked in the secure code execution system. These requirements are grouped based on the category of a program under analysis. This block yields formalized functional constraints on the program in the form of temporal logic formulas and security automaton's configuration.

In block 3, the model of the executable file, which is constructed based on the CFG and DFG, under-

goes formal verification, i.e., it is checked for compliance with the functional requirements of the static analysis stage. Here, the behavioral requirements are described by a specification that limits the admissible behavior of the program. Since the system uses mathematical verification, the decision about the correspondence between the possible behavior and the required one is regarded as correct. Model checking algorithms are generally based on exhaustive reachability of the whole set of model states [13].

Thus, all states are checked for compliance with the specification. In the simplest form, model checking algorithms make it possible to answer the question about reachability of the states. In this case, it is required to find all forbidden (unsafe) states and to determine whether there is a sequence of state transitions that leads to a forbidden state. If such a sequence exists, then the use of the program is prohibited. It should be noted that the exhaustive reachability of the set of states is guaranteed due to the finiteness of the number of model states [14]. If the program model does not comply with the security specification, then the executable file is regarded as unsafe and the analysis terminates. This block yields a decision about safety of a given program, which is based on the results of verification at the static analysis stage.

To control the fulfillment of the functional constraints during program run, we use a system similar to that for intrusion prevention at the computer level. The execution monitor runs in parallel with an executable program and intercepts all its system calls. First, the execution monitor uploads the admissible behavior model and sets the initial state. All calls of system functions are checked for compliance with this model; then, a transition to a new state occurs; if no transition followed, then the process terminates. The execution monitor is based on a security automaton [8], which, in turn, is generally constructed based on pushdown automata. The input symbols of the automaton are elements of the set of process events, while the configuration of the automaton determines the set of operations allowed in each state.

Block 4 translates the functional requirements for the program into the configuration of the pushdown automaton. Block 5 carries out constant monitoring of the program's behavior in the framework of the given secure execution model. This block yields a decision whether it is safe to execute the program file.

4. FORMAL LOGICAL LANGUAGE FOR DESCRIPTION OF FUNCTIONAL REQUIREMENTS

Process and resource are the basic concepts characterizing the work of an operating system (OS). According to [15], a process is a container for a set of resources used to execute an exemplar of a program. The main types of OS resources are [16]

- CPU time;
- random access memory;
- external memory;
- input-output devices.

The proposed secure code execution system is based on the model that describes the behavior of a process in the OS. In this case, subjects are processes that act on objects. In turn, objects are OS resources and processes the subjects act on:

- "Process" (p);
- "Random access memory" (m);
- "External memory" (e);
- "Peripheral devices" (d);
- "Network subsystem" (n).

To access a resource, a process calls the corresponding OS function, i.e., requests for executing certain actions. The OS, using the resource allocation mechanism, based on a certain security policy, makes a decision on granting the access to the resource requested.

When running, applied programs have full access to their virtual address space for read and write operations. To input or output data outside a private address space, an applied program needs to call the corresponding OS functions provided that it has the corresponding privileges to carry out these operations. These OS functions are read and write operations, initiation and termination of a process, memory allocation and deallocation, etc.

The analysis of researches in the filed of formal verification shows that the existing approaches to description of specifications for malware detection are not universal because some of them are oriented to assembler commands, while others are oriented to API functions.

In this paper, we propose a formal logical language for setting functional requirements that allows unequivocal transition to temporal logic formulas for further verification based on models.

According to the definition of the first-order logic [17], the following subsets need to be defined:

$$FormSpec = Func \bigcup Pred \bigcup Var \times \bigcup Log \bigcup Aux.$$

In this case, a set of functional symbols includes the following operations:

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

where *create* is the operation of creating an object, *open* is the operation of opening an object, *delete* is the operation of deleting (terminating) an object, *read* is the operation of reading from an object, and *write* is the operation of writing into an object.

A set of predicate symbols includes the basic CTL* predicates [18] and a security check predicate:

Table 1. Categories of objects and subjects

Category	Description
Subject "Process" (p)	
1	system process
2	privileged process
3	user process
Object "Random access memory" (m)	
1	address space of a system process
2	address space of other process
3	private address space of a process
Object "External memory" (e)	
1	executable files
2	system catalogs and system configuration
3	files and catalogs of other users
4	system libraries
5	private files and catalogs
Object "Peripheral devices" (d)	
1	output devices
2	input devices
Object "Network subsystem" (n)	
1	services of global network nodes
2	services of local network nodes
3	local network services

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

where *IsSecure* is the predicate for checking the security of the current state with respect to the entire execution route, A is the generality quantifier indicating that a given property holds for all paths, E is the existential quantifier indicating that a given property holds for a certain path, X is a unary operator indicating that a given property holds in the next state of the current path, G is a unary operator indicating that a given property holds in all states of the current path, F is a unary operator indicating that a given property will hold in a certain future state, U is a unary operator indicating that the first property holds in all states of a path that precede the state in which the second property holds, R is a unary operator indicating that the second property holds in all states preceding the state in which the first property holds, and C is a unary operator indicating that a given property holds in the current state of the current path (this operator is additionally introduced by the authors).

A set of symbols for subject variables is

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

where p, m, n, e, d are objects exposed to certain actions and *cat* is the category number of an object (subject) (see table 1).

A set of logical symbols is

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

where $\neg$ represents logical negation, $\wedge$ represents conjunction, $\vee$ represents disjunction, $\rightarrow$ represents implication, $\exists$ is the existential quantifier, and $\forall$ is the generality quantifier.

A set of auxiliary symbols is

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

5. BASIS OF FUNCTIONAL REQUIREMENTS FOR SECURE CODE EXECUTION

Using the proposed formal logical language *FormSpec*, we formulate basic rules (formulas) of secure code execution for all functional symbols. The symbol * denotes an object (subject) of any category available for a given class.

For the operation of creating an object,

- $\neg EF\ create(p,*,p,*)$, creation of child processes is prohibited;
- $EF\ create(p,*,m,3)$, memory is allocated only within the private address space of a process;
- $EF\ create(p,*,e,5)$, new files (catalogs) can be created only in the catalog of the current process;
- $\neg EF\ create(p,*,n,*)$, creation of network connections is prohibited;
- $\neg EF\ create(p,*,d,*)$, creation of devices (drivers) is prohibited.

For the operation of opening an object,

- $EF\ open(p,*,p,*)$: opening of processes is prohibited;
- $EF\ open(p,*,e,4) \vee EF\ open(p,*,e,5)$: system libraries and files contained in the current catalog of a process are allowed to be opened;
- $\neg EF\ open(p,*,d,*)$: opening of devices is prohibited.

For the operation of deleting (terminating) an object,

- $EF\ delete(p_i,*,p_i,*)$, a process is allowed to terminate itself;
- $EF\ open(p,*,e_j,5) \wedge EF\ delete(p,*,e_j,5)$, a process is allowed to delete files it created.

For the operation of reading from an object,

- $\neg EF\ read(p,*,p,*)$, reading of the information about the processes is prohibited;
- $EF\ read(p,*,m,3)$, reading from the address space of a private process is allowed;
- $EC\ open(p,*,e_j,4) \wedge EF\ read(p,*,e_j,4)$, reading from the system libraries is allowed;
- $(EC\ open(p,*,e_j,5) \vee EC\ create(p,*,e_j,5)) \wedge EF\ read(p,*,e_j,5)$, reading of the files opened (created) by a process is allowed;

• $\neg EF\,read(p,*,n,*)$, network operations are prohibited;

• $\neg EF\,read(p,*,d,*)$, operations with the devices are prohibited.

For the operation of writing into an object,

• $EF\,write(p,*,m,3)$, writing into the address space of a private process is allowed;

• $(EC\,open(p,*,e_j,5) \vee EC\,create(p,*,e_j,5)) \wedge EF\,write(p,*,e_j,5)$, reading of the files opened (created) by a process is allowed;

• $\neg EF\,write(p,*,n,*)$, network operations are prohibited;

• $\neg EF\,write(p,*,d,*)$, operations with the devices are prohibited.

It should be noted that this basis can be regarded as an axiomatic one because its fulfillment ensures secure code execution (execution of the predicate *IsSecure*) in terms of protection against malicious code. In addition, the constraints on interactions with the network and file subsystems can be overcome by imposing some constraints on the sequence of actions, as well as by isolating possible information contours.

6. RESULTS AND DISCUSSION

The proposed formal logical language for describing functional requirements to program code can be applied to formalization of threats from the Data Security Threats Database (Federal Service for Technical and Export Control of Russia) [19].

"The threat of modifying system and global variables" by the malefactor can have an indirect destructive effect on certain programs or the whole system. To neutralize this threat, we specify the following rule: "the processes of the third category are not allowed to modify the system and global variables;" this rule is written as follows:

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

Formally, expression (1) means that the processes of the third category cannot create or modify system catalogs and configuration files. This rule can also be used to deal with "the threat of unauthorized registry editing."

"The threat of unauthorized copying of protected data" implies that the malefactor copies the protected data belonging to another user and gets this copy out of the system. The rule for restricting the sequence of these actions is as follows:

$$\neg EF(ECread(p,*,e,3) \wedge (EFcreate(p,*,e,5) \vee EFwrite(p,*,e,5) \vee EFwrite(p,*,d,l) \vee EFwrite(p,*,n,l)). \qquad (2)$$

Expression (2) means that the process of any category is not allowed to, sequentially, read certain data from files of other users and, then, write these data into the files in its private catalog, or send these data to output devices or to the network.

For "the threat of intercepting the data inputted to or outputted from peripheral devices," the following rule is specified that restricts direct interaction between the processes of the third category and the input devices:

$$\neg EFread(p,3,d,2) \qquad (3)$$

Expression (3) prohibits direct data reading (bypassing the corresponding OS mechanisms) from the input devices.

These examples confirm the consistency and completeness of the threat descriptions made using the formal logical language proposed. Taking these descriptions into account in the secure code execution system allows one to prevent these threats from being materialized.

7. CONCLUSIONS

The proposed formal logical language for functional requirements allows one to describe the behavior of a process without concretization of operations or elementary actions (i.e., at a high level of abstraction), as well as to express (in a generalized mathematical form) subject–object relationships between processes and resources of different categories. This language has been used to develop a secure code execution system that allows one to confidently run new program code without violating the integrity of the isolated program environment.

The further researches will be focused on constructing a complete set of secure code execution rules by using the proposed formal logical language in order to remove the constraints imposed by the axiomatic basis.

REFERENCES

1. Draft Federal Law no. 47571-7 "On the Security of the Critical Information Infrastructure of the Russian Federation." http://asozd2.duma.gov.ru/main.nsf/(Spravka)?OpenAgent&RN=47571-7.
2. Kozachok, A.V., Kochetkov, E.V., and Tatarinov, A.M., Substantiation of the possibility to construct a heuristic mechanism for malware recognition based on static analysis of executable files, *Vestn. Komp'yut. Inf. Tekhnol.,* 2017, no. 3, pp. 50–56.
3. Kozachok, A.V. and Kochetkov, E.V., Substantiation of the possibility to apply program verification to malicious code detection, *Vopr. Kiberbezopasnosti,* 2016, vol. **16**, no. 3, pp. 25–32.
4. Kinder, J., et al., Detecting malicious code by model checking, *Proc. Int. Conf. Detection of Intrusions and Malware and Vulnerability Assessment,* Berlin: Springer, 2005, pp. 174–187.

5. Song, F. and Touili, T., Efficient malware detection using model-checking, *Proc. Int. Symp. Formal Methods,* Berlin: Springer, 2012, pp. 418–433.
6. Song, F. and Touili, T., PoMMaDe: Pushdown model-checking for malware detection, *Proc. 9th Joint Meeting on Foundations of Software Engineering,* ACM, 2013, pp. 607–610.
7. Jasiul, B., Szpyrka, M., and Sliwa, J., Formal specification of malware models in the form of colored Petri nets, in *Computer Science and its Applications,* Springer, 2015, pp. 475–482.
8. Schneider, F.B., Enforceable security policies, *ACM Trans. Inf. Syst. Secur. (TISSEC),* 2000, vol. **3**, no. 1, pp. 30–50.
9. Feng, H.H., et al., Formalizing sensitivity in static analysis for intrusion detection, *Proc. IEEE Symp. Security and Privacy,* 2004, pp. 194–208.
10. Basin, D., et al., Enforceable security policies revisited, *ACM Trans. Inf. Syst. Secur. (TISSEC),* 2013, vol. **16**, no. 1, pp. 3–8.
11. Feng, H.H., et al., Anomaly detection using call stack information, *Proc. IEEE Symp. Security and Privacy,* 2003, pp. 62–75.
12. Basin, D., Klaedtke, F., and Zalinescu, E., Algorithms for monitoring real-time properties, *Proc. Int. Conf. Runtime Verification,* Berlin: Springer, 2011, pp. 260–275.
13. Clarke, E.M., Grumberg, O., and Peled, D., *Model Checking,* MIT Press, 1999.
14. Vel'der, S.E., Lukin, M.A., Shalyto, A.A., and Yaminov, B.R., *Verifikatsiya avtomatnykh programm* (Verification of Automata-Based Programs), St. Petersburg: Nauka, 2011.
15. Russinovich, M.E. and Solomon, D.A., *Windows Internals,* Microsoft Press, 2012, 6th ed.
16. Gordeev, A.V., *Operatsionnye sistemy* (Operating systems), St. Petersburg: Piter, 2009.
17. Korotkov, M.A. and Stepanov, E.O., *Osnovy formal'nykh logicheskikh yazykov* (Basics of Formal Logic Languages), St. Petersburg: State Inst. Precision Mech. Opt. (Tech. Univ.), 2003.
18. Hafer, T. and Thomas, W., Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree, *Proc. Int. Colloq. Automata, Languages and Programming,* Berlin: Springer, 1987, pp. 269–279.
19. Databank of Information Security Threats. http://www.bdu.fstec.ru.

Translated by Yu. Kornienko