The Technique of Dynamic Binary Analysis and Its Application in the Information Security Sphere

Maxim Shudrak[#1], Vyacheslav Zolotarev[#2].
[#] *Information Security Department, Siberian State Aerospace University*
Krasnoyarsk, Russian Federation
[1] mxmssh@gmail.com
[2] amida@land.ru

Abstract— **the article describes static and dynamic analysis techniques and its applicability in information security problems such as software protection against unauthorized research. The basic idea of the article is that techniques of dynamic and static analysis must be used in combination with each other to increase the effectiveness of binary code analysis. In the article authors make contributions in binary code decompilation and dynamic execution analysis techniques. The results are applied to the problem of software protection against unauthorized reverse engineering. Authors used analysis of the basic program control flow algorithm to obfuscate the program and protect it against research.**

Keywords – dynamic, static analysis, control flow, software protection, obfuscation.

I. INTRODUCTION

Today, software companies faces serious risks associated with software security because the process of software development always contains some errors and mistakes. In turn, these errors can create favorable conditions for emergence serious vulnerabilities in software. Vulnerabilities exploitation affects not only the system security that uses vulnerable product but also the company competitive advantages engaged in the software development. It should be noted that the process of vulnerabilities research is greatly complicated, because in this process there is number of fundamental technological problems. Also, in practice any attack technique on software security is based on preliminary disassembly and analysis of binary code protection mechanism. The purpose of this research is to restore protection algorithm, to identify its weaknesses and undocumented features for their future modification and (or) to automate the process of attack [1]. To resolve these problems techniques of static and dynamic binary analysis are widely used.

There are several types of analysis which includes «white and black box» techniques. The «white box» technique or static analysis operates with source code using manual or semi-automated analysis. This technique can indicate a precise line of code where a flaw or error occurs. It allows detecting simple errors or vulnerabilities in software code. But «white box» technique requires source code access and doesn't provide dependent code coverage (shared libraries and functions without source code etc.). The technique also limited by the ability of errors potential functionality identification and also generates a high level of false positive errors.

The «black box» technique or dynamic analysis can perform detection during software execution which allows exploitable errors and vulnerabilities detection. «Black box» technique uses potentially erroneous data dynamic generation and exception monitoring. However dynamic analysis can't identify a precise line of code where a flaw occurs and also doesn't provide full code coverage for test data. The second serious problem is performance, because software testing analysis requires debugging which hundred or thousand number of times reduces the speed of the application (performance degrades because the time spend on context switching between application and debugger) and also testing process. In addition, the process of dynamic analysis includes protocol description for potential erroneous data generation which often requires a considerable time especially with complexity of various data protocols for latest applications.

In articles [2][3][4] authors demonstrated techniques of dynamic and static binary analysis for vulnerabilities detection in binary code. Dynamic analysis is based on the execution of the program on the CPU. In turn, the static analysis is based on the analysis PE-module technique in memory.

In [2][3] dynamic analysis technique is used to restore software execution flow. In [4] authors used static binary analysis technique to detect vulnerabilities patterns in PE (portable executable) modules. In [5][6][7] authors demonstrated that binary execution flow obfuscation was effective for software security against unauthorized research. In articles [5][6], authors use dynamic binary execution techniques for software obfuscation. In turn, in [7] author uses a static analyzer for analysis of execution and data flow algorithms.

The paper introduces special techniques of dynamic and static analysis and also optimization of existing ones. We used a combination of static and dynamic analysis techniques to maximize the effectiveness of protection and vulnerabilities detection. To implement the static analyzer, we propose to use our previous works [1][8] in decompilation and intermediate language (IL) implementation .

Thus, in this article we make the following contributions:

1. *IL analysis technique***.** We introduce a new scheme for static binary analysis — that allows us to search vulnerabilities in the intermediate language (IL).

978-1-4673-2232-4/13/$31.00 ©2013 IEEE

2. *Dynamic execution analysis technique*. The paper provides optimization algorithms for dynamic binary code execution.

3. *Hybrid analysis technique***.** Technique combines static and dynamic algorithms for software protection against unauthorized research.

II. Dynamic and Static Analysis Techniques

A. Static Binary Analysis Technique

To implement the technique of static analysis, binary code must be interpreted into the intermediate language. In our previous work, we have formulated and described this IL. [8] To translate x86 mnemonics in the IL we used a basis which consisted of 16 basic operations. This notation allows us to describe the code in resource - oriented form (Figure 1). Registry resource is denoted by bracers which entered number of resources. Memory resource is indicated by brackets which entered resource address. Number of such resources is taken out of brackets. The result is separated by «=». The operations and constants are written «as is».

This form of instruction writing presents instructions in the form of mathematical formulas. Next, we proposed to use binary trees for describing the syntactic transformation of the resources to establish communication between operations. Tree nodes contain the operations from the basis and the leaves contain variables (Figure 2). Thus, to obtain the final expression it is necessary to perform a recursive traversal: {left leaf, node, right leaf}.

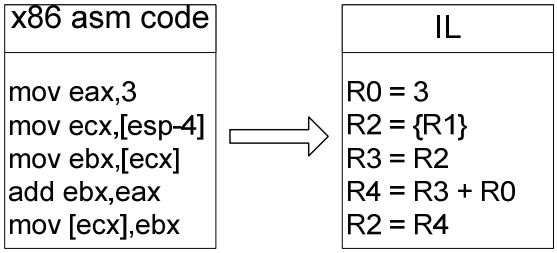

Fig. 1. Binary code translation

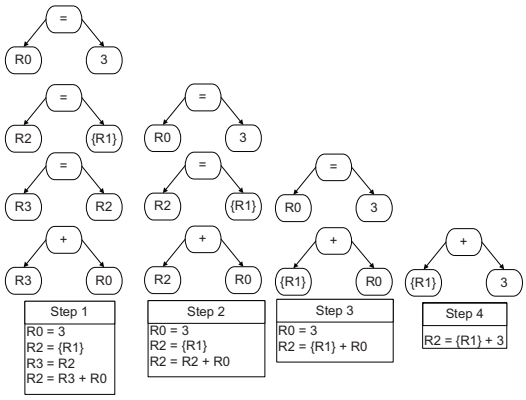

Fig. 2. Analysis of final expression

The output from the node makes braces in the final expression. Such trees should be formed for each output resource.

B. Dynamic Binary Analysis Technique

It should be noted that static analysis is a resource-intensive process because research is performed in whole PE module of analyzed software. Dynamic analysis is typically more precise than static analysis because it works with real values in the run-time mode [9]. For the same reason, dynamic analyses are often much simpler than static analyses.

Describing the dynamic binary analysis technique let's consider the concept of a linear plot of binary code. The section of binary code is linear when the section doesn't contain any instructions of control transfer to another section of the binary code. Instructions, which are nonlinear, are presented in Table I.

To analyze the binary code effectively it is necessary to determine the nonlinear plots for which we used in our previous work [8] the following algorithm shown in Figure 3. According to Figure 3, the algorithm that assigns code fragments for nonlinear and linear blocks consists of following stages:

- Stage 1.1. Algorithm reads instruction and performs preliminary decompilation;
- Stage 1.2. Algorithm defines instruction to particular class;
- Stage 1.3. Algorithm pushes instruction in the stack;
- Stage 1.4 Algorithm check the instruction relation with nonlinear instructions group;
- Stage 1.5. Algorithm checks the instruction relation with «ret» instruction.

TABLE I
Nonlinear Operations Description

Instruction type	Instruction samples		
Unconditional	Jmp		
Procedure call	Call		
	Ret		
	Int		
	Iret		
Conditional	Jl=Jnge	Jc	Jnc
	Jle=Jng	Jp	Jnp
	Jg=Jnle	Jz	Jnz
	Jge=Jnl	Js	Jns
	Jb=Jnae	Jo	Jno
	Ja=Jnbe	Jcxz	
	Jae=Jnb	Jecxz	
Loops	Loop		
	Loope		
	Loopz		
	Loopne		
	Loopnz		

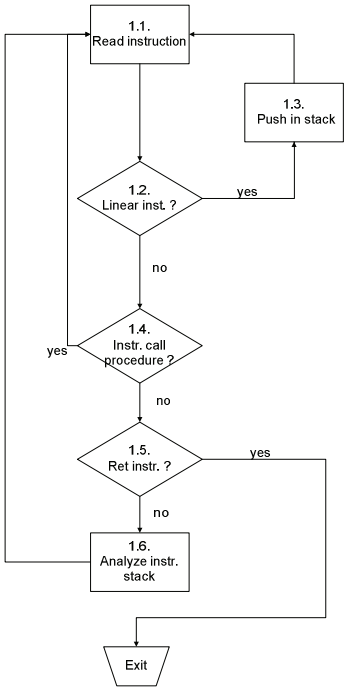

Fig. 3. Nonlinear instruction decompile algorithm

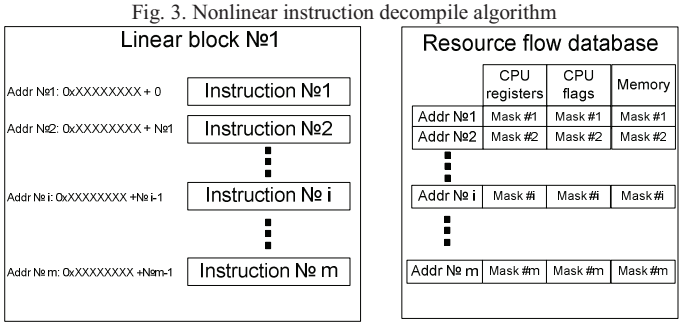

Fig. 4. Resource flow database description

The basic approach described in this paper is to provide information about program execution such as: functions calls, resources which were used on linear plots.

This approach allows us to execute some blocks of the program and perform control under CPU registers, flags and memory after each executed linear block. The results of this execution allow us to track the control and data flow of the program and describe the sequence of the program steps with saving result of each of them (Figure 4).

III. TECHNIQUE APPROBATION

These two approaches (static and dynamic) are complementary. The combination of techniques is used because the static analysis does not allow analyzing the run-time values of the program, so-called problem: «What You See Is Not What You eXecute» [10]. In case of dynamic analysis it is not possible to analyze all possible execution paths and states of the program.

Thus, in this paper we use a combination of the methods described above, the binary code sequence is tested by static and dynamic analyzer. It should be noted that the main problem in the implementation of the technologies described above is performance. During the static analysis the main performance problem that it is necessary to analyze the whole program code but in the case of large programs it is problematic. The process of dynamic analysis has also serious problems with performance because the technique used for sequential analysis of each instruction in the software [11]. Let's consider this problem in details.

To perform dataflow analysis an effective analyzer must collect as much data as possible, it should perform a single step of a certain execution path and save registers values in each step of software execution. There are several methods to perform single step in executable module. It should be noted that the described technologies have been implemented as the software tools.

1. Win 32 API debugging through «official» interfaces. To do this, we must run automatic debugger, forcing single-stepping by settings TF bit in the executive context and collecting information about registers, memory and flags. The Intel x86 processor uses complex instruction set computer (CISC) architecture, which means there is a modest number of special-purpose registers instead of large quantities of general-purpose registers. If TF flag set in TRUE, the processor will raise a STATUS_SINGLE_STEP exception after the execution of one instruction [12]. However, it is slow technique, because a context switch after each instruction and the debugger needs to retrieve context and resume execution (Figure 5).

2. Dynamic binary instrumentation (DBI) technique. Binary instrumentation is technique that modifies a binary program, either pre-execution or during execution to observe, monitors and modify a binary program. As described in Figure 6, at the first stage launcher loads test application, injects DBI dll with instrumentation code, calculates entry point of test application, injects jump instruction to DBI dll and starts the program. After jump, DBI code performs some analysis and inject second jump in second instruction. This allows doing efficient analysis in the context of the process with the ability to dynamically modify the binary code.

To compare techniques of debugging and DBI a simple program was written which runs a loop given number of times to collect some benchmarks (Figure 7). The techniques of static and dynamic analysis through DBI were tested in the task of binary code protection. In this technique, protector injects additional operations and instruction in the protected software.

That can greatly complicate analysis of binary code protection mechanism. Thus, there are the following stages of code protection which generator must perform:

1. Run the test application in the launcher and fill resource flow database.

2. Perform resources analysis which used on each step in the linear code.

3. Generate instructions which operate with unused resources at each step of the test application.

The technique of binary code protection through dynamic analysis has been realized as a software tool. The software includes: dynamic analyzer, polymorphic code generator, resource flow database.

Let's describe each module of the system:

- DBI launcher. The module triggers the test application and injects DLL with DBI code.
- Resource flow analyzer. The module performs analysis of resource flow database which described above.
- PE module analyzer. The analyzer verifies the PE file, checksum, parses the code or data sections, and defines the entry point in the test application.

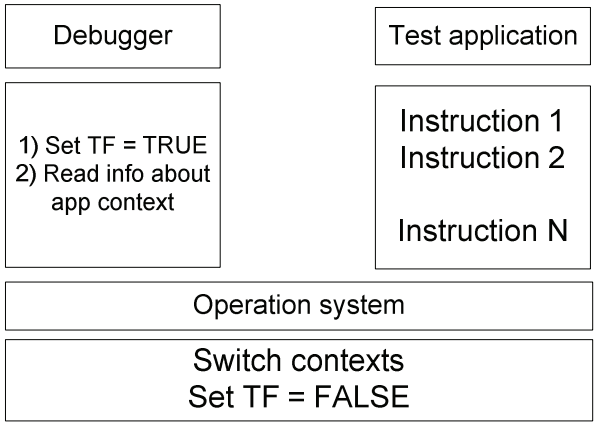

Fig. 5. Communications between Debugger and Test application

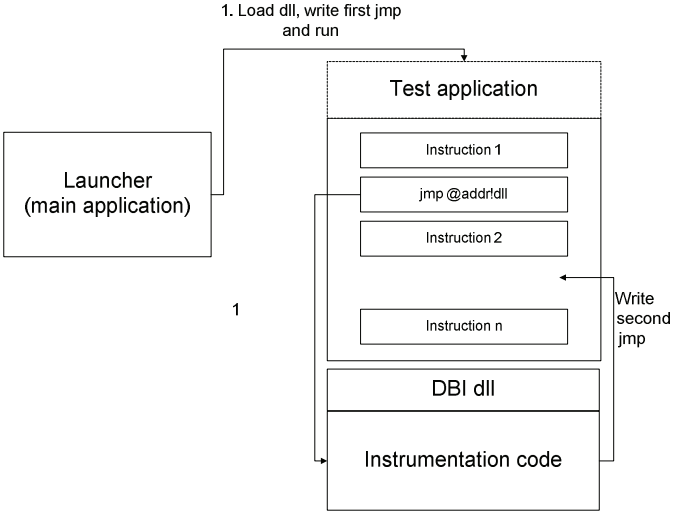

Fig.6. DBI instrumentation technique

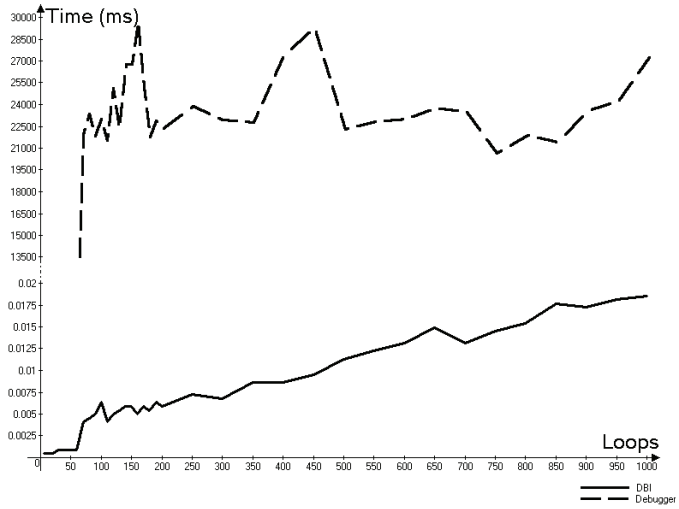

Fig. 7. Performance comparison. DBI technique over 1000 times faster than the debugging

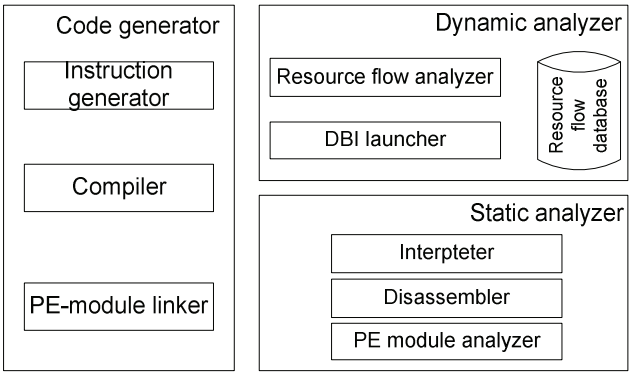

Fig. 8. Binary code protector architecture

- Disassembler. To implement disassembler module we was used existing open source library.
- Interpreter. To implement interpreter module we was used our decompiler, which was described in our previous work [8].
- Instruction generator. This module performs operations generation according with resource flow database.
- Compiler. The module performs compilation of x86 machine instructions.
- PE-module linker, assembling the resulting protected application in the single PE module.

It has a modular architecture and developed with the possibility of using modules separately. Architecture of the program is shown on the Figure 8.

It should be noted that the structure of software is portable between various platforms and flexible for addition of new processor instructions. Also, it allows the use of various software modules funds for other purposes other than the code protection to transfer various modules of the program to other

platforms (e.g., UNIX, etc.).

To evaluate the effectiveness of binary code obfuscation techniques, in this paper we have used a number of evaluation metrics.

But, unfortunately, not all metrics are useful for evaluation. We used metrics of data control flow's complexity that does not take into account the interaction between linear blocks.

On the basis of the concept of information flows *Henry & Kafur's* measure has been created [13]. On the basis of these metric it is information complexity of a procedure:

$$I = length \times (fan_{in} \times fan_{out})^2 \qquad (1)$$

Here: $length$ is complexity of the procedure's text.

- fan_{in} is the number of local flows incoming into the procedure plus the number of data structures from which the procedure gets information.
- fan_{out} is the number of local flows outgoing from the procedure plus the number of data structures which are updated by the procedure.

We can define information complexity of a module as a sum of information complexities of the procedures included into it. The next step is to consider information complexity of a module in relation to some data structure. Measure of information complexity of a module in relation to a data structure:

$$J = W \times R + W \times RW + RW \times R + RW \times (RW - 1) \qquad (2)$$

- W is the number of sub procedures which only update the data structure.
- R only read information from the data structure;
- RW both read and update information in the data structure.

One more measure of this group is *Oviedo's* measure [14]. It consists in splitting the program into linear non-intersecting sections - rays of operators which comprise the control graph of the program.

Let's define by R(i) a set of defining occurrences of variables which are situated within the range of i ray (a defining occurrence of a variable is considered to be within the range of a ray if the variable is either local in it and has a defining occurrence or has a defining occurrence in some previous ray and no local path definition). Let's define by V(i) a set of variables whose using occurrences are already presented in i ray. Then the complexity measure of i ray is defined in this way:

$$DF(i) = \sum_i^{||V(i)||} DEF(V_j) \qquad (3)$$

Where $DEF(V_j)$ is the number of defining occurrences of V_j variable from the set R(i), and ||V(i)|| is the potency of V(i) set.

Using (1) (2) (3), we evaluated the protection of a single linear block, which was implemented by the protector (Table 2). The testing linear block was consists 54 basic x86 instructions before obfuscation which perform some mathematical calculations.

TABLE II
OBFUSCATION EVALUATION

Metric before obfuscation	Test №				
	1	2	3	4	5
Henry & Kafur	7776				
	910				
Oviedo	46				
Metric after obfuscation	1	2	3	4	5
Henry & Kafur	43298	41312	44992	40381	36912
	1950	1710	3190	1670	1420
Oviedo	140	120	160	105	96

During testing of the generator, we used no more than 10 polymorphic instructions for one area between instructions. Thus, according to Table 2, the evaluation results show that the complexity of the binary code increases as the increase all three evaluation metrics before and after obfuscation.

IV. DISCUSSION

Performance. To evaluate the performance of the protector, it is necessary to analyze the performance of the test application after obfuscation. The technique also was not optimized. Our tests have shown that the part of the code after obfuscation slower 3-4 times. However, the protection slightly effect on the productivity of the test application because obfuscation performs under small areas of binary code in test application.

Limitations: Protector does not generate all binary instruction of x86-64 platform. To perform better protection mechanism more x86-64 instructions need to be modeled.

Last, protector can generate only instructions of IA-32 architecture, but this technique may be used on the different processor platforms. This requires special disassembler module and database with special instructions algorithms which depends on the processor instruction set.

Future work: Our experiments show that tool can obfuscate typical binary code blocks but without special processor instructions. An interesting future direction is to extend protector to generate difficult binary instructions and functions calls with loops and comparisons (if, else, while, for and other high-level language constructions).

V. CONCLUSION

In conclusion, it should be noted that the static and dynamic analysis technique can be widely applied in various fields of information security especially in the sections relating to the software protection which has been demonstrated in the previous sections. We also compared two techniques of binary code analysis: DBI technique and debugging and showed that the DBI over 100 times faster than debugger.

In the article we introduced a hybrid technique that includes technique of binary code dynamic execution and static analysis. We also tested technique in the tasks of software protection against unauthorized research but this technique may spread to solve the problem of binary vulnerabilities detection in the various software modules.

ACKNOWLEDGEMENTS

The research was supported by The Ministry of education and science of Russia, project № 14.132.21.1365.

REFERENCES

[1] Shudrak M. Lubkin I. «The method and code protection technique against unathorized analyze». «Software and systems» magazine, Tver, vol. 4.

[2] Sang Kil Cha, Thanassis Averginos, Alexandre Rebert and David Brumley «Unleashing Mayhem on Binary Code» in Proc. of the 2012 IEEE Symposium on Security and Privacy.

[3] Zhi Liu; Xiaosong Zhang; Xiongda Li; «Proactive Vulnerability Finding via Information Flow Tracking» Multimedia Information Networking and Security (MINES), 2010 International Conference on , vol., no., pp.481-485.

[4] Marco Cova; Viktoria Felmetsger; Greg Banks; Giovanni Vigna; "Static Detection of Vulnerabilities in x86 Executables, "Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual, vol., no., pp.269-278.

[5] Darwish, S.M.; Guirguis, S.K.; Zalat, M.S.; «Stealthy code obfuscation technique for software security» Computer Engineering and Systems (ICCES), 2010 International Conference on., pp.93-99.

[6] Haibo Chen; Liwei Yuan; Xi Wu; Binyu Zang; Bo Huang; Pen-chung Yew; «Control flow obfuscation with information flow tracking» Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on., vol., no., pp.391-400.

[7] Seeger, M.M.; «Using Control-Flow Techniques in a Security Context: A Survey on Common Prototypes and Their Common Weakness» Network Computing and Information Security (NCIS), 2011 International Conference on , vol.2, no., pp.133-137.

[8] Shudrak M. Zolotarev V.; «The new technique of decompilation and its application in information security» in Proc. of the 6th European Modelling Symposium (EMS-2012), 2012.

[9] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A. Fisher. Deli: A new run-time control point. In Proceedings of the 35th Annual Symposium on Microarchitecture (MICRO35), pages 257 - 270, Istanbul, Turkey, November 2002.

[10] Balakrishnan, G. and Reps, T., WYSINWYX: What You See Is Not What You eXecute. In ACM Transactions on Programming Languages and Systems, vol.32 Issue 6, August 2010.

[11] Qixue Xiao; Feifei Ren; Jing Zhao; Lan-lan Qi; «Survey of Dynamic Taint Propagation for Binary Code» Instrumentation, Measurement, Computer, Communication and Control, 2011 First International Conference on , vol., no., pp.392-395, 21-23 Oct. 2011.

[12] TF flag description. Published at http://msdn.microsoft.com/en-us/library/windows/hardware/ff561502(v=vs.85).aspx.

[13] Henry S, Kafura D. IEEE Transactions on Software Engineering Volume SE-7, Issue 5, Sept. 1981 Page(s): 510-518.

[14] E. Oviedo, "Control Flow, Data Flow and Program Complexity," Proc. COMPSAC80, pp. 146-152.