International Journal of Multimedia and Ubiquitous Engineering
Vol.11, No.10 (2016), pp.33-44
http://dx.doi.org/10.14257/ijmue.2016.11.10.03

The Interactive Mechanism of Static and Dynamic Analysis in the Reverse Analysis of Embedded Software

Liu Tie-ming[1,*], Jinag Lie-hui[1], Zhu Jing-si[2] and Meng Gang[1]

[1]The State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, 450000, P. R. China;
[2]ZhengZhou University of Science and Technology, Zhengzhou, 450064, P. R. China
{fxliutm, jingliehui }@163.com, {zjs006, zzxdmg}@126.com
[*]Corresponding Author: LIU Tie-ming(fxliutm@163.com)

Abstract

Because the software reverse analysis method which combined the dynamic and static analyses lacks normative interactive mode, the work of the software reverse analysis is inefficient, and its reusability is poor. Based on dynamic and static analysis process of the embedded software, three kinds of interactive mechanism are proposed, including Static To Dynamic (STD), Dynamic To Static (DTS), Static and Dynamic simultaneous (SDM), and has also presented the method of realizing these three interaction mechanisms in detail. The test results show that interactive mechanisms of STD, DTS and SDM are suitable for correction of abnormal nodes in the results of static analysis, optimization of dynamic information extraction, identification of hidden codes and so on. It can greatly improve work efficiency of the embedded software reverse analysis.

Keywords: *software reverse analysis; embedded software; static analysis; dynamic analysis; dynamic and static combination; interactive mechanism*

1. Introduction

The technology of combined the dynamic and static analyses can be divided into three types by analytical approaches: static analysis, dynamic analysis and combination of both static and dynamic analyses, and it is usually used in the fields of binary-translation, decompilation, vulnerability detection, malicious behavior analysis and so on. Due to the lack of run-time information, static analysis can't accurately analyze indirect branch statements, having the shortcomings of false analysis and omissive analysis[1-2]; Dynamic analysis has the problems of incomplete analysis and detection over the codes because it can't traverse all the possible execution paths of the codes [3]; the method of the software reverse analysis can integrate the results of dynamic analysis and the results of static analysis, effectively lowering the probability of false analysis and omissive analysis. It has been widely used in the combined the dynamic and static analyses on common platforms. However, it lacks an uniform information interaction mechanism between the dynamic and static. The reusability of the analysis work is not strong and the analysis efficiency is relatively low.

With the development of software technology and embedded technology, various software vulnerabilities and malicious attacks based on software vulnerability gradually spread from such common platforms as x86 to such embedded platforms as ARM, MIPS, PPC [4]. Because of large differences among hardware architectures of the embedded platform, the system software can be arbitrarily cut freely, so the software reverse analysis tools in universal platform are unsuitable in embedded platform. Therefore, the reverse analysis work of embedded software is facing severe challenges [5]. Based on the

ISSN: 1975-0080 IJMUE
Copyright ⓒ 2016 SERSC

dynamic and static analysis process of the embedded software, the interaction mechanism of static and dynamic analysis which is applicable to the reverse analysis of embedded software has been put forward in this paper, so as to provide support for improvement of accuracy and efficiency of combined the dynamic and static analyses of the embedded platform.

2. Static and Dynamic Analysis Process of the Embedded Software

Because the hardware architectures of the embedded platforms are different, even for the same architecture with different series, the Instruction Set Architecture (ISA) can be different. In addition, because the versions of embedded operating systems are various, and can be cut any part and the running time circumstances are different, so it is very difficult to provide the united static disassembler and decompiler for the software of the embedded platform for static analysis. And it is difficult to provide the united dynamic execution engine for dynamic analysis, too. Figure 1 is the interaction mechanism of static and dynamic analysis aiming at the reverse analysis of embedded software, in which the static analysis section consists of the relevance platform transparent processing and static multidimensional graphs extraction, and dynamic analysis section consists of multi-architecture dynamic execution engine and platform-independent dynamic information extraction.

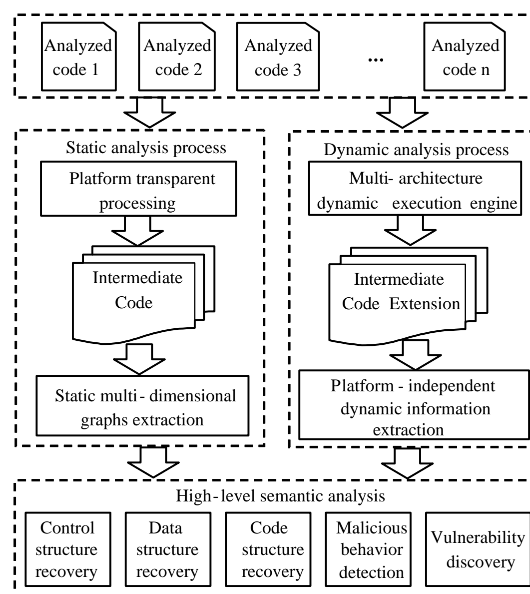

Figure 1. Process of Static and Dynamic Analysis in Embedded Software

Platform transparent processing means a process of using formalized methods to abstract and describe the attributes related to the Instruction Set Architecture (ISA) and the system development platform ABI in the analyzed codes on various analysis platforms, and then converting the binary codes on various platforms to the unified intermediate language codes form by code conversion on the basis of the abstract description (the universal disassembler[6] and the intermediate code converter [7-8]).

Static multidimensional graphs extraction is a process based on unified intermediate language codes, which adopts various static analysis algorithms [1,8] to complete the analysis of control flow and data flow and generate the dependency diagram, the control flow diagram, the structure diagram, the calls diagram and so on under the granularity of statement level, basic block level, structure level, function level and so on, and then display them in a graphical manner.

 Copyright ⓒ 2016 SERSC

The multi-architecture dynamic execution engine refers to the generation platform that applies the multi-architecture simulators like QEMU [9], through the proper configuration to set up a simulation platform that is suitable for the execution of various embedded software, which is a virtual execution environment provided for the dynamic execution of embedded software.

Whereas platform-independent dynamic information extraction is based on the multi-architecture dynamic execution engine, through the intermediate language extension, to establish the instrument interface based on intermediate language, and meanwhile to provide the API (sentence level, basic block level, function level, *etc.*) of dynamic information extraction with different granularities, which is the run-time information extraction interface during the dynamic execution process of embedded software offered to users.

Through static analysis process, the users can get a series of graphs in intermediate language codes; through dynamic analysis process, the users can get run-time information of intermediate language codes layers with different granularities. However, in order to better combine the two organically and better provide service for the high-level semantic analysis such as vulnerability discovery and malicious activity detection, an effective information interaction mechanism must be established between the static and dynamic analyses.

3. Interactive Mechanism of Static and Dynamic Analysis

After intensively researching the analysis cases of the software reverse analysis on the current platform which is used universally, it is discovered that the graphs got in the static analysis can be applied to guide the path which should be covered during the dynamic information extraction, and the information extracted from the dynamic analysis can be applied to correct the inaccurate places in the results of the static analysis. Building up an interactive mechanism between static analysis and dynamic analysis process, integrating the results of dynamic and static analyses, can greatly improve the accuracy and efficiency of the high-level semantic analysis. The interactive mechanism, according to the order of dynamic and static analysis, combined with the target of analysis, can be categorized into three basic types: Static To Dynamic (STD) interaction mechanism used to amend results of static analysis, Dynamic To Static (DTS) interaction mechanism used in quick analysis of the code behavior, and Static and Dynamic simultaneous mechanism(SDM) used in identifying hidden code.

3.1. Static to Dynamic Interactive Mechanism (STD)

The interactive mechanism of Static To Dynamic is called STD mechanism for short, which is usually used to dynamically correct the results of static analysis (for example, the correction on static decompilation results, *etc.*), and is the common way of combined the dynamic and static analyses on present common platforms. STD mechanism first utilizes modules including static analysis process and platform transparent processing through static multidimensional graphs extraction to get all kinds of analysis graphics based on the unified intermediate language code, and then uses methods of combining the abnormal nodes that appear in the graph (such as statements that appear in the nodes without being recognized and that are marked as unknown(string,τ), jump statements that the jump target is not within the derived API or the code section, as well as particularly complicated memory read-write statements and particularly complicated expression statements) and utilizing symbol execution and solving with constraint path to generate testcase; Then based on the testcase start dynamic analysis procedure, running code on dynamic execution engine, and use relative dynamic information extraction interface to proceed dynamic tracing. Finally, use the dynamic extraction information to correct the

Copyright ⓒ 2016 SERSC

inaccurate places in the results of the static analysis abnormal nodes. The sketch map of STD interaction mechanism is as shown in Figure 2.

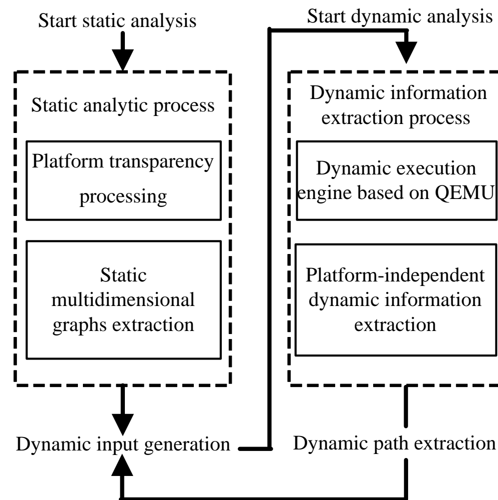

Figure 2. STD Interactive Mechanism Diagram

If SS is used to represent starting the static analysis process, DIG represents dynamic input generation, SD represents starting dynamic analysis process, and WB represents dynamic message write-back, then the static-to-dynamic information interactive mechanism can be simply represented by (SS, DIG, SD, WB)+. The symbol "+" means that this process can be repeated for many times, which is that after one STD information interactivity, the results of static analysis will be updated once. If there are new reachable code blocks after the updating, then the STD information interactivity is started again for the new code blocks. The process is repeated until there are no more reachable code blocks.

3.2. Dynamic to Static Interactive Mechanism (DTS)

The interactive mechanism of Dynamic to Static is called DTS mechanism for short, which is usually used to the rapid analysis and feature extraction of the code behavior. First, start dynamic analysis process to run the executable program on the dynamic engine. Meanwhile dynamic information extraction module is used to record the dynamic execution path; Then based on the dynamic execution path, starting the static analysis process, namely use static analysis process module in platform transparent processing to change the code in dynamic path into intermediate language code, and use multidimensional graphs extraction module to optimize it, to generate dynamic path's corresponding varies collections of graphs, and to realize the user's rapid analysis and judgment of code behavior. The sketch map of DTS mechanism is shown as Figure 3.

If SD represents starting dynamic analysis, TT represents dynamic path extraction, CS represents calling static analysis algorithm, and TO represents dynamic path optimization, then the dynamic-to-static information interactive mechanism can be simply represented by (SD, TT, CS, TO).

 Copyright ⓒ 2016 SERSC

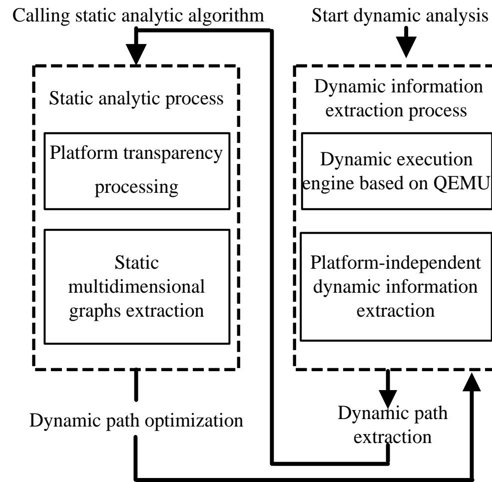

Figure 3. DTS Interactive Mechanism Diagram

3.3. Static and Dynamic Simultaneous Interactive Mechanism (SDM)

The interactive mechanism of Static and Dynamic simultaneous is called SDM mechanism for short, which is usually used to identify self-modifying codes or hidden codes. First, we need to call static analysis process and DTS interactive process respectively, and finish the static analysis and the code behavior analysis from the dynamic state to the static state; Then match the results generated by the static analysis and dynamic analysis, and through the match, find out and identify the possible differences between the results of the static analysis and dynamic analysis and then mark the differences as possible hidden codes. Under this interactive mechanism, the dynamic path analyzed and extracted from the dynamic analysis need to be submitted to the static analysis process for handling, namely to utilize DTS mechanism. But the static analysis process and the DTS interactive mechanism don't affect each other, they only take the results gained separately for matching analysis. Figure 4 is the sketch map of SDM interactive mechanism.

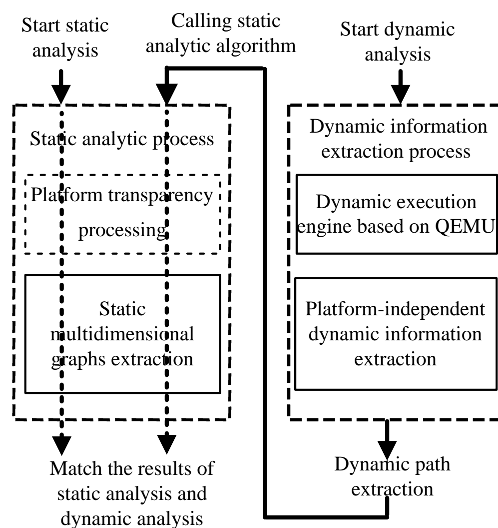

Figure 4. SDM Interactive Mechanism Diagram

Copyright ⓒ 2016 SERSC

If SS is used to represent starting the static analysis process, DTS is used to represent starting up analysis process from dynamic state to static state and MACH is used to represent the matching process of results of dynamic and static analysis, then SDM information interactive mechanism can be simply represented by (SS,DTS, MACH).

4. Realization of Interactive Mechanism in Static and Dynamic Analysis

Actually, three kinds of interactive mechanism are the control of reverse analysis in embedded software. The key to realize mechanism of static and dynamic analysis is how to integrate the results of static and dynamic analysis.

4.1. Realization of STD Mechanism

In essence, the Static to Dynamic Mechanism guides the path of the dynamic analysis by using the abnormal nodes in the static analysis, and then uses the information extracted by dynamic analysis to modify the abnormal results. So firstly we should use the abnormal nodes of the static analysis to provide input examples for the dynamic execution, making the dynamic analysis can cover the abnormal nodes; Then complete the extraction of information on the dynamic path by using extraction plug-in; Finally, correct the abnormal nodes by using the extracted run-time information.

4.1.1. Dynamic Input Generation Based on the Abnormal Nodes: Definition 1 Abnormal Node

A node in control flow graph (CFG) generated by static analysis process is called an abnormal node ,if it matches one of the following conditions:

(1) A node in which statements that is not identified and marked as unknown*(string,* τ *)* appear;

(2) A node in which jump statements whose jump target doesn't exist in the derived API or in the code area appear;

(3) A node in which jump statements or call statements whose destination address is marked as □ (□ means the destination address is arbitrary);

(4) A node in which particularly complex memory read-write statements or particularly complex expression statements appear.

Definition 2 Partial Path: A sequence of instructions that satisfy the following constraint conditions $\{inst_0, inst_1, \cdots, inst_k\}$ is called a partial path, denoted as: T^P

(1) $inst_0$ is the program entry point, and $inst_k$ is an Abnormal Node;

(2) $\forall i(0 \le i \le k)(inst_i, inst_{i+1}) \in E$, E is the set of the edges in CFG.

Definition 3 Construct the Set of Partial Path: Construct all Partial Path from the entry point to all Abnormal Node in CFG which gained based on static analysis.

It usually adopts the reverse depth-first search algorithm to construct the set of partial paths: namely first from the beginning of the largest abnormal nodes number, backward search for the current node precursor, and choose one from its precursor list to continue searching (if the precursor is an Abnormal Node, it will be selected in priority), until reaching the entry point (note: in this case, the node is the basic block).

Attention: When constructing the set of partial path, if there exists a loop in the paths, we will get different paths if the loop times are different. And generally the loop executing times cannot be determined by static analysis, so it's necessary to take a reasonable strategy to process the loops [10] when constructing the set of partial path.

After the partial path set is obtained, at first, the whole partial paths should be scanned from the entry point; When encountering conditional jump statements, the conditional expression *Expr i*n the current conditional statements is required to be figured out, then let *Expr = True* or *False* (namely, the constraint condition that partial paths should meet to

 Copyright ⓒ 2016 SERSC

jump on current conditions); Combining all the constraint conditions corresponding to every branch statement, the path constraint condition that covers the current partial path should be obtained; Finally, using the constraint solving tool STP[11] to automatically work out a group of inputs satisfying the constraint conditions. These inputs are utilized to enable the program to execute along a given path and cover the abnormal nodes.

4.1.2. Plug-Ins for Extracting Dynamic Path

At present, the relatively widely used plug-ins for extracting dynamic path are mostly on the instruction level [12]. To trace a tiny process dynamically, it usually takes a dozens of gigabytes on hard disk to record the path information and it often needs to filter the path according to the specific analysis content. Therefore, a kind of multi-granularity plug-ins for extracting dynamic route is designed here which can set the granularity of path extraction according to the need of analysis, and provide support for the analyzer to carry out the work more purposefully. The major performance function list included in the multi-granularity dynamic path extracting plug-ins is as shown in Table 1:

Table 1. Major Function List Included in the Multi-Granularity Dynamic Path Extracting Plug-Ins

Granularity	Function name	Function description
User Statement level	*trace_inst_user(int pid, char *tracefile)*	Monitor the process whose id number is *pid*, and record the extracted statement-level path information in the file pointed by *tracefile*, not including the system code
User Basic block level	*trace_bb_user(int pid, char *tracefile)*	Monitor the process whose id number is *pid,* and record the extracted basic-block-level path information in the file pointed by *tracefile*, not including the system code
User Function level	*trace_func_user (int pid, char *tracefile)*	Monitor the process whose id number is *pid*, and record the extracted function-level path information in the file pointed by *tracefile*, not including the system code
Whole system Statement level	*trace_inst_ws(int pid, char *tracefile)*	Monitor the process whose id number is *pid* and record the extracted statement-level path information in the file pointed by *tracefile*, including the system code
Whole system Basic block level	*trace_bb_ws(int pid, char *tracefile)*	Monitor the process whose id number is *pid* and record the extracted basic-block-level path information in the file pointed by *tracefile*, including the system code
Whole system Function level	*trace_func_ws(int pid, char *tracefile)*	Monitor the process whose id number is *pid* and record the extracted function-level path information in the file pointed by *tracefile*, including the system code

When users need dynamic path information of different granularities, they can just use the performance function of the corresponding granularity. Functions in different granularities not only need to use instrumentation interface **_stub_enable( )* of different granularities, but also need to determine whether the entry address and exit address of the code basic block are in the user's virtual address space, in order to make sure that whether it is necessary to record the path information of system functions. In addition, the recording format of path files is stored according to the fixed data structure . When you need to display the path files, just read them in accordance with the format of the data structure.

Attention: Make use of instrumentation interface of the intermediate language to realize the extraction of dynamic path, the information stored in the file path does not include the information of the intermediate code and still only records the information of machine instructions corresponding to the intermediate language, which takes the

Copyright ⓒ 2016 SERSC

efficiency of storage and convenience of reading into consideration. What's more, the binary coding of machine instructions obtained from the file path can be transformed into the corresponding assembly instructions and the intermediate language instruction by calling the universal disassembler, so there is no need to store intermediate codes in the file path.

4.1.3. Corrections of Abnormal Nodes

According to the abnormal node type, path extraction functions of different granularities are called and trace files that record the path information are obtained. Then, search the path information corresponding to abnormal nodes according to the information recorded in the trace files.

The path information includes virtual address corresponding to dynamic execution statement as well as the register information and the memory units information when the statements are executed. Based on the address range of sentences corresponding to abnormal nodes in the results of static analysis, search the corresponding content in the path information, and the corresponding abnormal nodes are rewritten, namely finish the corrections of abnormal nodes. Suppose the addressing of target address of jump instruction in address A is register indirect addressing, and it is marked abnormal. Then through the construction of partial paths and paths constraint solving, the input of dynamic execution can be obtained. The basic-block-level path extracting plug-ins are used to get dynamic path information. By means of searching the path information whose address is "*A*" in the path file, reading the numeric value "*addr*" stored in the corresponding register. In addition, read out the data stored in memory address "*addr*" through the memory access interface. Eventually, the data is used to replace the target address of jump instruction.

4.2. Realization of DTS Mechanism

The dynamic-to-static mechanism is mainly used to optimize dynamic path information and help users accomplish analysis work better. So first of all, using dynamic path extraction plug-ins unit to extract the path information, and then call the algorithm in the static multidimensional graphs extraction algorithms to optimize accordingly.

The dynamic path extraction above is using extracting plug-ins of the dynamic path introduced in 4.1.2 to perform extraction, that is, calling different dynamic path extraction functions according to the granularity of dynamic tracing. As the information stored in the path file includes only the instruction binary codes and does not include the corresponding intermediate code, therefore, the binary codes corresponding to all the path information need be extracted from the path information, then deliver the binary codes to platform independent transparent framework and convert them into unified intermediate language.

Call the corresponding algorithms in the algorithms library of different granularities in static multidimensional graphs extraction framework for optimization according to the granularity of the extraction path information. For example, by calling constant propagation and using definition analysis in the statement-level algorithms library, we can delete the dead codes and reduce the number of statements. Furthermore, the reconstruction algorithm of control flow graph in the basic block-level algorithms library is used to get the control process of the dynamic path and can easily find and recover the loop body and the loop count according to the address of the basic block.

4.3. Realization of SDM Mechanism

The mechanism of SDM can be used to discover hidden codes. Static analysis part of the executable program is completed by platform transparent and static multidimensional graphs extraction. Dynamic analysis part makes use of the plug-ins for dynamic path extraction to accomplish the extraction of statement-level (generally refers to user

 Copyright ⓒ 2016 SERSC

statement level, only refers to system statement level in analyzing system code) path information; Then deliver the path information to the part of static analysis, and calls universal disassembler, intermediate code convertor and the optimization algorithms in static multidimensional graphs extraction framework, to complete the static optimization of the dynamic results. Finally, the results of static and dynamic analysis will be matched.

The matching method is mainly using the mapping relationship between addresses, namely the path file extracted dynamically recording the corresponding virtual address of each instruction, and the record in the static analysis of each statement address is a virtual address. Therefore, for the virtual address *dva* of any statement in a dynamic path, the virtual address of its corresponding static code *va* = *dva*.

Suppose (N_s, E_s) denotes the static abstract control flow graph and (N_d, E_d) denotes the dynamic abstract control flow graph. Ns and Nd denotes the node collection and Es and Ed denotes the side collection, *DVA()* denotes the instruction virtual address in the dynamic extraction path file and *VA()* denotes the virtual address of the nodes in the static analysis. During the matching process, the following non-match situation may occur:

(1) For $(d_i, d_k) \in E_d, s_i \in N_s, VA(s_j) = DVA(d_j)$, but for any *sid* (s_j, s_x) starting from s_j, $VA(s_x) \neq DVA(d_j)$, which means the dynamic path reaches a branch which is not recognized by the static control flow graph;

(2) For $(d_i, d_k) \in E_d, s_i, s_k \in N_s, VA(s_i) = DVA(d_i)$, $VA(s_k) = DVA(d_k)$, but $(s_i, s_k) \notin E_s$, which means the dynamic path reach a non-existing side in the static control flow graph.

(3) The dynamic and static control flow graph can be matched, and there is at least a node "*n*" in dynamic control flow graph so that the code statement which node "*n*" contains is different from the code contained in the corresponding node in the static control flow graph.

When the first situation occurs, it means that a code block which is not identified by the static analysis has been found. The static analysis can be carried out again from address $DVA(d_i)$; When the second situation occurs, it means that a side which has not been identified by the static analysis has been found, we can add (s_i, s_k) to the E_s set; When the third situation occurs, it means that the codes corresponding to the node "*n*" are modified at running time. They are the real codes and can replace the sentences in the corresponding static node.

5. Test

5.1. Test of the STD Interactive Mechanism

As the STD interactive mechanism is mainly applied to modify the abnormal nodes in the results of static analysis, and improve the accuracy of decompilation. Therefore, we choose some procedures in the open source suite coreutils8.20 which may contain abnormal nodes and C++ program: pointer.cpp and funcpp.cpp which containing pointers and virtual functions for the test. After cross-compilation the program is converted into an executable program in ARM platform. The number of instructions and the number of processes obtained through the universal disassemble are listed in Table 2. First make use of the framework of static multidimensional graphs extraction to extract the CFG (control flow graph) and count the number of abnormal nodes. Then use partial path constraint solving to get the dynamic input. Through the dynamic execution and calling the basic-block-level extracting plug-ins of the dynamic path, we can get the basic-block-level path information. After that, through matching the path information with the result of static analysis, the corresponding abnormal node information can be modified. The test results are shown in Table 3.

Copyright ⓒ 2016 SERSC

Table 2. Basic Information about the Test Cases

Test object	Number of instructions	Number of processes
Dd	2303	36
factor	2531	35
install	1002	19
Join	1185	26
Ls	4882	78
Cat	783	6
pointer	312	4
funcpp	549	8

Table 3. STD Interactive Test

Test object	Number of abnormal nodes (ty=3)	Number of abnormal nodes (ty=4)	Number of partial paths	Number of corrected abnormal nodes
dd	4	6	7	10
factor	2	3	3	5
install	1	2	2	3
join	2	1	2	3
ls	4	7	9	11
cat	1	1	2	2
pointer	1	2	2	3
funcpp	1	1	2	2

Judging from the static-to-dynamic interaction mechanism, the two main abnormal nodes in the tested program are Type 3 and Type 4(definition in 4.1.1), namely the indirect jump address cannot be determined, and the expressions for storing operation address are too complex. After dynamic information extraction, we have corrected all the abnormal nodes and the accuracy of the control flow graphs have been improved.

5.2. Test of the DTS Interactive Mechanism

DTS mainly used to optimize the path file extracted dynamically. We choose to use part of the programs in the open source suite coreutils8.20 for DTS interactive mechanism of test and cross-compile it to the executable programs in PPC platform. First of all, run it on dynamic execution engine, and use the user statement-level path extracting plug-ins extract path file. Then extract the corresponding binary codes and optimize the codes by calling the process of static analysis. Test program information and test results are shown in Table 4:

Table 4. Test Results of DTS

Test object	Number of dynamic executing instructions	Number of process	Number of instructions after optimization
pr	5716	38	2109
ptx	4253	22	1985
sort	8967	96	4513
tail	4420	32	3254
tr	3968	35	1256
pr	5925	38	2784

 Copyright ⓒ 2016 SERSC

ptx	4369	22	2241
sort	9458	96	5087

It can be seen from the results that, after static optimization, route number of dynamic execution are reduced obviously, which can reduce the work amounts of analysis personnel to a large extent.

5.3. Test of SDM Interactive Mechanism

SDM interactive mode is mainly used to discover the hidden codes, therefore we choose selfmodify1.c and selfmodify2.c, which are written by ourselves and contain self-modified codes, as the targets of test. Utilizing gcc4.3 to compile the test objects to executable programs on x86, and then use static and dynamic respectively to analyze, and send the dynamically extracted instruction binary codes which correspond to path information to the static analysis part to have another analysis. Finally match the two results of static analysis. The test program and the final results are shown in the Table 5.

According to the test results we can see that just once matching using SDM may not find self-modifying codes. The reason is that dynamic execution is random and the self-modifying codes may not be covered.

Table 5. Test Results of SDM

Test object	Number of instructions in static analysis	Number of dynamically executed instructions	Whether the static results match the dynamic results	Nonmatch types
Selfmodify1	345	469	No	1、3
Selfmodify2	216	427	No	1

6. Conclusion

This paper systematically studies the combination mode between the static and dynamic interactive mechanism, establishes three kinds of interactive mechanism including Static To Dynamic (STD), Dynamic To Static (DTS), Static and Dynamic simultaneous (SDM). In addition, by analyzing different interactive mechanisms' applications and functions, this paper gives detailed methods and realizes mutual supplement and modification between the dynamic information and static information, which has provided strong support in finding the hidden codes, exploiting vulnerability effectively, and improving analysis efficiency, *etc.*

The experimental test results have proved the efficiency of the three interactive mechanisms. However, as for the DSM mode, we have currently only conducted test the matching results, without testing the matching time. In the follow-up work, it's necessary for us to conduct relevant tests on the time and space complexity of the matching algorithm and further improve the matching algorithm on the basis of the test results.

Acknowledgements

This work is supported by National Natural Science Foundation of China and National High Technology Research and Development Program of China.

References

[1] J. Kinder, “Static Analysis of x86 Executables”, Technische Universität Darmstadt, **(2010)**.
[2] B. G. Reps and T. Wysinwyx, “What you see is not what you execute”, ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 32, no. 6, **(2010)**, pp.23-24.
[3] E. J. Schwartz, T. Avgerinos and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)”, IEEE Symposium on

Copyright ⓒ 2016 SERSC

Security and Privacy, The Claremont Resort, Okland, California, USA, **(2010)**, pp.317-33.
[4] P. Clarke, "Embedded systems next for hack attacks", http://www.esmchina.com/ART_8800125181_1400_2304_3803_0_5f29b1c0-02.HTM, **(2015)**.
[5] DIMITRIOS N Serpanos, ARTEMIOS G Voyiatzis. Security Challenges in Embedded Systems. ACM Transactions on embedded computing systems (TECS), **(2013)**,12(1s): No.66
[6] J. L. Hui, "Research on Key Techniques for Firm-Code Reverse Analysis", PH.D thesis, Information Engineering University, Zhengzhou, China, **(2007)**.
[7] C. Cifuentes, "The University of Queensland Binary Translator (UQBT) Framework", the University of Queensland and Sun Microsystems, Inc., **(2002)**.
[8] L. X. Ying, "Research on Technologies of Control Flow Reconstruction and Control Structure Recovery in Decompilation", Information Engineering University, M.D. thesis, Zhengzhou, China, **(2010)**.
[9] B. F. Qemu, "A Fast and Portable Dynamic Translator. Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference", Marriott Anaheim, California, USA, **(2005)**, pp.41-46.
[10] L. Dan, "Research on control Flow Reconstruction of Multi-source Decompilation", M.D. thesis, Information Engineering University, Zhengzhou, China, **(2013)**.
[11] V. Ganesh and D. L. Dill, "STP: A Decision Procedure for Bit-vectors and Arrays", Computer Aided Verification Lecture Notes in Computer Science, vol. 4590, **(2007)**, pp. 519-531.
[12] Y. Y. Qiu, "Research and Application on the ARM architecture of full system dynamic analysis technology", M.D. thesis, Information Engineering University, Zhengzhou, China, **(2014)**.

Authors

Liu Tie-ming, received B.S. degree and M.S. degree in computer science from Tsinghua University in 2003 and 2006, Beijing, China. Currently, He is a Ph.D. student and vice-professor in the State Key Laboratory of Mathematical Engineering and Advanced Computing of China. His research interest includes system reverse engineering and embedded system.

Jiang Lie-hui, received B.S. degree, M.S. degree and Ph.D degree in computer science from the State Key Laboratory of Mathematical Engineering and Advanced Computing of China, Zhengzhou, China. Currently, He is a researcher and professor in the State Key Laboratory of Mathematical Engineering and Advanced Computing of China. His research interest includes information security, system reverse engineering and embedded system.

Zhu Jing-si, received B.S. degree and M.S. degree in Zhengzhou University of Science and Technology in 2006 and 2009, Zhengzhou, China. Currently, She is a researcher and assistant professor in Zhengzhou University of Science and Technology. Her research interest includes system reverse engineering and Computer-Aided Design.

Meng Gang, is a M.S. course student in the department of Computer Science at the State Key Laboratory of Mathematical Engineering and Advanced Computing of China. He is also assistant researcher at the Information Engineering University. He is current research interests are system reverse engineering and embedded system.

 Copyright ⓒ 2016 SERSC