Applied Mathematics & Information Sciences
An International Journal

An Input Data Related Behavior Extracting and Measuring Model

Dan Wang, Min Dong, Wenbing Zhao

College of Computer Science, Beijing University of Technology, Beijing 100124 P.R.China

Received: Sep. 20, 2012; Revised Oct.4,2012; Accepted Oct.24,2012
Published online: 1 Mar. 2013

Abstract: It is difficult to dynamically assess the runtime trustworthiness of a software program. Improperly validated user input is the underlying root cause for a wide variety of attacks on applications. This paper proposes an approach for constructing a trusted software behaviour model related with the input data for identifying and tracking the insecure information flows based on dynamic tainting analysis and dynamic slicing technology. It can tag and track user input at runtime and prevents its improper use to maliciously affect the execution of the program. We regard an instruction as a basic analysis unit and focus on information flow caused by variable assignment, the information flow of each instruction is defined as its behaviour specification. During the execution, instructions that use untrusted variable are tracked to determine whether the address modified by the instructions belongs to the specification or not. A method of extraction and checking of the behaviour specification was researched and designed. In order to prove for efficiency and performance of the model, a set of tests were conducted, and preliminary results show the validity of our approach.

Keywords: dynamic behaviour, taint, information flow, slice technology

1. Introduction

The increasing size and complexity of modern software systems lead to an increasing number of security vulnerabilities, such as buffer overflow, heap corruption, format string, integer overflow, etc. By carefully exploiting these vulnerabilities, attackers may cause severe damages to the running process or even ultimately gain the control of victim computers.

In the trusted software computing field [1], the solution for a software system to reduce be attacked is to construct softwares expected behavior, then to verify its conformation with its expected ones during runtime. If a software systems behavior and runtime result can meet its users expectation and can provide continuous service when strained by malicious interference, it can be regarded as a kind of trusted software system. It is evident that how to construct and verify expected softwares behavior is the fundamental work. On the other hand, as we know, many vulnerabilities in various applications are caused by permitting unchecked input to take control of the application, which an attacker will turn to unexpected purposes [2–4]. For example, improper input validation accounts for most security problems in database and web applications [5]. If an attacker tampers with important data of the process by using existing vulnerability when the software is running, such as modification of a functions return address, a function pointer, etc., it can interfere with or change the behavior of the software. The optimal approach to prevent attacks caused by input data would be to eliminate the vulnerabilities in the affected applications. It means an application must properly validate all input data and its related data.

In order to reduce the code amount that needs to be examined when validating the insecure flow caused by input data, this paper focuses on modifying the outcome of specific conditions by narrowing the search space to include only untrusted sensitive input, and we need only take into consideration those instructions that either directly or indirectly process untrusted input data.

This paper proposes an approach for constructing a software behavior model related with input data for identifying and tracking the insecure information flows [6]. Our approach is based on dynamic tainting analysis and dynamic slicing technology to ensure a software systems behavior to act as its expectation. Based on our model, the

* Corresponding author: Email: wangdan@bjut.edu.cn

produced information flow of untrusted data is tracked inside the application. Whenever an attempt to relay such information is detected, the user is warned and given the possibility to stop the transfer.

The idea behind dynamic taint analysis [**?**] is to tag untrusted data and track its propagation through the system. Any new data derived from untrusted data is also tagged. If tainted data is used in a potentially unsafe manner, such as executing a tagged SQL command or dereferencing a tagged pointer, a security exception is raised. More recently, researchers have started to investigate the use of tainting based approaches in domains other than security, such as program understanding, software testing, and debugging. On the other hand, dynamic slicing technology can compute a conservative estimate of all instructions in a program that are either affected by or affecting the value of a variable at a specific program point and for a given execution, and it widely be used in software analysis field.

In our study, we firstly analyze code for the presence of taint vulnerabilities related with input data, and then dynamically track tainted data at runtime. The dynamic tainting analysis is used to detect insecure information flows about an input data and the dynamic slicing is gotten. It focuses on computing which subset of the data in the program is affected by a given input set of data, then the dynamic slicing technology is used to extract the relevant portion of the code, which is the set of instructions that propagated information along the insecure flow. This set of instructions, which is called concerned instructions in this paper, is used to reduce the amount of code that needs to be examined instead of the tagging the untrusted data.

The rest of this paper is structured as follows. In Section 2, we present related works. In Section 3, we introduce our concerned instruction set and its extracting approach. Then, in Section 4, we discuss our dynamic checking and some implementation issues, respectively. Section 5 presents the evaluation of our approach, and, finally, Section 6 concludes this paper.

2. Motivation and Related works

At present, to help deal with software debugging and verification, a variety of runtime checking and tracking approaches have been proposed. Among them, a number of these proposals have adopted dynamic taint propagation technology, and dynamic taint tracking and checking has been widely accepted as a promising mitigation method. Based on the observation that attacks are always launched from suspicious I/O channels such as files or network sockets, it seeks to capture the essence of attacks. It treats data from those suspicious input channels as tainted data and keeps track of the tainted data propagation as they may directly or indirectly affect other data values in the program. Besides, various approaches have been explored to attack the taint problem. Broadly, these fall into two categories including statically analyzing code for the presence of taint vulnerabilities, and dynamic approaches that track tainted data at runtime. However, although there has been significant previous work in the use of tainting to protect user applications, there is a few of research to combine the construct software behavior model with tainting checking technology.

From the view point of software behavior, there has been some research on constructing software expected behavior models. In 1996, Forrest et al. proposed a behavior model N-gram [9] based on a short sequence of system calls. Inspired by that, other research based on system calls sprang up. Reference [10] proposed a model combining static analysis and dynamic binding, which had a more powerful capability of detection and lower rates of false alarm. Reference [11] proposed a model utilizing FSA to construct a calling sequence model, which can describe the structure of loop and branch better. Reference [12] abstracted system calling sequence and information of context from practicing software over and again, and defined the difference between two different running on information of context as behavior model, VtPath model. It significantly increased accuracy and lowered rates of false alarm compared to N-gram model. Reference [13] proposed a Control Flow Integrity (CFI) model based on function call relations. CFI constructs function call graph by static analyzing function call relations, and abstracted normal relations as expected behavior model. By rewriting binary execution files, CFI added a piece of codes into function calls and returns jump respectively, then checked the real jump if matched the expected. If not, it recognized the jump as abnormal behavior. Reference [4] proposed a software information flow expected model. It marks data coming from outside untrusted and traces the transition of that data. According to its defined security strategy, it monitors trust level assigned by the data to protect untrusted data used for address transition, format string, system calling parameters, etc. Reference [14] also proposed an information flow behavior model which protects software control flow integrity from untrusted data contaminating control data. It ensures software control flow trusted by ensuring the integrity of control data.

Our work essentially brings the idea of taint propagation to construct software behavior model related with untrusted input data. The important difference is that our approach is more flexible and extensible because the list of sources is expressed by concerned instructions rather than by annoation.

3. Model analysis and design

Generally, the existence of the vulnerability does not affect the normal function of the software unless triggered by the malicious user through the elaborately-designed input data. Therefore, when we construct the model of software behavior, it is not necessary to focus on each of the instructions in the instruction set for a program. On the contrary, for reducing overhead we only take into consideration those instructions that either directly or indirectly

process untrusted input data. In order to track tainted user input, we need to specify the following thing:

(1) Sources: A source is a method that returns user input. Generally, all strings emanating from sources must be marked tainted. For reduce the cost of tracking the source, we regard our concerned instruction as the tracking source in this paper. It includes all the instruction which use untrusted input data directly or indirectly.

(2) Propagation: Strings from sources are usually manipulated to form other strings such as queries, or scripts, or file system paths. Strings that are derived from tainted strings also need to be marked tainted. In this paper, we mainly concentrate on the variable assignment behavior related with input data and take the assignment behavior as the propagation action.

(3) Sinks: A sink is a method that consumes input or derivative of user input. This includes methods that execute some form of code, or methods that output data. Tainted strings must be prevented from being used as parameters to sinks.

3.1. Model extracting concerned instruction set

Firstly, we need to determine the taint source by analyzing the instruction set for instructions which have used untrusted input data directly or indirectly. Under our approach, instructions are classified as either taintless instructions or tainted instructions prior to program execution. An instruction is called a tainted instruction if it is supposed to deal with tainted data. Otherwise it is called a taintless instruction. A security alert is raised whenever a taintless instruction encounters tainted data at runtime. Here a data value becomes tainted if it is arithmetically derived or simply copied from tainted data.

Whether an instruction is a taintless instruction or tainted instruction must be determined before program runs. It can be collected either through manual annotation, static analysis or dynamic training. In this paper, static analysis is adopted to acquire the tainted instruction and help to identify taint program variables and those corresponding instructions. The forward slicing technology [15] is adopted to extract our concerned instruction set. Based on the slice criterion $< instruction1, variablen >$, the acquired program slicing is the program subset which is composed of the partial instructions and the control predicate expressions in the program.

Assume that the program ***P*** reads data from untrusted input u_{in}, the instruction set which reads data from u_{in} in ***P*** is $\{s_1, s_2, \ldots, s_i, \ldots, s_n\}$, for any instruction si among them, and variable set which reads data from si notes for Vari=$\{v_1, v_2, \ldots, v_i, \ldots, v_m\}$. The steps of the extraction the concerned instruction set are described as follows:

(1)Calculating program slicing. Make the forward slice analysis to the source file to obtain the instruction set *slicing(si,v)* (not including the dependence of control flow) by for any $v \in Var_i$ by using slice technology and taking $< si, v >$ as the slice criterion.

(2)Calculate $FS(s_i)$:

$$FS(s_i) = slicing(s_i, v_1) \bigcup slicing(s_i, v_2) \bigcup \ldots \bigcup slicing(s_i, v_l) \quad (1)$$

(3)Calculate $FS_{u_{in}}$:

$$FS_{u_{in}} = FS(s_1) \bigcup FS(s_2) \bigcup \ldots \bigcup FS(s_3) \quad (2)$$

$FS_{u_{in}}$ is what we want, the instructions in which are the source to be tracked.

3.2. Process of extracting related variables

If instruction s_1 uses the variable defined in the instruction s_2, and the variable in any path from s_2 to s_1 has not been redefined, it is believed that there is information between them. Therefore, we need to further extract variables which are assigned by instructions in the concerned instruction set.

An abstract syntax tree is one kind of middle expression of the program, and can express grammar structure quite intuitively and contain all static information in the source program. Furthermore, it is very convenient to carry out the traversal and the inquiry to AST. With the help of AST, the analysis process of assigned variables of instructions is summarized as follows:

(1) Extract variable identifier of assignment instruction from the AST(Abstract Syntax Tree) including a pointer variable.

(2) Determine the scope of the variable identifier by traversing the programs AST.

(3) Use points-to analysis to compute the variable set to which pointer variables may point, and translate the pointer dereference into the corresponding variables set.

In addition, according to different types of variables, we adopt different extracting methods. For basic type variables, the variable identifier is extracted. For pointer variables, the variable identifier and the number of the reference of the pointers solution are extracted. For arrays, structure-type variables, the variable identifier as a whole without distinguishing between the arrays elements and structures member variables is extracted.

Accordingly, for the different the types of instruction, the correspondent method of the extracting variable identifier is described as follows:

(1)*Expression* instructions are like *a = b +1; i ++;* etc. After found in AST, assignment variable identifier can be extracted by the structure of AST directly. It is evident that the assigned variable set of branch instructions is empty.

(2)*Do while* instruction and *while* instructions are conditional judgment instructions, so the set of the assigned variable is empty. Variable set assigned by *for* instruction is loop control variable. Variable set assigned by *break*,

continue, *goto* instruction is also an empty set. Similaly, the assigned variable by *return* instruction is empty set or the temporary variable assigned by the compiler.

However, calling a library function needs special treatment. We introduce the explanatory document for a standard library function, in which the output parameters of the library function are marked out. Combining with the variable identifier AST provides and makes use of the instruction of the parameter list, we can obtain the assigned variable identifier of the call library information.

To determine the scope of a variable, from the scope of a function where the instruction begins, and according to the identifier variable, we will traverse in turn from inside to outside until global scope to search the definition of the variable. Within this set, the formal parameters of a function and its locally-defined variables are considered to have function scope.

Pointer analysis [16, 17] is a kind of static analysis technologies. It can calculate the set of storage locations for each pointer variable of source file, including global variables, local variables and the spaces dynamic allocation.

Taking into account that the Andersen algorithm is more balanced in terms of accuracy and efficiency and suitable for analyzing large-scale software, we chose the Andersen algorithm to analyze pointer variables. Because some applications use pointers that can have multiple targets, we need to analyze these multiple-target pointers. Our analysis method is as follows: assuming that program has a pointer variable p, p points to the set of variable $pts(p) = \{p_1, p_2, \ldots, p_i, \ldots, p_n\}$, among them, $p_1, p_2, \ldots, p_i, \ldots, p_n$ are all pointer variables. When the instruction s assigns to $**p$, then the variable set that s can assign is denoted by $pts(p_1) \bigcup pts(p_2) \bigcup \ldots \bigcup pts(p_n)$.

Since tracking the software behavior during execution will impose additional expenses in time and space, we need make some optimization to minimize the time and space overhead in the phase of dynamic tracking and checking. If an instruction only writes into a fixed address, or only writes a fixed number of bytes, or the assigned variable is a temporary variable that the compiler allocates, we consider that the instruction is safe, then our optimized strategy is to remove the following two types of instructions from the concerned set of instructions:

(1)Control instructions. According to the above analysis, the set of variable which a control instruction can assign is empty set or only contains the loop control variable or temporary variable, and the loop control variable is assumed to be of a basic type, either global or local variable, so we can think of these as safe instructions.

(2)Expression instructions which assign variables of basic types directly, such as *a=b+1*, *i++*. They accord with the above safety conditions and dont produce abnormal information flow, so we can consider they are safe instructions.

So far, the concerned instruction set and its variables have been acquired. However, the extracted variable is expressed in the form of (*scope, variable identifier*), which is based on source-level representation, and cannot be directly used for dynamic tracking. Therefore, source-level representation needs to be converted into actual memory addresses. As we know, variables can be divided into global variables, local variables and dynamically allocated heap space. Global and local variables have been allocated addresses at compile time, so the address can be obtained from the debugging information directly. However, heap space data arent allocated by address at compile time, they are assigned address dynamically by functions such as *malloc()*.

In this paper, we mainly concentrate on the variable assignment behavior related with input data and take the assignment behavior as the insecure information flows. Therefore, we regard an instruction as a basic analysis unit and focus on information flow caused by variable assignment. Concretely, we are concerned only with those instructions which write data into memory. We analyze this kind of writing instructions and their corresponding addresses taken from debugging information. First, we locate a group of instructions corresponding to concerned instructions after compilation, then filter out the other types of instructions, retaining only the writing instructions and calls to the standard library, then extracting their address.

4. Runtime Tracking and Checking

During runtime execution, taint checking will monitor the actual behavior of software through obtaining the assignment behavior of concerned statements and extracting its assignment memory address, then it compare them with the corresponding expected statements behavior. It is carried out in the following steps:

(1)Load the collected taintless-instructions profile along with program code into the main memory. These instructions are considered safe.

(2)Tag data from suspicious input channels as tainted. All data derived from input data has been added into our concerned instruction set during statically code analysis.

(3)Track taintedness propagation through execution. It is tracked when a program is executed upon a sink, such as the methods output data.

(4)Raise an alarm when a taintless-instruction encounters some tainted operand.

As a whole, writing instructions and call instructions of library functions are our concerned sink. However, because the address of local variables changes constantly with the call to and exit from a function, it is difficult to determine the sink which is the location or timing of updating and maintaining a local variables address. This space may be dynamically allocated and released continually. We classify the runtime taint sink into the following categories:

(1) Call and exit of function. During software runtime, the expected address of a local variable needs to be updated when function is called or exits.

(2) Allocation and release of heap space. During software runtime, heap space is allocated and released through calling *malloc()* and *free()*.

(3) The loading of dynamic link library. When the libraries are loaded and linked into the process space, their address range of sections is added to address set that can be assigned by library functions. This paper assumes that the program only links the runtime library *glibc* of C.

Then, we need to track the taintedness propagation through execution. We need to check whether the writing instruction accord with their address range, which is described as follows:

For writing instructions, if the actual writing address doesnt belong to any address range of set, we think that it is inconsistent with expectations and raise an alert. During the running process of library functions, all written addresses are verified after returning from library functions. The addresses written by the library functions can be divided into the following categories:

(1)The local variable address of library functions. If writing address is the local variable address of library functions or belongs to the local variable address of function call in its internal implementation, we think that the assignment is legal and doesnt need to be tracked.

(2)The address of the heap area. If the writing address belongs to an allocated address in the heap, we first check whether the address belongs to the allocated heap space of application-defined functions. If not, it indicates that it is the internal heap space of library functions, we believe that the assignment operation is legal, otherwise, we search for in the address set that can be assigned in the sink and verify whether the writing to heap space belongs to the expected address set.

(3)Other Address. If a writing address doesnt belong to the address above, for example, the global variables of process, the local variables of application-defined functions and so on, we scan the address set that can be assigned in the sink and verify whether the address belongs to the expected address set.

5. Analysis and tests

For the sake of simplicity, accuracy as well as effectiveness, this paper utilized the tool ROSE [18] to construct the expected behavior model at first. During dynamic checking, we need to monitor binary codes execution and extract features during software runtime, so we used dynamic Instrumentation tool Dyninst [19,20] to achieve dynamic checking of software behavior.

We tested the expected behavior model by using some test programs with artificial security vulnerabilities including stack buffer overflow, heap buffer overflow and format string attack. Fig.5.1 shows our test program. It treated inputs from stdin as untrusted data.

(1) Stack buffer overflow. In the code fragment shown in Fig.5.1(a), instruction 3 had a vulnerability of stack buffer overflow. If it inputs 18 bytes into *buf* and covers the return address of foo() with the entrance address of *fun()*, the function *foo()* would be unable to return to its correct address after execution.

Stack buffer overflow

```
1. void fun(){return;}
   void foo(){
2.    char buf[10];
3.    scanf("%s",buf);
4.    return;
   }

   int main(){
5.    foo();
6.    return;
7. }
```

(a)

Format String Attack

```
1. int i;
2. void foo(){return;}
   int main(){
3.    char buf[100];
4.    i= 0;
5.    scanf("%s",buf);
6.    printf(buf);
7.    if(i==0)
8.       foo();
9.    return;
   }
```

(b)

Figure.5.1 Two tests

(2) Format string attacks. In the codes show in Fig.5.1(b), instruction 6 had a vulnerability of format strings. The address of global variable *i* is 0x8049604. If we input string \x04\x96\x04\08%d%d%d%d%n into buf via function *scanf()* at instruction 5, then while performing function *printf()* at instruction 6, the value of *i* would be tampered as call instruction 8, and branch instruction 7 would be tampered subsequently.

We tested effectiveness of our approach with two practical examples, which treated remote input as untrusted:

(1) *wu-ftpd* format string vulnerability. *wu-ftpd* is a commonly used server on Linux, and offers basic and simple *ftp* service. In version 2.6.0 or earlier of *wu-ftpd* server, it had the format vulnerabilities on calling function *vsnprintf()*. Attackers could use the vulnerability to get superuser permissions by rewriting user login ID, or to tamper with function return addresses to change program control flow. Our test tampered function return address.

(2) *Openssh* integer overflow vulnerability. *Openssh* is a set of connection tools for safe access to remote computers. Before version of 2.9, *Openssh* had a integer vulnerability in the process of authenticating remote accesses. We used a 32-bit integer to assign a 16-bit integer variable which is the parameter of function *malloc()*. After the variable overflowing by malicious input, attackers can tamper data in any address. Our test used this vulnerability to tamper the value of decision variables of branch instructions in authentication function, and that would allow users attempting a connection to bypass the security authentication mechanism.

Test results showed that our expected behavior model did detect the two security attacks above.

6. Conclusion and future works

The most prevalent attacks on software applications, such as command injection, parameter tampering, cookie poisoning, cross-site scripting, all have the same root cause-improperly validated user input. In this paper, we proposed

an approach for tracking and detecting the improper use of improperly validating user input. We marked data originating from the client input as tainted, and this attribute is propagated throughout the execution of the program. Concretely, we regard an instruction as a basic analysis unit and focus on information flow caused by variable assignment, instructions that use untrusted variable. These behaviors are tracked to determine whether the address modified by the instructions belongs to the specification or not during execution.

In the future we plan to further investigate how to increase accuracy and lower miss-reports for tracking different types of variables, especially the taint source analysis and sink methods capture as well as the taint propagation policy. How to analyze assignment behavior of library functions effectively for library functions is also our concern. In addition, our method is a kind of static approaches which requires the presence of source code. Follow-up research will conduct the approach for the software behavior with executing code and conduct validation experiments on larger data sets.

Acknowledgement

This work is partially supported by Beijing Natural Science Foundation of China under Grant No.4122007.

References

[1] Changxiang Shen, Huanguo Zhang, Huaimin Wang et al. Research and development of Trusted Computing China Science:Information Science,**40**(2):139-166(2010).

[2] A.Nguyen-Tuong, S.Guarnieri, D.Greene, J. Shirley, and D.Evans. Automatically Hardening Web Applications Using Precise Tainting. Proc. of the 20th IFIP International Information Security Conference,295-308 (2005).

[3] G.Venkataramani, I.Doudalis, Y.Solihin, M. Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. Proc. of the 14th Intl Symp. on High Performance Computer Architecture (HPCA). New York: ACM Press,173-184(2008)

[4] J.Newsome. D.Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. Proc. 12th Annual Network and Distributed System Security Symposium, **2**,10-17(2005).

[5] Martin Szydlowski, Christopher Kruegel, Engin Kirda. Secure Input for Web Applications. Proc. of Twenty-Third Annual Computer Security Applications Conference,375-384 (2007)

[6] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. Proc. of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems,85-96(2004)

[7] James Clause, Wanchun Li, Ro Orso Venue. Dytan: A generic dynamic taint analysis framework. Proc. of the International Symposium on Software Testing and Analysis (ISSTA),196-206 (2007)

[8] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, Sy-Yen Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. Proc. of the Thirteenth International World Wide Web Conference, 40-52 (2004).

[9] S. Forrest, S.A.Hofmeyr, A.Somayaji, T.A.Longstaff. A Sense of Self for UNIX Processes. Proc. IEEE Symposium on Security and Privacy. Los Alamitos, CA. IEEE Computer Society Press,120-1289(1996).

[10] W.Li,Y.X.Dai,Y.F.Lian. Context sensitive Host-based IDS using Hybrid Automaton. Journal of Software. **20**(1),138-151(2009)

[11] C.Michael, A.Ghosh. Using Finite Automate to Mine Execution Data for Intrusion Detection: A Preliminary Report.Lecture Notes in Computer Science (1907),66-79(2000).

[12] H.Feng, O.Kolesnikov, P.Fogla, W.Lee, W.Gong. Anomaly Detection Using Call Stack Information. In IEEE Symposium on Security and Privacy, Oakland, California, 62-76(2003).

[13] M.Abadi, M.Budiu, .Erlingsson. Control-flow integrity Principles, Implementations and Applications. Proc. 12th ACM conference on Computer and Communications security, New York, USA, 340-353(2005).

[14] R.Jedidiah. Crandall, T.C.Frederic. Minos: Control Data Attack Prevention Orthogonal to Memory Model. Proc. of the 37th annual IEEE/ACM International Symposium on Microarchitecture,221-232(2004).

[15] S.Horwitz, T.Reps, D.Binkley. Interprocedural slicing using dependence graphs. ACM SIGPLAN Notices,**39**(4): 229-243(2004).

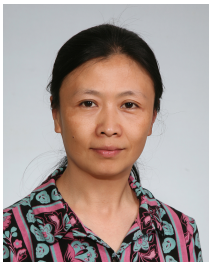
[16] N.Wang,J.Liu. Proficiency and effectiveness comparision of five types pointer analysis algorithm. Computer Engineer and design,**24**(12):38-42(2003).

[17] Shuo Chen, Jun Xu, Nithin Nakka, Abigniew Kalbarczyk, and Ravi Iyer. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. Proc. of IEEE International Conference on Dependable Systems and Networks, 378-387 (2005)

[18] ROSE. https:// www.rosecompiler.org/.

[19] Dyninst . http:// www.dyninst.org.

[20] L.DeRose, T.Hoover, J.K.Hollingsworth. The Dynamic Probe Class Library-an infrastructure for Developing Instrumentation for Performance Tools. Proc. of 15th International Parallel and Distributed Processing Symposium, 66-72(2001).



Dan Wang received the Ph.D degree in computer software from Northeastern University in 2002. She is currently a professor at Beijing University of Technology. Her research interests include software verification, trustworthy software and distributed computing.

Min Dong is a postgraduate candidate at Beijing University of Technology.

Wenbing Zhao received the Ph.D. degree in Signal and Information Process from Peking University in 2004. She is an assistant professor at Beijing University of Technology. Her research interests include data mining, trustworthy software.