A Software Behavior Automaton Model Based on System Call and Context

Zhen Li and Junfeng Tian
College of Mathematics and Computer, Hebei University, Baoding, China
Email: lizhen_hbu@yahoo.com.cn, jftian@hbu.edu.cn

Abstract—**According to the problems of high time overhead of capturing the system call context by walking the stack and inaccuracy of system call argument policies for traditional software behavior models, a software behavior automaton model based on system call and context is proposed.**

First, data flow information containing system call argument policies is combined with software control flow and is used to anomaly detection of software behavior. Second, a new approach of context value for capturing system call context with accuracy and low time overhead is proposed. Third, system call argument context based on system call context is introduced and system call argument policies based on context including system call context and system call argument context are presented. The experimental results show that the software behavior automaton model based on system call and context can capture the system call context accurately with low time overhead, can describe system call argument policies precisely, and can well detect the anomaly of software behavior based on control flow and data flow.

Index Terms—**software behavior, automaton, system call, context, system call argument**

I. INTRODUCTION

With the development of computer and network technology, software plays an important role in the information society. The expansion of software makes software flaws and vulnerabilities difficult to avoid. When the software is under attack, its behavior will deviate from the normal trace. Therefore, anomaly detection by modeling normal behavior of software has become a current research focus in intrusion detection. Most model-based intrusion detection systems monitor the sequence of system calls issued by the software, possibly taking into account some execution state. However, if the system only focuses on the sequence of system calls, that is control flow, without considering data flow information involving system call arguments, attacks such as mimicry attack, non-control-flow hijacking attack and race condition attack can not be detected. In order to detect these attacks, it is necessary to develop techniques that can augment the control flow information captured by existing software behavior models with data flow information pertaining to system call arguments.

A natural approach for gaining system call argument policies is using sets. This approach combines the system call argument policies that exist in any trace. Such a combination can lead to significant losses in accuracy. To provide improved accuracy, it is therefore desirable to partition the argument policy set of each system call into subsets, and gain the argument policies of each subset separately. A good method is to train models according to the execution context of program.

The possible methods for capturing the current calling context are walking the stack or building a calling context tree dynamically. Unfortunately, the time overhead of these methods is unacceptable for most deployed systems. The method of probabilistic calling context (PCC) [1] reduces the time overhead dramatically. However, if the non-commutative function of computing PCC value is used to intrusion detection based on system call, the problem of inaccurate context presentation arises. The paper proposes a new approach for presentation of system call context by context value, and uses the approach to the software behavior automaton model combining system call argument policies based on context with software control flow. The experimental results show that it can capture the system call context accurately with low time overhead and can detect the anomaly of software behavior effectively.

The paper is organized as follows. Section Ⅱ discusses the related work. Section Ⅲ presents the software behavior automation model based on system call and context. Section Ⅳ provides the details on implementation. Section Ⅴ presents the experiments to validate our model's effectiveness and feasibility, followed by concluding our work in Section Ⅵ.

II. RELATED WORK

Because system call is the interface provided by the operating system to access system resources, system call can reflect the features of program behavior to a great extent. A lot of software behavior models based on system call have appeared [2]. Stephanie Forrest et al. [3,

Supported by the National Natural Science Foundation of China (Grant No. 60873203), the Outstanding Youth Foundation of Hebei Province (Grant No. F2010000317), the Natural Science Foundation of Hebei Province (Grant No. F2008000646) and the Science Foundation of Hebei University (Grant No. 2008Q09)

doi:10.4304/jcp.6.5.889-896

4] modeled the behavior of privileged process by short sequences of system calls and constructed the normal behavior feature library for intrusion detection. Inspired by Forrest et al.'s work, many researchers proposed the software behavior models. Wagner et al. [5] proposed program behavior model using static analysis. In their research, a finite state automaton (FSA) or push down automaton (PDA) is built by analyzing the source code of the program. Due to the limited capability of the FSA, function call context is not captured and the method is susceptible to the impossible path problem. Their PDA model captured the precise call and return behavior of function calls due to the stack state maintained in PDA. However, the approach is too computationally expensive to be used in practice. Feng et al. [6] created the VtPath between two system calls by capturing the control flow of program and learning dynamically. Liu et al. [7] proposed an HPDA model, learned from static analysis and supplemented by dynamic learning. Monitoring with the resulting model gave a higher detection capability than with a model acquired by static analysis alone and had a lower false positive rate than with a model acquired by dynamic learning alone. Li et al. [8] also combined static analysis and dynamic learning and proposed a hybrid finite automaton (HFA). Frossi et al. [9] proposed a syscall-based anomaly detection system that incorporated both deterministic model (FSA) and stochastic model (Markov chains). However, the above software behavior models either do not involve calling context or capture calling context by walking the stack with high overhead.

Calling context can provide a rich representation of program location. The simplest method for capturing the current calling context is walking the stack. If the client of calling contexts very rarely needs to know the context, then the high overhead of stack-walking is easily tolerated. However, walking the stack more than infrequently is too expensive for monitoring software. An alternative to walking the stack is to build a calling context tree (CCT) dynamically and to track continuously the program's position in the CCT [10]. Unfortunately, tracking the program's current position in a CCT adds a factor of 2 to 4 to program runtimes. These overheads are unacceptable for most deployed systems. Another method is to sample hot calling contexts to reduce overhead for optimizations [11]. However, sampling is not appropriate for testing, debugging, or checking security violations since these applications need coverage of both hot and cold contexts.

Bond et al. [1] proposed an approach called probabilistic calling context (PCC) for object-oriented programs. PCC computes the context value by evaluating a function at each call site. It uses the following non-commutative but efficiently composable function to compute the next PCC value from the current PCC value: $\mathrm{f}(V, cs) := 3 \times V + cs$, where V is the current value of the calling context, cs is used to identify the current call site and can be computed statically for a call site with a hash of the method and line number. However, there are some problems when PCC is applied to intrusion detection based on system call. System call number can identify a system call, so it can be used as *cs*. For the same system call in two different positions of program, the context value of current system call depends on the value of *V*, that is, last PCC value, because the value of *cs* is fixed. Fig. 1 shows an example. Each circle denotes a state of program execution, and each arrow denotes a system call that executes when state changes from one to another. Because the call stack information of both two *open* system calls in Fig. 1 represented by function names is "f_1, f_2, *open*", the call contexts of two *open* system calls are the same. However, because the PCC value is accumulated generating, the PCC values of two *open* system calls computed by f(*V*,*cs*) are different unless hash collision occurs. Therefore, the method of computing PCC value is not appropriate for system calls in intrusion detection.

According to these problems, the paper proposes a new approach for presentation of system call context by context value, discusses system call argument policies based on context, and combines the argument policies with software control flow based on automaton.

III. Software Behavior Automaton Model Based on System Call and Context

A. Software Behavior Automaton Model

The software behavior automaton model improves HPDA model [7] capturing system call context by walking the stack, introduces a new approach for presentation of system call context by context value, and provides system call argument policies based on context. Fig. 2 shows the source code for an example program and the corresponding software behavior automaton model. The automaton model is a 5-tuple: (S, Σ, T, s, A).

(1) S is a finite set of states.

(2) Σ is a finite set of input symbols. In our model, the set of input symbols is defined as

$$\Sigma = (X \times Addr) \cup (Y \times ContVal) \cup \varepsilon . \quad (1)$$

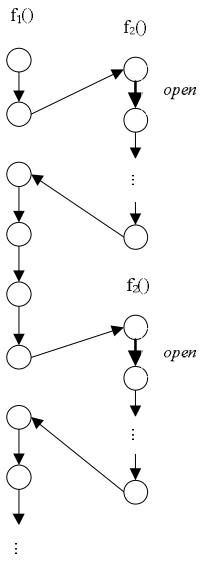

Figure 1. *open* system call in two different positions of program.

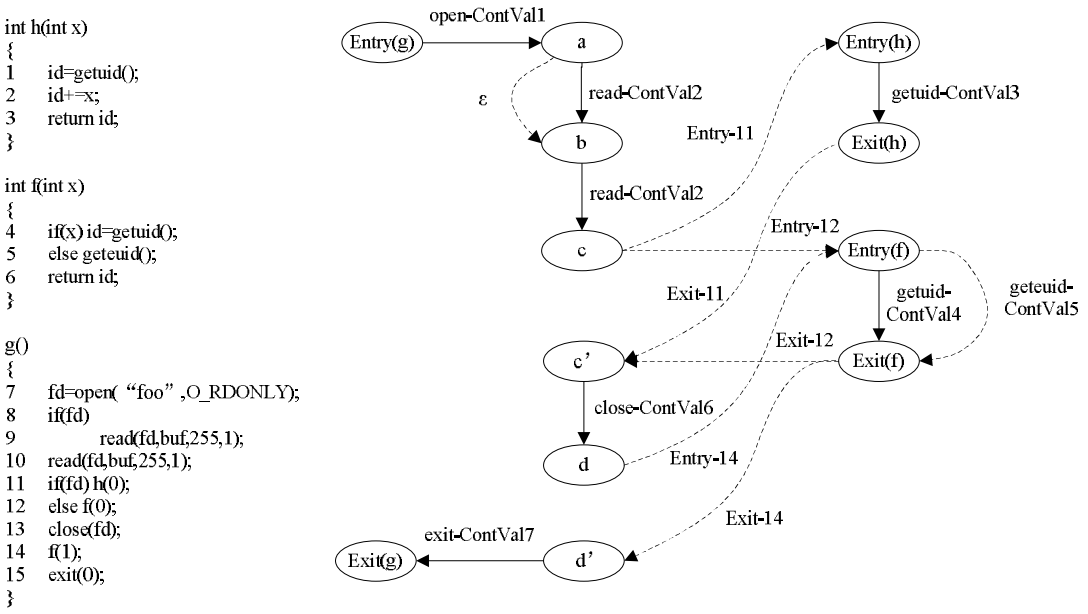

Figure 2. Program code and its corresponding software behavior automaton model.

X={*Entry*, *Exit*}, Y is the set of system calls, *Addr* is the set of all possible addresses defined by the program executable, *ContVal* is the set of context values of system calls. The types of inputs are as follows:

- *Entry* and *Exit* are two new added system calls. *Entry* is added before each function call and *Exit* is added after each function call. *Entry* and *Exit* get the return address before and after the function call respectively.
- Each system call in Y is the invocation point of a system call, where the associated context value represents the context of current system call.
- ε is the empty string.

(3) T is the set of transition functions: $(S \times \sum) \to S$.

(4) s is the start state: $s \in S$.

(5) A is the set of accept states: $A \subseteq S$.

B. System Call Context

System call context is represented by call stack of system call. If the call stacks of two system calls are the same, the system call contexts of two system calls are the same; otherwise, the system call contexts of two system calls are different. The system call context described in Fig. 3 can be expressed as (a_0 , a_1 ,…, a_{m-1} , a_m).

Different system call contexts can be expressed by different context values, and the same context value expresses the same system call context. Context value should have the following properties:

(1) Context values must be basically distributed randomly so that the number of value conflicts is close to the ideal;

(2) The context value must be deterministic, i.e., a given calling context always computes the same value.

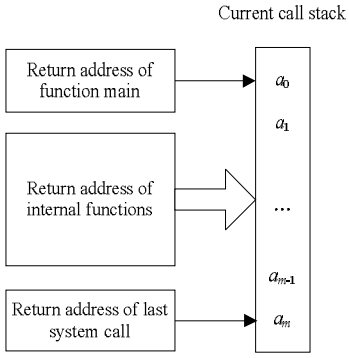

Figure 3. System call context.

Equation (2) is used to determine the number of expected conflicts given population size n and 32- or 64-bit values [12]:

$$E[conflicts] := n - m + m\left(\frac{m-1}{m}\right)^n. \qquad (2)$$

m is the size of the value range (e.g., m=2^{32} for 32-bit values). Bond et al. [1] analyzed the feasibility of calling context representation by context value based on (2). If a program executes 10 million distinct calling contexts, we expect to miss contexts at a rate of just over 0.1%, which is likely good enough for many clients. For programs with many more distinct calling contexts, or for clients that need greater probability guarantees, 64-bit values should suffice. For example, one can expect only a handful of conflicts for as many as 10 billion distinct calling contexts.

Fig. 4 shows the process of gaining system call argument policies based on system call context value.

(1) Form the system call context string.

For system call sc_i, concatenate function names of multilevel functions which call sc_i and system call sc_i with proper separators between two function names or function name and system call, and form a string $\mathrm{SysContStr}_i$ which can represent the current system call context uniquely.

For example, if function f_1 calls function f_2, function f_2 calls function f_3, …, function f_{k-1} calls function f_k, function f_k calls system call sc_i, then the context string $\mathrm{SysContStr}_i$ can be expressed as follows:

$$\mathrm{strcat}(\mathrm{strcat}(...\mathrm{strcat}(\mathrm{strcat}(\mathrm{f}_1,\mathrm{f}_2),\mathrm{f}_3),...,\mathrm{f}_k),\mathrm{sc}_i).$$

Because full context provides too much sensitivity and results in many false positives on real program, especially recursive programs, we limit context to the top h function calls on the stack, that is, $k \leqslant h$.

(2) Compute the hash value $\mathrm{SysContVal}_i$ of the context string $\mathrm{SysContStr}_i$.

(3) Determine the position p_i of system call context in hash table. If the size of hash table is Hashsize, then $\mathrm{p}_i = \mathrm{SysContVal}_i \,\%\, \mathrm{Hashsize}$, where % is the modulus operator.

(4) Gain the argument policies of system call sc_i in context $\mathrm{SysContStr}_i$ from the position p_i of hash table.

It is possible that multiple context strings are mapped into one context value, that is, hash table collision occurs. We use the approach described in [13] to resolve hash table collision problem. There are three different hashes: one for the hash table offset, two for verification. These two verification hashes are used in place of the actual context string. Of course, this leaves the possibility that two different context strings would hash to the same three hashes, but the chances of this happening are, on average, 1:188894659314785808547 84, which should be safe enough for just about anyone. Instead of using a linked list for each entry, when a collision occurs, the entry will be shifted to the next slot, and the process repeats until a free space is found.

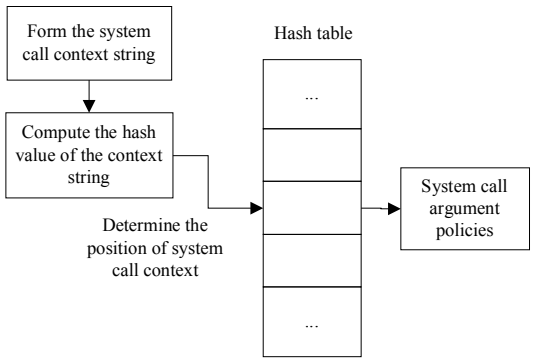

Figure 4. The process of gaining system call argument policies based on system call context value.

C. System Call Argument Policies Based on Context

Let $\mathrm{SC} = \{\mathrm{sc}_1, \mathrm{sc}_2, ...\}$ be the set of system calls, and let $A^{\mathrm{sc}_i} = < a_1^{\mathrm{sc}_i}, ..., a_n^{\mathrm{sc}_i} >$ be the vector of formal arguments for system call sc_i. Each invocation sc_{ij} of sc_i has a concrete vector of values for A^{sc_i} defined as $A^{\mathrm{sc}_{ij}} = < a_1^{\mathrm{sc}_{ij}}, ..., a_n^{\mathrm{sc}_{ij}} >$, and if any of their subvalues $a_l^{sc_{ij}}$ and $a_l^{sc_{ij'}}$ ($1 \le l \le n$) differ, then $A^{sc_{ij}} \neq A^{sc_{ij'}}$. An argument set for a system call sc_i in context c_u, denoted by $\mathrm{AS}(c_u, \mathrm{sc}_i)$, is the set of all argument vectors $A^{sc_{ij}}$ observed for system call sc_i issued in the calling context c_u.

There are mainly two kinds of system call arguments we considered.

(1) Static constant: The value of the system call argument can be determined statically.

(2) Dynamic constant: The value of the system call argument is not a constant but it depends only on values in input files, environment variables, or command line arguments. The value of the system call argument can be decided after the initialization phase and never changes afterward.

The combination of system call argument policies from different functions without system call argument context is shown in Fig. 5(a). The system call contexts of *open* in f() is fixed and has nothing to do with execution paths. As a result, if the argument value allowed only depends on system call context, the context of system call argument is lost. In Fig. 5(a), the argument policies of *open* coming from $\mathrm{f}_1()$ and $\mathrm{f}_2()$ combine in f(). The direct combination of *open* argument policies from different paths results in the loss of relations between argument policies and execution paths. For a system call, attackers can apply the argument policies of one execution path to the other without detection. Therefore, both system call context and system call argument context should be considered for the combination of system call argument policies from different functions.

We make a data-flow analysis to identify system call argument policies. The analysis recovers argument policies using a finite-height lattice of values. $\mathrm{AP}(c_u, \mathrm{sc}_i)$ is the set of argument policies of system call sc_i in system call context c_u. P is the power set lattice of $\mathrm{AP}(c_u, \mathrm{sc}_i)$. Lattice values are elements of $D_p = P(\mathrm{AP}(c_u, \mathrm{sc}_i))$ with $\perp_P = \Phi$ and $\top_P = \mathrm{AP}(c_u, \mathrm{sc}_i)$. The join operator of lattice A and B of system call argument policies from different execution path in system call context c_u is expressed as follows:

$$A \sqcup_P B = A \bigcup B. \tag{3}$$

In order to resolve the problem of the loss of relations between argument policies and execution paths, context

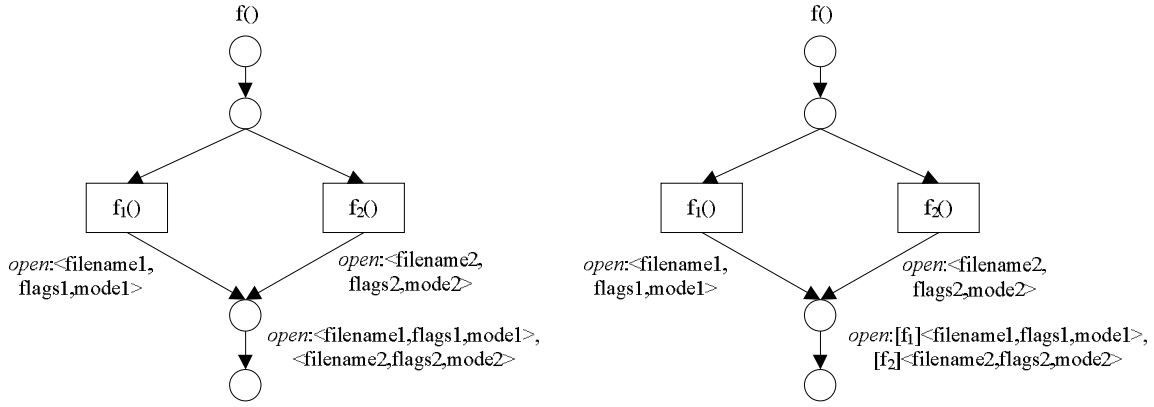

Figure 5. Combination of system call argument policies from different functions.

information should be added to system call argument policies.

Let $\mathrm{AC}=\{\mathrm{ac}_1,...,\mathrm{ac}_n\}$ be the set of system call argument contexts and Q be the power set lattice of $\mathrm{AP}(c_u,\mathrm{sc}_i)$ in system call argument context. Domain $D_Q=P(D_P\times\mathrm{AC})$, $\perp_Q=\{\perp_P,\Phi\}$, $\top_Q=\bigcup_{i=0}^{n}\{(\top_P,\mathrm{ac}_i)\}$.

Let $A,B\in D_Q$ be $A=\{(A_v,\mathrm{ac}_v)\}$ and $B=\{(B_w,\mathrm{ac}_w)\}$, where $\forall v(1\le v\le n)$: $A_v\in D_P$, $\mathrm{ac}_v\in\mathrm{AC}$; $\forall w(1\le w\le n)$: $B_w\in D_P$, $\mathrm{ac}_w\in\mathrm{AC}$.

The join operator of lattice A and B after introducing system call argument context can be expressed as follows:

$$\begin{aligned}A\sqcup_Q B=&\{(A_v\sqcup_P B_w,\mathrm{ac}_v)\mid \mathrm{ac}_v=\mathrm{ac}_w\}\\&\bigcup\{(A_v,\mathrm{ac}_v)\mid(\forall w)\mathrm{ac}_v\neq\mathrm{ac}_w\}\\&\bigcup\{(B_w,\mathrm{ac}_w)\mid(\forall v)\mathrm{ac}_v\neq\mathrm{ac}_w\}.\end{aligned}\quad(4)$$

Equation (4) involves three cases:

- The system call argument context ac_v of A_v equals to the system call argument context ac_w of B_w;
- The system call argument context ac_v of A_v is different from any system call argument context ac_w of B_w;
- The system call argument context ac_w of B_w is different from any system call argument context ac_v of A_v.

After introducing system call argument context, the relations between argument policies and execution paths can be saved for the combination of system call argument policies from different functions. As shown in Fig. 5(b), owing to considering system call context, the system call argument context can be expressed by marking with function name that argument policies come from.

We limit our attention to unary and binary relations on system call argument policies based on context including system call context c_u and system call argument context ac_v.

(1) Unary relations

Unary relations capture properties of a single argument. They can be represented using the form $A^{sc_{ij}}\in\mathrm{AS}(c_u,ac_v,sc_i)$. Such a relationship can be learnt in every trace. The set $\mathrm{AS}(c_u,ac_v,sc_i)$ is represented using an enumeration by listing all $A^{sc_{ij}}$ in system call context c_u and system call argument context ac_v. The system call arguments that don't make any sense should be ignored, for example, integer arguments of file descriptors or memory addresses.

(2) Binary relations

Binary relations capture relationships between two system call arguments or between one system call arguments and another system call return values.

① Equal relations. For example, the *open* system call is in system call context c_{u1} and system call argument context ac_{v1} and a subsequent *write* system call is in system call context c_{u2} and system call argument context ac_{v2}. The file descriptor fd_1 returned by the *open* system call is equal to the operand fd_2 of the *write* system call. The relation can be expressed as follows:

$$\mathrm{fd}_1(c_{u1},\mathrm{ac}_{v1},open)\quad\text{equal}\quad\mathrm{fd}_2(c_{u2},\mathrm{ac}_{u2},write).\quad(5)$$

② String relations. String relations include prefix, suffix, substring, etc. For example, string R of system call sc_i in system call context c_{u1} and system call argument context ac_{v1} is the prefix of string T of system call sc_j in system call context c_{u2} and system call argument context ac_{v2}. The relation can be expressed as follows:

$$R(c_{u1},\mathrm{ac}_{v1},\mathrm{sc}_i)=\mathrm{prefix}(T(c_{u2},\mathrm{ac}_{v2},\mathrm{sc}_j)).\quad(6)$$

Ⅳ. IMPLEMENTATION

We have implemented our model in Linux. The generation of software behavior automaton based on system call and context adopts the similar method of HPDA [7]. The differences between our model and HPDA are as follows:

(1) For system call sc_i, the context of sc_i should be computed and recorded;

(2) For two new system calls *Entry* and *Exit*, the return address before and after the function call should be recorded respectively.

Each system call is intercepted by loadable kernel module (LKM) and modified. We take *open* system call as an example. Fig. 6 shows *open* system call after loading kernel module. When *open* system call is called, first of all, decide whether the system call in the corresponding position of automaton is *open* or not. If not, report that the software behavior is abnormal; otherwise, detect data flow information. Gain *open* system call context in automaton, and determine the position of *open* system call context in hash table. Then decide whether the arguments of *open* system call intercepted are in accord with argument policies of *open* system call in the above context. If not, report that the software behavior is abnormal; otherwise, invoke the original *open* system call.

Ⅴ. EXPERIMENTS AND ANALYSIS

We have made experiments on a PC with AMD Phenom(tm) 8400 Triple-Core Processor 2.10GHz and 2GB of main memory running Linux kernel 2.4.20.

A. Accuracy of Capturing System Call Context

We have made experiments on four common programs (gzip, cat, find and tar) in Linux according to three methods of capturing system call context that is call stack, PCC and our method. Among them, call stack method adopts __builtin_return_address() function to gain the return address of function in system call stack; PCC method adopts system call number of current system call as *cs*; Our method limits context to the top five function calls on the stack, that is, h=5 and the size of hash table in PCC and our method is 2^{20}. The statistical results are shown in Table Ⅰ. When two or more different calling contexts map to the same value, the conflict occurs.

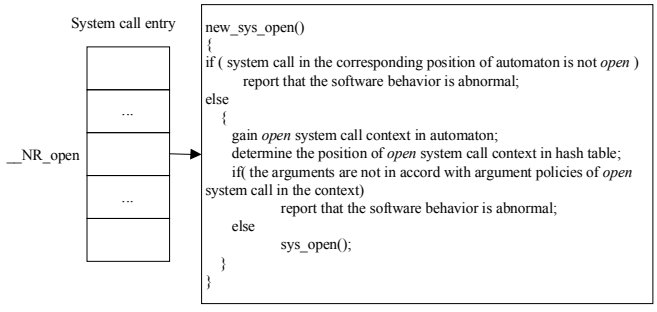

Figure 6. The process of *open* system call after loading kernel module.

When there are k different calling contexts mapping to the same value, the conflict number is k-1.

From Table Ⅰ, we can see that for most programs, the number of system call context is more than that of different system calls, that is, a system call can be called in many different system call contexts. The representation of system call context in Section Ⅲ shows that it is accurate to capture system call context by call stack, which can be the measure of accuracy of capturing context. The experimental results show that there is large difference between the number of system call context captured by PCC and the number captured by call stack. The number of system call context captured by PCC is nearly equal to the total number of system calls and the difference between them is just the conflict number. The reason of this has been described in Fig. 1 of Section Ⅱ. The number of system call context captured by our method is equal to the number captured by call stack, which means our method can capture system call contexts accurately. There are some conflicts in PCC because the number of system call contexts is a little large, and there is no conflict in our method owing to a small number of contexts.

B. Ability of Anomaly Detection

We have test several attacks, and the experimental results are shown in Table Ⅱ. Our model can detect these attacks successfully by control flow expressed as automaton or system call argument policies based on context.

(1) Attacks against control flow

Take *kon*2 software for example. *kon*2 is a Kanji emulator for the console. It is a setuid root application program. For version 0.3.9b in Red Hat 9, there is buffer overflow exploit in function ConfigCoding() when using the –Coding command line parameter. This vulnerability, if appropriately exploited, can lead to local users being able to gain root privileges [14].

Our attack is to make the return address of function ConfigCoding() in *vt.c* file overflow. The faked return address redirects to a piece of shellcode. The attacking code generates a shell.

With our model, when the software requests *execve*("/bin/sh"), the control flow of the software behavior deviates from the normal and the software behavior is detected anomaly successfully.

(2) Attacks against data flow

User identity data attacks and directory traversal attacks are described in [15], including attack for Site Exec command string of WU-FTPD [16] and attack for stack overflow of GHTTP [17]. For attacks on file descriptors, *vi* 6.1 in Red Hat 9 Linux exists potential TOCTTOU (Time of Check to Time of Use) vulnerabilities [18]. Let's take the software *vi* 6.1 for example.

Red Hat 9 Linux distribution includes *vi* 6.1 which exists potential TOCTTOU vulnerabilities. Specifically, if *vi* 6.1 is run by root to edit a file owned by a normal

TABLE I.
STATISTICAL RESULTS OF SYSTEM CALL CONTEXT

Program	Workload	Total number of system calls	Number of different system calls	Number of system call context			Conflict number	
				Call stack	PCC	Our method	PCC	Our method
gzip	Compress a 4.2MB file	309	17	44	307	44	2	0
cat	Display the content of a 160KB file	184	11	33	183	33	1	0
find	Find a file in directory including 1980 files	1537	17	64	1533	64	4	0
tar	Pack 200 files to a 13.6MB tar file	4088	19	83	4078	83	10	0

user, then the normal user may become the owner of sensitive files such as */etc/passwd*. The vulnerability relates to the vulnerability window <*open*, *chown32*> in *fileio.c* file. Fig. 7 shows the related source code. The sequence of system calls that the vulnerability window <*open*, *chown32*> is related to is as follows:

…, *open*, *write*, *close*, *chmod*, *stat64*, *chown32*, *chmod*, ...

Our attack consists of a tight loop constantly checking whether the owner of the *wfname* file has become root. Once this happens, the attacker replaces the file with a symbolic link to */etc/passwd*. When *vi* 6.1 exits, it should change the ownership of */etc/passwd* to the attacker. The attack will succeed only when *vi* 6.1 relinquishes CPU voluntarily or involuntarily and the attacker is scheduled to run, and the attacker process finishes the file redirection during this run. If *vi* 6.1 finishes and */etc/passwd* is still owned by root, the attack fails.

If the attack succeeds, when *vi* 6.1 runs to *chown32* system call, the abnormal can be detected by our model because the *wfname* argument of *chown32* system call is not equal to the *wfname* argument of *open* system call, which is not satisfied with the argument policies of *chown32* system call.

C. Performance

1. Time overhead

We test the *open* system call in four common programs (gzip, cat, find and tar). Fig. 8 shows the average time of capturing *open* system call context by call stack, PCC and our method. The time of capturing system call context by call stack is the longest, while the time by PCC is the shortest. The time of former is about 2.5~10 times longer than that of latter. The time of capturing system call context is different for different number of *open* system calls and realized functions in different programs. The time capturing system call context by our method is between call stack and PCC, and is closer to PCC. Although the time of PCC method is the shortest, we can see from Table Ⅰ that capturing system call context by PCC is inaccuracy. Our method can capture the system call context accurately with low time overhead.

```
while((fd=mch_open((char *)wfname,...)
......
chown((char *)wfname, st_old.st_uid, st_old.st_gid);
```

Figure 7. The vulnerability of *vi* 6.1(*fileio.c*).

TABLE II.
SEVERAL ATTACKS AND THEIR DETECTION METHODS

Attack type	Attacks	Detection method
Control flow	*kon*2 0.3.9b buffer overflow	Detect by control flow expressed as automaton.
Data flow	User identity data attacks	Detect by equal relations of system call argument policies based on context.
	Directory traversal attacks	Detect by unary relations of system call argument policies based on context.
	vi 6.1 TOCTTOU attacks	Detect by equal relations of system call argument policies based on context.

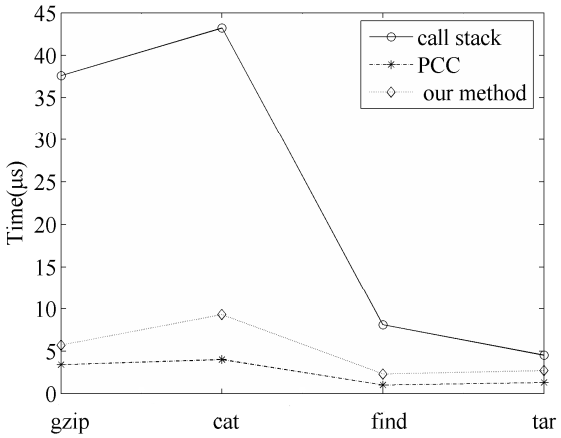

Figure 8. Average time of capturing *open* system call context.

2. Space overhead

Space overhead mainly reflects on automaton storage and hash table storage. The software behavior automaton model based on system call and context uses adjacency list to store data, which space complexity is $O(n+e)$. n is the number of states in automaton, e is the number of edges, that is, the number of system call, *Entry* and *Exit*.

Hash table is a method that uses storage for time. It is suitable for enough memory that can be provided. Our experiments use a fixed-size hash table with

2^{20}=1,048,576 slots (4MB), but real clients would use space that is proportional to their needs. The hash table approach is efficient as long as the table size is roughly at least twice the size of the number of entries in the table, or table conflicts will lead to high overhead.

VI. Conclusion

The paper proposes a new approach for presentation of system call context by context value, builds a software behavior automaton model based on system call and context, and applies the system call argument policies based on context to attack detection. The experimental results show that our model can capture the system call context accurately with low time overhead and can well detect attacks based on system calls. Our approach of capturing context can be applied to any anomaly detection system to capture system call context. However, our model still has some limitations in representation of context. It resolves the problem of system call context and system call argument context from different functions, but can not identify system call argument context from different branches in the same function, which requires further research.

References

[1] M. D. Bond and K. S. McKinley, "Probabilistic calling context," *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, Montreal, Canada, October 2007, pp. 97–112, doi:10.1145/1297027.1297035.

[2] F. Tao, Z. Y. Yin, and J. M. Fu, "Software behavior model based on system calls," *Computer Science*, vol. 37, pp. 151–157, 2010. (in Chinese)

[3] S. Forrest, S. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes", *Proceedings of IEEE Symp. Security and Privacy*, IEEE Computer Society, Washington DC, USA, 1996, pp. 120–128, doi:10.1109/SECPRI.1996.502675.

[4] S. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, pp. 151–180, 1998.

[5] D. Wagner and D. Dean, "Intrusion detection via static analysis," *Proceedings of IEEE Symp. Security and Privacy*, IEEE Computer Society, Washington DC, USA, 2001, pp. 156–169, doi:10.1109/SECPRI.2001.924296.

[6] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," *Proceedings of IEEE Symp. Security and Privacy*, IEEE Computer Society, Washington DC, USA, 2003, pp. 62–75, doi:10.1109/SECPRI.2003.1199328.

[7] Z. Liu, S. M. Bridges, and R. B. Vaughn, "Combining static analysis and dynamic learning to build accurate intrusion detection models," *Proceedings of the 3rd IEEE International Workshop on Information Assurance*, College Park, MD, USA, March 2005, pp. 164–177, doi:10.1109/IWIA.2005.6.

[8] W. Li, Y. X. Dai, Y. F. Lian, and P. H. Feng, "Context sensitive host-Based IDS using hybrid automaton," *Journal of Software*, vol. 20, pp. 138–151, 2009. (in Chinese)

[9] A. Frossi, F. Maggi, G. L. Rizzo, and S. Zanero, "Selecting and improving system call models for anomaly detection," *Proceedings of the 6th Detection of Intrusions and Malware, and Vulnerability Assessment*, Como, Italy, 2009, pp. 206–223, doi:10.1007/978-3-642-02918-9_13.

[10] J. M. Spivey, "Fast, accurate call graph profiling," *Software-Practice and Experience*, vol. 34, pp. 249–264, 2004.

[11] X. Zhuang, M. J. Serrano, H. W. Cain, and J. D. Choi, "Accurate, efficient, and adaptive calling context profiling," *Proceedings of ACM Conference on Programming Language Design and Implementation*, Ottawa, Ontario, Canada, 2006, pp. 263–271, doi:10.1145/1133981.1134012.

[12] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. New York: Cambridge University Press, 2005.

[13] J. Olbrantz, Inside MoPaQ, http://shadowflare.samods.org/inside_mopaq/chapter2.htm #hashes, 2002.

[14] Red Hat Security: Updated kon2 packages fix buffer overflow, http://rhn.redhat.com/errata/RHSA-2003-047.html, 2003.

[15] S. Chen, J. Xu, E. C.Sezer, P. Gauriar, and R. K. Iyer. "Non-control-data attacks are realistic threats," *Proceedings of the 14th Conference on USENIX Security Symposium*, Baltimore, MD, 2005, pp. 177–192, doi:10.1.1.113.5408.

[16] Ghttpd Log() Function Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/5960:SecurityFocus, 2002.

[17] H. Chen, D. Dean, and D. Wagner, "Model checking one million lines of C code," *Proceedings of Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2004, pp. 171–185, doi:10.1.1.3.1535.

[18] J. P. Wei and C. Pu, "TOCTTOU vulnerabilities in UNIX-style file systems: an anatomical study," *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, Dec. 2005, pp. 155–167.

Zhen Li, born in Baoding, China, 1981, received M.S. degree of computer application from Hebei University, Baoding, China in 2006.

She is a lecturer of the College of Mathematics and Computer of Hebei University. In the past five years, she has published over 10 technical papers in refereed journals and conference proceedings. Her current research interests include computer security, trust computing and distributed computing.

Junfeng Tian, born in Baoding, China, 1965, received Ph.D degree of computer science from University of Science and Technology of China, Hefei, China in 2004.

He is a professor of Computer Science at Hebei University. In the past five years, he has published over 30 technical papers in refereed journals and conference proceedings. His research interests include network security, trust computing, distributed computing and wireless sensor network.

More information about Prof. Tian can be found at http://int.hbu.edu.cn/int/szll/tjf.html.