Intrusion Detection System Based on Binary Code and Execution Stack Analysis

†Mizuki OKA †Hirotake ABE †Yoshihiro OYAMA †Kazuhiko KATO
†Master’s Program in Science and Engineering, University of Tsukuba
†Doctoral Program in Engineering, University of Tsukuba
†Department of Computer Science, Graduate School of Information Science and Technology, University of Tokyo
†Institute of Information Sciences and Electronics, University of Tsukuba. JST PRESTO.

A malicious user can manipulate control flow of a program by abusing the program’s vulnerabilities. We have implemented a system that dynamically detects such malicious activities. We extract program’s control flow and construct an automaton model. The model is created by statically analyzing the binary code of the program. This model contains all the possible system call sequences of the program. At runtime the system dynamically extracts information from execution stack and checks the validity of the program by tracing the constructed model using the execution stack-state. We also extract valid transition paths from the constructed model and use the information when checking the validity of the path to reduce the runtime check overhead. Moreover, our system uses search history of transition paths to reduce the overhead further. We measure the overhead of the system.

1 Introduction

An intrusion detection system (IDS) is a system that identifies misuse and intrusions. Misuse is used to describe an attack from internal network and intrusion is used to describe attacks from outside. Typically there are two types of IDS. The first type is network-based IDS. They are set in place on network to detect attacks. The second type is host-based IDS. They are set on a host to inspect the behavior of that host. The main approaches to intrusion detection are anomaly-based detection and signature-based detection. The anomaly-based detection models normal behavior of the systems and use the model to detect anomalous activity. Any system activity that differs from the model is considered anomalous. The signature-based detection uses known anomalous signatures of the attacks and use them to detect anomalous activity by pattern matching. In this paper we consider host-based IDS and anomaly-based detection.

There has been a lot of research on detecting anomalous system activity by examining the behavior of programs. Some of the proposed methods create models based on system call traces [1, 2, 3, 4, 5] The information of the system call traces is useful but not enough to determine the current status of the program.

In this paper, we propose a method to use not only system call traces but also return address information extracted from the execution stack. The execution stack provides valuable information to determine the current status and history of program execution. Consequently using the execution stack information helps to reduce runtime overhead while detecting more attacks. Our method first extracts program control flow and construct an automaton model. The model is created by statically analyzing the binary code of the program. This model contains all the possible system call sequences of the program. It then performs anomaly detection by using the execution stack and sequences of system calls. Our system performs intrusion detection by verifying whether the transition paths in the execution stack are included in the generated model. To increase the performance of the system, we save the search history of transition paths and use them to reduce the runtime overhead. Our method is an extension of the method of Wagner et al. [1].

The rest of the paper is organized as follows. Section 2 describes existing IDS. Section 3 presents our proposed method. Section 4 presents experimental results. Section 5 describes the related research. Section 6 summarizes the paper and discusses future work.

2 Existing IDS

Wagner et al. proposed an IDS based on static analysis [1]. Their method generates models from source code for determining whether executions are normal or abnormal. One of the advantages of their method is that it can detect unknown attacks while generating no false positives. Their method statically generates a pushdown automaton models that represents the global control-flow of a program. This automaton is non-deterministic because it is not possi-
ble to predict statically which branch of choices will be taken. A context-free grammar (CFG) that expresses system call sequences is then created. The CFG is used as the model of the program. The system traces transition paths of the program in the model each time a system call is issued. If none of the non-deterministic paths are included in the model, then it is considered as abnormal. One of the problems in their method is that the runtime overhead is too large. For example, [1] states that it takes more than one hour to execute sendmail per transaction. The non-determinism of the model may be a cause of the large runtime overhead.

3 Proposed Method

Our method solves the problem described in Section 2 by specifying the control point more deterministically using the execution stack information. Unlike in Wagner et al. [1] our method first generates a model from binary code. Using binary code enables to use our system for programs whose source codes are not available. Our system then extracts return addresses from the execution stack each time a system call is issued and generates abstract execution path between two program execution points. The two points of the abstract path are the previous execution stack extracted when the previous system call is issued and the current execution stack extracted when the current system call is issued. Our method traces the transition of the abstract path in the model and verifies whether such transition path exists in the model. If such a path does not exist then we consider the transition as abnormal.

3.1 Checking Transitions Using Execution Stack

Anomaly detection progresses in five stages:

1. As each system call is issued, we extract the execution stack information (current execution stack), and obtain the sequence of return addresses. We call the sequence list $A = \{a_0, a_1, a_2, \ldots, a_{n-1}\}$, where $n$ is the number of frames in the execution stack and $a_{n-1}$ is the return address of the function last called. The program counter (PC) that points to the address of system call is added as $a_n$ in the list $A$.

2. Let list $B = \{b_0, b_1, b_2, \ldots, b_{m-1}\}$ the previously extracted execution stack (previous execution stack), where $m$ is the number of frames in the stack and $b_{m-1}$ is the return address of the function last called. The PC that points to the address of system call is added as $b_m$ in the list $B$. We compare the list $A$ with the list $B$ from the bottom frames until we find $k$ so that $a_{k+1} \neq b_{k+1}$. We call the frame common frame as shown in Figure 3.

3. In list $B$, for all $b_{k+1}, \ldots, b_{m-1}$, go to the function call corresponding to the return address $b_{i-1}$ in the generated model, where $i$ ranges from $m - 1$ to $k + 1$. Check if there is a transition path from the function corresponding to the return address $b_i$ to an exit (Exit) of the function without calling any system calls. If such a path does not exist then the transition is considered as abnormal.
(4) Go to the function corresponding to the return address $a_k$ or $b_k$ (common frame) in the generated model. Check if there exists a transition path from the function call corresponding to the return address $a_{k+1}$ to the function call corresponding to the return address $b_{k+1}$ without calling any system calls. If such a path does not exist then it is considered as abnormal.

(5) In list $A$, for all $\{a_{k+1}, ..., a_{n-1}\}$ go to the function corresponding to the return address $a_i$ in the generated model, where $i$ ranges from $k+1$ to $n-1$. Check whether there is a transition path from the beginning ($Entry$) of the function to the function call corresponding to the return address $a_{i+1}$. If such a path does not exist then the transition is considered as abnormal.

Figure 3: Execution Stacks A and B extracted at the current system call $a_n$ and the previous system call $b_m$ respectively. Our system checks transition paths (3), (4), and (5) in the model.

### 3.2 Reutilization of Search History

Collaterally we obtain the following results during the process of the algorithm described in Section 3.1

For existent two endpoint nodes $a$ and $b$ in an automaton of a function, there is at least one transition path from $a$ to $b$ without calling any system calls.

Using the obtained results to verify transition paths reduces the runtime overhead. Our system saves the search history of transition paths during the runtime so that we can abbreviate to search the same transition paths that were previously traversed. The search history is represented in a form of matrix with each matrix corresponding to the each model of the function. Let $n$ be the number of nodes in a function. The search history of the function is represented as the square matrix of $n\times n$. Figure 4 shows an example of search history in a matrix, where $T$ denotes that there is a transition path from the node $do\_copy$ to the node $Exit$ without calling any system calls. Once the monitored program stopped to run, all the search history obtained during the execution is saved to a file. The search history is valid as long as the monitored binary code and the model stay the same. For this reason, it is possible to omit some searches of transition path from the first run of the program by loading the history search file. The more the monitored program is executed, the more search history will be accumulated. In the ideal case, the runtime overhead will only be the time to refer to the search history in the matrix.

Figure 4: An example of search history

### 4 Experimental Results

We have implemented our method and measured the runtime overhead. The purpose of the experiment is to measure the difference of runtime overhead of the program with and without the IDS. We also measure how much the runtime overhead is reduced by the reutilization of the search history. The programs that we used are $wc$, $cp$ and $tar$ of GNU. We counted size, number of lines and words of the file of about 3.8MB with the command $wc$. We copied the file of 13MB with the command $cp$. 
With the `tar` command we compressed file with the total size of about 66MB and created a tar file. Figure 5 indicates the results of the execution runtime of normal commands and that of with IDS. In the case of the runtime with IDS, we measured the runtime of with and without the use of search history. In the case of runtime with IDS and search history we have measured the runtime of with and without the use of the matrix of transition paths obtained from the previous execution of the program. We can see from the results that using the search history greatly reduces the runtime overhead. The runtime overhead ranges from the ratio of 2.7 to 16.0 compared to the normal command when there is no use of search history. The difference of runtime overhead is caused by the different size of functions that the program have. The larger the function is, the more time is required to find the transition path. The runtime overhead is reduced to the ratio of between 2.0 and 7.0. This is because the search history eliminates the overlapping search in the process of execution. Using the matrix of transition paths reduces the overhead to the ratio of less than 2.0. In the experiment with the use of matrix, we executed the same program with the same data therefore there was no need of searching the transition paths in the model.

### 5 Related Work

Recently, Feng et al. proposed a method, called VtPath[3], to perform anomaly detection using execution stack. Their basic idea is very close to our idea. VtPath also uses the return addresses of the execution stacks to monitor the program execution on-line. However VtPath generates a control flow model by learning program behavior during many program runs whereas our method generates the model statically from binary code. Constructing the model of the program behavior by learning method does not guarantee that every behavior of the program is included in the model. It can therefore cause false positives and/or false negatives. As our method generates no false positives and few false negatives, we believe that our method can detect attacks that VtPath might miss.

### 6 Summary and Future Work

We have proposed an approach to use return address information extracted from the execution stack. Its main advantages are that it is more deterministic than the method of Wagner et al. Our method specifies which function boundaries a transition traverses and consequently reduces the number of transition paths to trace. Moreover, we have introduced the reutilization of the search history to reduce the overhead of transition paths during the execution. In our experiments we have shown that accumulation of search history plays an effective role in reducing the runtime overhead.

As future work, we will perform more experiments to test the capabilities of our system and compare with other systems. Models constructed by using system calls represent only one aspect of program behavior. We would like to continue seeking other parameters that can be used to construct models.

### References


