Bridging the Gap between Data-flow and Control-flow Analysis for Anomaly Detection*

Peng Li
University of North Carolina at Chapel Hill
pengli@email.unc.edu

Hyundo Park
Korea University, Seoul, Korea
hyundo95@korea.ac.kr

Debin Gao
Singapore Management University, Singapore
dbgao@smu.edu.sg

Jianming Fu
Wuhan University, Wuhan, China
jmfu@whu.edu.cn

Abstract

Host-based anomaly detectors monitor the control-flow and data-flow behavior of system calls to detect intrusions. Control-flow-based detectors monitor the sequence of system calls, while data-flow-based detectors monitor the data propagation among arguments of system calls. Besides pointing out that data-flow-based detectors can be layered on top of control-flow-based ones (or vice versa) to improve accuracy, there is a large gap between the two research directions in that research along one direction had been fairly isolated and had not made good use of results from the other direction.

In this paper, we show how data-flow analysis can leverage results from control-flow analysis to learn more accurate and useful rules for anomaly detection. Our results show that the proposed control-flow-analysis-aided data-flow analysis reveals some accurate and useful rules that cannot be learned in prior data-flow analysis techniques. These relations among system call arguments and return values are useful in detecting many real attacks. A trace-driven evaluation shows that the proposed technique enjoys low false-alarm rates and overhead when implemented on a production server.

1 Introduction

Many host-based anomaly detectors have been proposed to monitor system calls. Some of these detectors [5–8, 13, 17, 19, 20] monitor the sequence of system calls emitted by the application and utilize control-flow information of the system calls for intrusion detection. Control-flow-based detectors have been shown to be effective in detecting intrusions, e.g., code-injection attacks, because many such intrusions change the control flow of the program to make additional system calls.

Some detectors [2, 11, 15, 16], on the other hand, monitor the arguments of system calls and use the data-flow information for intrusion detection. Monitoring system call arguments has the advantage of detecting more stealthy attacks that do not change the control flow of the program but merely change system call arguments. Despite the success of many data-flow techniques, research has not been done on how data-flow analysis can leverage the results from control-flow analysis to detect intrusion. Bhatkar [2] pointed out that making use of control-flow context can help learning data-flow properties, but all it does was to make use of the program counter information to differentiate instructions at different locations in a program. To show how control-flow information could give further help in data-flow analysis, please refer to a simple example in Figure 1.

```
fd = open(dir, "r");
read(fd, buf, size);
```
Block A

```
fd1 = open(dir1, "r");
fd2 = open(dir2, "w");
read(fd1, buf, size);
write(fd2, buf, size);
```
Block B

```
fd1 = open(dir1, "r");
if (need_to_write)
 fd2 = open(dir2, "w");
read(fd1, buf, size);
if (need_to_write)
 write(fd2, buf, size);
```
Block C

Figure 1. Control-flow information helps data-flow analysis

*This research was mostly done when the first two authors, Peng Li and Hyundo Park, were researchers working in Singapore Management University. The project was partially supported by NSF China under the agreement 90718005, and by MIC Korea under the ITRC support program supervised by IITA (IITA-2008-(C1090-0801-0016)) and the IT R&D program of MKE/IITA (2008-S-026-01).

1063-9527/08 $25.00 © 2008 IEEE
DOI 10.1109/ACSAC.2008.17

Figure 1 shows a simple example with three blocks of source code. When executing Block A, the first argument of system call `read` always equals to the return value of its immediate preceding system call `open`. This is a very nice and useful rule that most of the existing techniques can learn. However, when Block B executes, the same rule does not apply, as the first argument of `read` now equals to its *second* preceding `open`. If both Block A and Block B are in the program to be monitored, training will be confused as the rules are valid with low probabilities.

A simple solution is to combine the two rules to be a single one, i.e., the first argument of `read` equals to either the first or the second preceding `open`. However, this results in a less precise rule and gives attackers more room to get evaded. Another solution is to use the different program counter values for the instructions in Block A and Block B to differentiate the two `read` system calls, so that a different rule can be used for each block. However, this simple solution will not work in cases where program counter values cannot differentiate the two cases, e.g., in Block C.

The example shown in Figure 1 motivates the idea that control-flow information is very important in learning data-flow relations in system call arguments and return values. In general, the same system call may have very different data-flow properties when used in different context, and this context information may not be available by simply examining the program counter values. Therefore, we need a better way of making use of control-flow information in order to perform data-flow analysis with improved accuracy.

In this paper, we introduce the first technique in leveraging results from control-flow analysis for the purpose of data-flow analysis for intrusion detection. In short, control-flow analysis helps putting each system call into the context of performing some individual task. We then learn data-flow relations among the arguments and return values of the system calls based on the different context in which the system calls are made. We design three *Rule Sets* to capture these relations: Rule Set A contains rules that reveal the argument and return value relations when the process being monitored is performing a particular task; Rule Set B exploits rules that govern the system calls when the process is performing the same task repeatedly; and Rule Set C reflects the argument and return value behavior when the process being monitored is performing different tasks.

In our trace-driven evaluation using logs from a production web server, we show that the proposed technique can not only detect real attacks, but learn useful rules for intrusion detection that cannot be learned in prior approaches. False-alarm rates of our system are shown to be low in our trace-driven evaluation. We further perform evaluations on the convergence of the training process and the overhead experienced when using our system in real-time monitoring.

The organization of the rest of the paper is as follows. In Section 2, we present the motivations of our technique. The details of the design of our system are presented in Section 3. Section 4 shows the evaluation results. Finally, we show some related work in Section 5 and conclude with future work in Section 6.

2 Motivations of our technique

Figure 1 shows that context information from control-flow analysis is needed in learning the data-flow relations among system call arguments and return values. Such context information can be obtained via static analysis of the source or binary, or from dynamic analysis of executions of the program. In this paper, we restrict ourselves to dynamic analysis, because of not only its wide applicability in most environments and its simplicity in the analysis, but also its accuracy in learning relations governing normal executions (instead of all possible executions in static analysis) of the program. We leave using static analysis as future work.

Control-flow-based detectors using dynamic analysis have proposed using real-time information, e.g., program counters, call stack, to learn the context of a system call [5, 7,8,13]. However, as pointed out in Section 1 using the code segment in Block C in Figure 1, the context information we need in learning data-flow relations is not readily available from this information. Intuitively, the context information needed is about program behavior before and after the system call is made. In other words, it is about the sub-task the program is performing when the system call is made.

Sliding window [6] of system calls is a relatively close fit to what we need. It provides the context information about system calls made before and after the system call under analysis, and could be used to differentiate the two cases in Block C of Figure 1. However, variable-length patterns extracted from system call sequences more naturally reflect the behavior of an application than fixed-length patterns [8, 19, 20]. Intuitively, each such variable-length pattern corresponds to a task performed by the application. For example, the system call `read` in the two cases shown in Block C of Figure 1 will fall into two different variable-length patterns. Another advantage of using variable-length patterns is that these patterns can be extracted without knowing the actual task each pattern is performing. Techniques for extracting these patterns are based on dynamic learning and have been used successfully in a number of host-based intrusion detection systems [8–10, 19, 20].

Since each system call pattern corresponds to a task performed by the application, there will be strong relations among arguments of system calls within a single pattern. E.g., system calls may be made within a single pattern to open a file and read from it. File descriptors used in the `open` and `read` system calls may have to be equal. System calls from two patterns may be related as well, where

the two patterns could be repetitions of the same pattern, or they could be different patterns. An example of the former case could be that a system call pattern is used to read one block of data, and this pattern needs to be performed repeatedly in order to read a large portion of the data. The file descriptors used in the repeating system call patterns may have to be the same. An example of the latter case could be that one system call pattern of reading a file is followed by another pattern to close the file, in which the file descriptors must be the same, too. Although we have been using the file descriptor example so far, there are many other examples: directories may share the same prefix in their paths within one system call pattern, the return value of a system call in a pattern may be used as an argument of a system call in another pattern, etc.

Motivated by the above observations, we define three *Rule Sets* to capture the relations among system call arguments and return values using the context information from variable-length patterns:

1. Rule Set A contains rules that reveal the argument and return value relations among system calls within one system call pattern, i.e., rules governing the system call behavior when the process being monitored is performing an individual task. An example of these rules could be: the return value of the first system call of a pattern must be the same as the second argument of the second system call.

2. Rule Set B exploits rules that govern the system calls from repeating system call patterns, i.e., when the process is performing the same task repeatedly. It may contain rules such as: for each repeated appearance of a pattern , the value of the first argument of the third system call must be greater than its value in the previous appearance by 1.

3. Rule Set C reflects the behavior of system calls from different patterns, i.e., when the process being monitored is performing different tasks. An example of the rules might be: the return value of the second system call in one pattern and the first argument of the third system call in another pattern must have the same value when these two patterns appear consecutively.

Rules in each of these three Rule Sets are learned using techniques inspired by association rule mining [1, 21]. We present the details of learning rules in the next Section.

3 Relations mining

In this section, we describe in detail the relations mining in system call arguments and return values by using the context information obtained from the variable-length system call patterns. We first give an overview in Section 3.1, followed by the extraction of system call patterns using control-flow analysis (Section 3.2). We then describe the relations our system is capable of learning and how they are learned (Section 3.3). Finally we show the on-line monitoring using our system in Section 3.4.

3.1 Overview

The input training data, which contains the system call sequences (along with arguments and return values) recorded when the application is running in a benign environment, is first passed to a pattern extraction engine to do control-flow analysis. The engine analyzes the sequences and outputs a set of patterns (system call subsequences) and a representation of the training data using these patterns. After that, three types of rules are learned by applying relations mining techniques. Finally, rules learned are used for online monitoring of the application.

3.2 System call patterns via control-flow analysis

The control-flow analysis techniques we use consists of the Teiresias algorithm [12] and a pattern reduction algorithm [20]. These algorithms have been used successfully in many projects for improving the intrusion detection systems [8–10, 19, 20]. Note that other techniques for extracting patterns can be used as well. Intuitively, each pattern extracted corresponds to a task performed by the program. Table 1 shows an pattern example which is composed of three system calls. Note that although the training data contains information of the system call arguments and return values, the control-flow analysis we perform here makes use of only the system call numbers to extract patterns.

syscall No.	168	003	078
syscall name	`poll`	`read`	`gettimeofday`

Table 1. Systems calls in a pattern

With the system call patterns found, the system call sequences in the training data can be represented in terms of the patterns and the corresponding system calls, as well as the arguments and return values of the system calls in the patterns. For simplicity, we call arguments and return value of a system call *attributes* in the rest of this paper.

3.3 Relations and relations mining

With pattern extraction, each system call in the training data falls in a particular pattern, which provides the context information for us to do more accurate relations mining. In this subsection, we show how the relations among system call arguments and return values are learned by using this

information. We first group system call arguments and return values based on their data types. We then present our generalized form for the rules and the three Rule Sets. Inspired by association rule mining techniques [1,21], we implement two evaluations on each rule learned to filter out rules that may cause too many false positives or negatives.

3.3.1 Macro-Types

Intuitively, only relations between two attributes of the same data type are useful. E.g., the relation between the value of an `int` and a `char` is not very meaningful. However, after checking the 111 distinct data types from the 324 system calls defined in Linux kernel 2.6.22, we realized that system call arguments of two different data types might be related as well. For example, the data type of the return value of system call `open` is a `long`, which is a file descriptor; while file descriptors in system calls `read` and `write` are defined as `unsigned int`. This suggests that some consolidation of the 111 data types is required. For this purpose and to simplify our analysis, we group the large number of data types into a small number of "Macro-Types". Any attributes with the same Macro-Type are considered *comparable* and their relations are candidate rules to be learned. Table 2 shows the 5 Macro-Types we define, as well as some of their members (originally defined data types).

Macro-Type	Data types defined in Linux kernel 2.6.22
Integer	long, int, unsigned int, unsigned long, size_t, pid_t, etc.
Integer*	int __user *, time_t __user *, old_sigset_t __user *, etc.
String*	char __user *, const char __user *, etc.
Struct*	struct __old_kernel_stat __user *, struct tms __user *, etc.
Others	struct pt_regs, void __user *, __sighandler_t, etc.

Table 2. Five Macro-Types in our system

Table 3 show the same system call pattern as shown in Table 1 before and after the consolidation of data types, respectively. Note that each system call has 6 *attributes* associated, out of which one is used to represent the return address (attr0) and the other 5 are used to represent the first 5 arguments of the system call[1]. We fill an unused attributes with "null" if there are less than 5 arguments.

3.3.2 Expressions, operators, and statements

A simple example of the relations that we want to learn could be of the form "The first attribute of one system call in a particular system call pattern equals to the second attribute of another system call in another pattern". We call this simple relation a *statement*. Within a statement, we call the attributes *expressions* and the relation (e.g., "equal") an *operator*. To show the expressiveness of the relations our system is able to learn, we describe what expressions, operators, and statements are in this subsection.

An expression (denoted e) could be either an attribute (denoted a) or a constant (denoted c). A statement (denoted s) defines the relation between two expressions, where the relation is defined by an operator (denoted o). Simple operators include equal to, not equal to, greater than, less than and etc. Note that a statement can only be formed by two expressions that are *comparable* using a corresponding operator. Two expressions are comparable if they are of the same Macro-Type (Section 3.3.1), and in order to simplify our system, we only define 5 Macro-Types as shown in Table 2. Two statements and a logical operator could be used to form another statement, e.g., "the first attribute equals 1 AND the second attribute equals 2". In summary,

$$
\begin{aligned}
a &::= \text{attribute} \\
c &::= \text{constant} \\
e &::= a|c \\
o &::= \text{equal|greater than|AND|GIVEN|etc.} \\
s &::= e\ o\ e|s\ o\ s
\end{aligned}
$$

We also allow *functions* to be defined. A function takes in an expression as input and outputs another expression. E.g., a function can be used to find the substring, which, in turn, can help to form a statement that two string expressions share the same prefix. Another very important use of functions is to dereference a pointer. In many cases it will make more sense to compare the data that the attributes point to instead of comparing the attributes themselves. E.g., a function can be defined to dereference a particular member in a structure.

Functions are very flexible and can help describe a wide variety of relations, although here we do not discuss further on other possible formats a function can take.

3.3.3 Three Rule Sets

System call patterns provide the context information for us to differentiate the same system call in different execution context (different patterns). Attributes in a statement may belong to system calls in the same pattern, in the repeated occurrences of one pattern, or in different patterns. We classify statements into three different Rule Sets that represent different types of relations among system calls.

Relations in different Rule Sets may play different roles in detecting intrusions. For example, in a particular program, training may reveal that attributes of system calls within the same pattern are closely related, whereas in another program, attributes of system calls from repeating patterns may have stronger relations. By classifying relations

[1]Of the 324 system calls defined in the Linux kernel 2.6.22, only 8 of them have more than 5 arguments yet are rarely seen in our training data. We do not examine these additional system call arguments in this project since the small number of appearances does not suffice to reveal any rule.

	Before Consolication			After Consolication		
syscall Number	168	003	078	168	003	078
syscall name	`poll`	`read`	`gettimeofday`	`poll`	`read`	`gettimeofday`
attr0	long	size_t	long	Integer	Integer	Integer
attr1	struct pollfd __user*	unsigned int	struct timeval __user*	Struct*	Integer	Struct*
attr2	unsigned int	char*	struct timezone __user*	Integer	String*	Struct*
attr3	long	size_t	null	Integer	Integer	null
attr4	null	null	null	null	null	null
attr5	null	null	null	null	null	null

Table 3. System call arguments and return values before and after data type consolidation

into different Rule Sets, we may assign different weights to them for online monitoring, though we leave it as our future work and in the implementation in Section 4, we assign the same weights to simplify our system.

Rule Set A contains statements in which attributes belong to system calls in the same pattern. Recall that a system call pattern corresponds to the performance of a single task by the application. System call made within one pattern are closely related to one another as they are steps in performing the same task. Statements in Rule Set A represent the relations among these system calls. A typical example of such statement could be

$$P[i].S[j].A[k] = P[i].S[j'].A[k']$$

where $P[i]$ denotes a particular pattern we extract from system call sequences (i is just an index to denote different patterns), $P[i].S[j]$ denotes the j^{th} system call of pattern $P[i]$, and $P[i].S[j].A[k]$ denotes the k^{th} attribute of $P[i].S[j]$ ($A[0]$ represents the return value).

Rule Set B contains statements in which attributes belong to system calls in repeated occurrences of a pattern. It is very common that the same system call pattern is used repeatedly to perform a long task. E.g., a system call pattern may be used to read a small portion of data, and such pattern needs to be used repeatedly in order to read data from a large file. In these cases, there is also close relations among the attributes of system calls from repeated patterns. A typical example of statements in Rule Set B could be

$$\mathsf{Occ}^{m}_{P[i].S[j].A[k]} = c_1 \text{ GIVEN } \mathsf{Occ}^{m-1}_{P[i].S[j].A[k]} = c_2$$

where $\mathsf{Occ}^{m}_{P[i].S[j].A[k]}$ denotes the m^{th} occurrence of $P[i].S[j].A[k]$. This statement says that if the k^{th} attribute of the system call $P[i].S[j]$ equals to c_2, then the same attribute of the same system call in the next occurrence of the pattern must equal to c_1.

Rule Set C contains statements in which attributes belong to system calls in different patterns. These statements are used to govern system calls from different patterns that are closely related. Intuitively, although one system call pattern corresponds to a particular task to be performed, in many cases a complicated task has to be done by a sequence of patterns. Attributes in these patterns could be highly correlated, and we use Rule Set C to represent them. A typical example of statements in Rule Set C could be

$$P[i].S[j].A[k] \overset{dist}{=} P[i'].S[j'].A[k'](dist < \mathsf{maxdist})$$

where $dist$ represents the number of patterns between $P[i]$ and $P[i']$ in the system call sequence, and maxdist is a threshold denoting the maximum distance between the two patterns where the statement is valid.

3.3.4 Minimum support and confidence level

Many statements can be found from a small size of training data. However, many of them may not be reliable and may cause false positives and false negatives. We define two thresholds to filter out statements that are not very reliable.

As in association rule mining process, two measurements *support* and *confidence* are calculated for each rule. In our system, the support of a statement is the number of times a statement is found valid in the training data. We define a threshold minsup to specify the lower bound of the support for a statement to be accepted in our Rule Sets. The confidence of a statement is (conditional) probability that the statement is found valid in the training data.

The values of minsup and minconf play an important role in tuning the intrusion detection system to have the right trade-off between false positives and false negatives. It is not our objective to propose comprehensive techniques for finding the right values for these two thresholds in this paper. However, we provide our evaluation results in Section 4 with our choice of the thresholds to demonstrate the detection capability of our system.

For our rule generation, we first match the attributes in pairs when two system calls are from a single pattern, from repeating occurrences of a pattern, or from two different patterns. A pair of attributes can form a simple statement with a specific operator. We calculate the support and confidence level by checking the relation between the values of the two attributes involved in the statement. We accept a statement with support and confidence over the thresholds minsup and minconf as a rule in our model and add it to the corresponding Rule Set. These simple statements are

then further extended to form more complicated statements following the format $s\ o\ s$, and the support and confidence level are measured similarly to decide if the newly formed statements should be accepted or not. The iterative process of forming more and more complicated statements will stop when these two levels drop below the thresholds.

3.4 Online monitoring

Relations learned can be used for online monitoring. The online monitor intercepts system calls made by the process in real time, and analyzes the system call along with its arguments and return value. Once our detection system recognizes one pattern from the system call sequence, we test the arguments and return values on all the rules with this pattern involved in three Rule Sets we constructed during the training. An alarm will be raised when the cumulative number of abnormal behavior reaches a previously defined threshold. We present the experimental results for measuring the online monitoring overhead in Section 4.6.

4 Evaluation

In this section, we first present our experimental setup and some examples of relations learned in each Rule Set (Section 4.1). In Section 4.2, we show that our system is capable of detecting real attacks with a couple of examples. The advantages of our technique when compared with prior ones are shown in Section 4.3. In particular, we show that our technique is able to learn data-flow properties that cannot be learned by previous data-flow analysis techniques. The performance of our system, including the false alarm rate, speed of convergence, as well as the overhead in online monitoring are presented in the last three subsections.

4.1 Experimental setup and relations learned

4.1.1 Experimental setup

Our system was implemented on a desktop computer with Linux kernel 2.6.22 that has 324 system calls defined. We used `Apache2` to host an `http` server to simulate the web server of Singapore Management University. We modified the Linux kernel to intercept system calls made by the `Apache2` web server to obtain real-time system call information including the system call numbers, arguments and return values. The static web pages on the Singapore Management University web server were downloaded and hosted on our web server. We then made use of the web logs from the real server to replay requests from August 2007 to December 2007.

In this part of the evaluation, we replayed 3 days (typical weekdays) of logs that contain $372,940$ `http` requests, and extracted 89 system call patterns which can cover all the system call sequences in the training data. We group the 111 data types for the system call arguments and return values into 5 Macro-Types as shown in Table 2. One of the simplest form of expressions and statements is used in this evaluation to show the effectiveness of our system. Statements we use are of the form

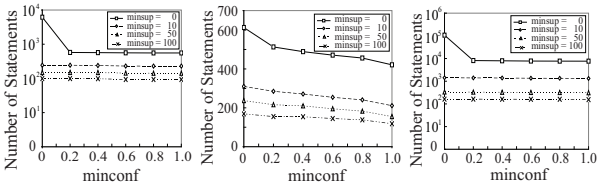
$$s ::= f(a_1) = f(a_2)$$

That is, we try to find rules that govern the equality of (some property, defined by function $f()$, of) two system call attributes. Function $f()$ does simple dereferencing when the attributes being examined are either integer pointers or string pointers.

We are interested in finding statements in all the three Rule Sets, and let $\mathsf{maxdist} = 10$ in finding rules in Rule Set C. Different settings of minsup and minconf are used in our evaluation to shed light on the settings of the thresholds.

4.1.2 Number of statements in each Rule Set

Figure 2 shows the number of rules learned in each Rule Set for a number of different settings of minsup and minconf.



(a) Rule Set A (b) Rule Set B (c) Rule Set C

Figure 2. Number of rules learned

From the 4 settings of minsup, we can see that the setting of minsup is effective in filtering out statements that have less coverage in the training data, i.e., when minsup increases, the number of statements drops significantly. When looking at different settings of minconf, we realize that the setting does not have a significant effect except when $\mathsf{minsup} = 0$. In particular, when both minsup and minconf are 0, our system learns a large number of statements which are not useful (their confidence level is 0). However, the relatively stable results with different settings of minconf when $\mathsf{minsup} \neq 0$ suggest that many of the statements learned by our system with nonzero support have confidence level of 1, which are very reliable rules.

4.1.3 Rule examples

Here we show some examples of the statements found.

Rule Set A The following is an example of statements found in Rule Set A with a support of $8,544$ and confidence level of 1.0, which says that within the particular pattern $P[7]$, the second argument of the 5^{th} system call must be equal to the return value of the 2^{nd} system call.

$$P[7].S[2].A[0] = P[7].S[5].A[2]$$

The details of the pattern $P[7]$ is shown as below:

```
P[7]: stat64 open socketcall writev sendfile64
      socketcall read write close
```

By analyzing the system calls in $P[7]$, we find that this system call pattern is probably involved in a file sending process. After examining the system call attributes, we find that the second argument of system call `sendfile64` is defined as an `int`, while the system call `open` returns a file descriptor, despite that it is defined as a `long`.

Rule Set B We find another statement about pattern $No.7$ in Rule Set B as follows with support of $8,543$ and confidence level of 1.0.

$$\mathsf{Occ}^{m}_{P[7].S[7].A[1]} = \mathsf{Occ}^{m-1}_{P[7].S[7].A[1]}$$

This statement says that when pattern $P[7]$ is used repeatedly, the 1^{st} argument of `read` (the 7^{th} system call) must not change.

Rule Set C We get the following statement in Rule Set C

$$P[88].S[2].A[1] \overset{dist=0}{=} P[1].S[4].A[1]$$

with support of 230 and confidence level of 1.0, where $dist = 0$ means that pattern $P[88]$ follows pattern $P[1]$ immediately.

System calls in pattern $P[88]$ and pattern $P[1]$ are presented in the expressions below respectively.

```
P[88]: poll sendfile64 read write close
P[1] : stat64 open socketcall writev sendfile64
       socketcall
```

This statement says that when $P[88]$ follows $P[1]$ immediately, the 1^{st} argument of the system call `sendfile64` in $P[88]$ is the same as the 1^{st} argument of the system call `writev` in $P[1]$.

The above examples show that our system is able to learn important and meaningful rules that govern the system call arguments and return values. However, we also find some less useful statements, e.g.,

$$\mathsf{Occ}^{m}_{P[7].S[9].A[0]} = \mathsf{Occ}^{m-1}_{P[7].S[9].A[0]}$$

in Rule Set B. The fact is, system call `close` (the 9^{th} system call in pattern $P[7]$) always returns a value 0 when it succeeds. This statement is less useful because it does not express a unique feature of the system calls made by the application being monitored. However, it is still a valid and good statement in the sense that it makes mimicry attacks more difficult because the attackers need to make sure that this particular system call returns the right value to avoid being detected. In contrast, attackers are free to make null system calls in a mimicry attack [8, 18].

4.2 Real attacks detection

To show the effectiveness of our approach in attack detection, in this subsection we give two examples to demonstrate the capability of our system of detecting real attacks. We then show in Section 4.3 that some relations learned by our system is useful in enhancing the accuracy of the detection system, which cannot be learned with prior approaches. For each of the attacks shown in this subsection, we trained the programs involved with benign input to learn relations in the three Rule Sets, and then ran the exploits and checked for the violation of the rules learned.

4.2.1 Attacks on file descriptors

The fact that programs may make assumptions about the meanings of file descriptors, e.g., the descriptor 2 corresponds to `stderr`, may render the programs vulnerable to some simple exploits. Chen [3] described in detail a program with such a vulnerability which contains the code fragment as shown in Figure 3.

```
fd = open("/etc/passwd");
str = read_from_user();
fprintf(stderr, "The user entered: \n%s\n", str);
```

Figure 3. The stderr attack

If the attacker closes `stderr` before executing this program, an open of "/etc/passwd" will return the file descriptor to both `fd` and `stderr`. Subsequently, `fprintf` will write user input data into the password file.

Our system is capable of recognizing the frequently used system call pattern `open, read, write` and generating a rule saying that the return value of system call `open` must be *not equal to* the first argument of system call `write`. The above mentioned attack was detected by our system as it violated the rule learned.

4.2.2 Directory traversal attacks

A buffer overflow in the `GHTTPD` web server may be used by the attacker to evade path checking and execute a malicious program [4]. Consider the code fragment in function `serverconnection` as presented in Figure 4.

```
if (strstr(ptr, "/..")
  return ...  //reject request
Log(...); \\
if (strstr(ptr, "cgi-bin"))
  execve(ptr, ...);
```

Figure 4. Directory traversal

This function only checks the presence of "cgi-bin" in the URL string pointed by the variable `ptr` before the CGI request is processed. By exploiting a buffer overflow vulnerability in function `Log`, attackers can change `ptr` to point to a string `/cgi-bin/../../../../bin/sh` and successfully gain access to a shell.

Training this program in our system, the repeating appearances of the system call `execve` lead to a rule in Rule Set B saying that the first argument of `execve` must have the common prefix of "/usr/local/ghttpd/cgi-bin/" across its repeated occurrences. Though our rule was learned by observing the repeating occurrences of one system call pattern, it describes the same observation as a unary relation that is covered by another data-flow analysis technique monitoring system call arguments [2].

4.3 New rules learned

As shown in Section 1 and Section 2, our technique is able to leverage results from control-flow analysis to learn rules that cannot be learned by prior techniques. In this subsection, we show a real example of such rules learned in our trace-driven evaluation.

Our system recognized two patterns $P[16]$ and $P[7]$ from the trace-evaluation of the `Apache2` web server. Both patterns contain system call `read` followed by system call `write` as shown in the following expressions.

```
P[16]: read poll write close
P[7] : stat64 open socketcall writev sendfile64
       socketcall read write close
```

A rule learned in pattern $P[16]$ shows that system call `read` and `write` share the common file descriptor in their arguments. However, this rule is not valid in pattern $P[7]$. Instead, we learned a different rule for the system calls `read` and `write` in pattern $P[7]$, which says that the string pointer arguments of these two system calls must point to strings of the same content.

This is a good example in which control-flow information helps learn an accurate and useful rule. Being unable to recognize the patterns from a system call sequence, prior approach in data-flow analysis would fail since neither of the two relations has high probability of being valid in all occurrences of the system calls `read` and `write` in the entire sequence. On the other hand, our technique makes use of the control-flow information to differentiate the system calls in two different patterns, which results in two rules that accurately describe relations in two patterns respectively.

4.4 False alarm rates

To evaluate the false alarm rate of our system simulating a university production web server, we first used 36 hours of web logs with $178,043$ benign `http` requests to learn rules in the three Rule Sets. After that, we consumed 60 hours of logs with $253,418$ benign requests and recorded the false alarms generated. Results are presented in Table 4.

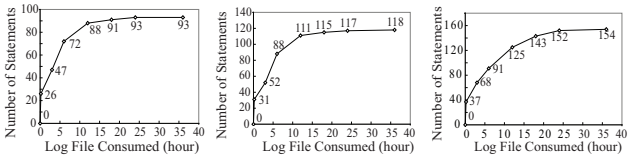
Training	Testing	False alarm rates in each Rule Set		
# of syscalls	# of syscalls	A	B	C
1.70×10^6	2.89×10^6	0	9.80×10^{-5}	1.37×10^{-5}

Table 4. Evaluation of a web server

We observe that the false alarm rates are of the order of 10^{-5}. In particular, we did not find any false alarms for relations in Rule Set A, and only found a small number of false alarms in Rule Set B and C. Considering the fact that some events might actually break the rules respectively from Rule Set B and Rule Set C simultaneously but we were double counting the violations, the total false alarm rate would be a smaller number than the sum of the individual records. Our system experiences low false alarm rates.

4.5 Speed of convergence

Fig. 5 shows the speed of convergence for each Rule Set in our evaluation with $\mathsf{maxdist} = 10$, $\mathsf{minsup} = 100$, and $\mathsf{minconf} = 1.0$.



(a) Rule Set A (b) Rule Set B (c) Rule Set C

Figure 5. Speed of Convergence

Figure 5 suggests that using 24 hours of logs of a production web server (113,732 `http` requests) is sufficient to learn more than 90% of the rules. By performing this training using a desktop computer with a 2.2 GHz CPU and 1 GB of memory, this training (un-optimized) takes about 12 hours. Note that this training can be done off-line.

4.6 Overhead in online monitoring

In this part, we show the overhead of our system when it is implemented for online monitoring of system calls of a

production web server. We host the web server on a desktop computer with a 2.2 GHz CPU and 1 GB of memory. The computer runs the Linux operating system with kernel 2.6.22 and the `Apache2` web server. The kernel is instrumented to intercept system calls made by the web server for real-time monitoring. Statements in the three Rule Sets are obtained by training 36 hours of logs of the Singapore Management University web server.

We use a program to simulate single or multiple clients sending `http` requests to the web server. Each client reads one entry from the log file at a time and then sends the request to the web server. Each client was configured to send requests with 10 milliseconds interval and each run last 60 seconds. To evaluate the monitoring overhead, we measure the latency experienced by the client. Latency is defined as the difference between the time when a `http` request is sent and the time when the client receives a response from the server. Note that our simulated client is located in the same Local Area Network as the web server is. We run a few tests, each with a different number of concurrent clients. The average latency for each test is presented in Figure 6.

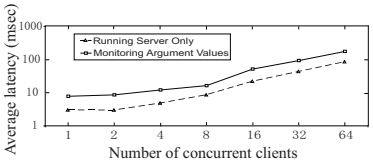


Figure 6. Average latency experienced

Results show that our monitoring system adds 5 to 8 milliseconds on average to the latency when there are less than (or equal to) 8 concurrent clients. When the number of concurrent clients increases to 64, the clients experience an additional 75-millisecond of latency on average. Such additional latency is hardly noticeable by human beings. Also note that the results presented in Figure 6 are latencies measured by clients on the same Local Area Network of the server. Considering the latency suffered by clients over the Internet, which is typically at least a few hundred milliseconds[2], the additional latency caused by our real-time monitoring accounts for a comparatively minor part.

5 Related work

Control-flow information, associated with system call sequences emitted by the application being monitored, has been used in the literature to combat intrusions. To improve Forrest et al.ś model [6] of fixed-length patterns, Wespi et al. introduced variable-length patterns to better describe the application behavior [19, 20]. Static analysis techniques were introduced by Wagner et al. [17] to thoroughly explore all possible executions of the application. Sekar's FSA model [13] utilized program counter information to capture both short term and long term temporal relations of system calls, while Gao's [8] and Feng's [5] models relied on the call stack information to extract paths of the program executions. All of these approaches, and many others, make use of system call sequence information but the dataflow among the arguments are missing, which leads to the possibility of evasion attacks, such as mimicry attack [8, 18].

In data-flow analysis detectors, there has been previous work on utilizing system call arguments. Kruegel et al. constructed models based on the characteristics, e.g., the length of strings, string character distribution, and structural inference, of system call arguments [11]. The model returns the probability that a system call argument has the corresponding value during detection. Low probabilities indicate potential attacks. While their work emphasized on the characteristics of each single argument, inter-relationships among the arguments are not explored. To enhance an IDS to combat mimicry attacks, Sufatrio et al. proposed a simple extension that incorporates system call arguments and process privileges [14]. They abstracted the values by categorizing them into classes that are defined by user-supplied category specifications. However, their abstraction rendered the relationship among the system call arguments not examinable. Tandon et al. integrated arguments and attributes of system calls into their LERAD system with a fixed-size window and focused on the value set allowed for each argument [15,16]. Bhatkar et al. managed to extract rules on system call arguments by analyzing the data-flow in a control-flow context [2]. However, the only information they employed from control-flow context was the program counter. On the other hand, our approach leverage the results from control-flow analysis, in particular, system call patterns that partition long system call sequences into sub-sequences that correspond to small tasks performed, to learn more accurate and useful rules that cannot be learned in prior approaches.

Association rule mining techniques that help discover rules from a large database of transactions [1, 21] are also related to our work. Association rule mining has many applications, e.g., in making business decisions such as what to put on sale, how to design coupons, etc. The problem of association rule mining is closely related to learning rules governing system call arguments and return values in that both are trying to find relations among a large data set. Unlike the ad hoc approaches taken by previous work on system call argument, we adopt the ideas in well-studied association rule mining techniques in time series and apply them to learn rules governing system call arguments and return values with moderate modifications and generalization.

[2] For example, we measured the latency between a machine on our campus network and the web server of `www.yahoo.com`. Results were between 569 milliseconds and 576 milliseconds in 15 runs.

6 Conclusion and future work

In this paper we propose a new model for data-flow analysis for intrusion detection, which leverage the results from control-flow analysis to learn more accurate and useful rules among system call arguments and return values. To the best of our knowledge, this is the first paper that tries to bridge the gap between data-flow and control-flow analysis for intrusion detection. Through trace-driven evaluations, we show that our technique is not only able to detect real attacks with low false alarm rates, but also capable to learning new rules that are useful in intrusion detection.

In our future work, we would like to design a more delicate way of grouping system calls and their arguments so that the number of non-significant rules could be reduced. We also would like to generate statements of form more complicated than equality, to enrich the profile in our model to achieve greater accuracy and scalability. Moreover, by assigning different weights to rules in different sets, we would try to further improve the accuracy of our system.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of 1993 ACM-SIGMOD International Conference on Management of Data, 207-216. Washington, D.C*, 1993.

[2] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[3] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of c code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium NDSS*, 2004.

[4] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th conference on USENIX Security Symposium*, 2006.

[5] H. Feng, O. Kolesnikov, P. Fogla, and W. Lee. Anomaly detection using call stack information. In *Proceedings: IEEE Symposium on Security and Privacy. Berkeley, California*, 2003.

[6] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy, pages 120-128, Los Alamitos, CA*, 1996.

[7] D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the 11th ACM Conference on Computer & Communication Security (CCS 2003)*, 2003.

[8] D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[9] D. Gao, M. K. Reiter, and D. Song. Behavioral distance for intrusion detection. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, 2005.

[10] D. Gao, M. K. Reiter, and D. Song. Behavioral distance measurement using Hidden Markov Models. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID 2006)*, 2006.

[11] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *In Proceeding of ESORICS 2003*, 2003.

[12] I. Rigoutsos and A. Floratos. Combinatorial pattern discovery in biological sequences the teiresias algorithm. In *Bioinformatics, 14(1):55-67*, 1998.

[13] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[14] Sufatrio and R. H. C. Yap. Improving host-based ids with argument abstraction to prevent mimicry attacks. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, 2005.

[15] G. Tandon and P. Chan. Learning rules from system calls arguments and sequences for anomaly detection. In *ICDM Workshop on Data Mining for Computer Security (DMSEC), Melbourne, FL*, 2003.

[16] G. Tandon and P. Chan. Learning useful system call attributes for anomaly detection. In *Proceedings of the 18th International FLAIRS Conference*, 2005.

[17] D. Wagner and D. Dean. Intrusion detection via static analysis. In *2001 IEEE Symposium on Security and Privacy*, 2001.

[18] D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer & Communication Security (CCS 2002)*, 2002.

[19] A. Wespi, M. Dacier, and H. Debar. An intrusion-detection system based on the teiresias pattern-discovery algorithm. In *Proceedings of the 1999 European Institute for Computer Anti-Virus Research Conference*, 1999.

[20] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable-length audit trail patterns. In *Proceedings of the 3rd International Symposium on Recent Advances in Intrusion Detection*, 2000.

[21] M. J. Zaki. Generating non-redundant association rules. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2000.