On Detection of Erratic Arguments

Jin Han, Qiang Yan, Robert H. Deng, and Debin Gao

Singapore Management University, Singapore
{jin.han.2007,qiang.yan.2008,robertdeng,dbgao}@smu.edu.sg

Abstract. Due to the erratic nature, the value of a function argument in one normal program execution could become illegal in another normal execution context. Attacks utilizing such erratic arguments are able to evade detections as fine-grained context information is unavailable in many existing detection schemes. In order to obtain such fine-grained context information, a precise model on the internal program states has to be built, which is impractical especially monitoring a closed source program alone. In this paper, we propose an intrusion detection scheme which builds on two diverse programs providing semantically-close functionality. Our model learns underlying semantic correlation of the argument values in these programs, and consequently gains more accurate context information compared to existing schemes. Through experiments, we show that such context information is effective in detecting attacks which manipulate erratic arguments with comparable false positive rates.

Keywords: Intrusion detection, system call argument, diversity

1 Introduction

Host-based anomaly detection techniques based on behaviors of programs in terms of system call sequences were first proposed by Forrest et al. [8], and improved and extended by a number of research work [7,9,13,14,19,23]. The normal-behavior models of the applications are learnt from the behaviors observed during a training phase; while during detection, any deviations from the established models are interpreted as attacks to the programs monitored. Later research [2,17,20,24] further enhanced the behavioral model by capturing the information of system call arguments.

Early schemes [17,20,24] model the argument behavior at the granularity of different system calls, i.e., each system call (e.g., `open`, `read`, `write`) is assigned with a profile. The granularity is then improved by differentiating the instances of the same system call when their call stacks are different [2]. For example, the legitimate arguments of `open@callstack1` and `open@callstack2` are assigned with different profiles so that they can be tested differently in the detection phase. However, since other context information is not captured during the training, an adversary is able to evade the detection of these existing schemes. Consider the following example code which assumes to contain a buffer overflow vulnerability:

M. Rajarajan et al. (Eds.): SecureComm 2011, LNICST 96, pp. 172–189, 2012.

```
int uid = geteuid();
char buf[128];
char* filename;
...
if (uid == 0)
    filename = "/www/admin/configure.ini";
else
    filename = "/www/user/configure.ini";
int fd = open(filename, O_RDWR);
write(fd, buf, sizeof(buf));
```

As illustrated in the example code, the system call `open` accepts two different parameter values in the training phase, both of which correspond to the same call stack. According to the existing schemes [2,17,24], both of these strings will be treated as legitimate values during detection. Thus, an attack which overflows `buf` and changes `uid` to 0 will be able to get the administrator privilege while evading detection. Such a situation is more common in modern software applications where code modules are extensively reused. Call stack is not able to tell a difference in the privilege in different executions.

The fundamental difficulty in detecting such attacks stems from the *erratic property* of function arguments. More formally, all legitimate values observed in different normal program executions are not necessarily legitimate at a particular execution. In a particular execution context, only a subset of the values (possibly one) is legitimate while others could potentially be malicious.

This problem seems deceptively simple. The fine-grained context information, which is required to differentiate the legitimate values at run-time, is difficult to gather when training merely one program [2,17,24], especially when the source code is not provided. Even for schemes which utilize two diverse applications, their model cannot be simply extended to detect such attacks. For example, hidden Markov models used in [10,11,12] (to train the normal-behavior profiles of the system call sequences) are only able to handle finite states, while the space of argument values is usually infinite.

In this paper, we propose an intrusion detection scheme which builds on two diverse programs providing semantically-close functionalities. Our model learns the underlying semantic correlation of the argument values in these programs to detect attacks manipulating erratic arguments, which are recognized as normal inputs by existing schemes. Specifically, we make the following contributions:

- We provide a formal approach of detecting attacks utilizing erratic arguments, by learning relations of the function arguments between programs providing semantically-close functionalities.
- We utilize taint analysis to further refine the detection model, which eliminates the coincident relations to decrease the false positive rates.
- We implement a prototype of our scheme and present a detailed experimental evaluation. The evaluation demonstrates that a number of real attacks which are hard to detect by existing schemes can be effectively detected using our technique. Specifically, it is shown that our detection model not only

detects sophisticated attacks on security-critical data, but also detects some Denial-of-Service attacks which are not addressed by existing techniques, with comparable false alarm rates.

2 Diversity Detection Model

In this section, we first introduce the framework of our detection approach, which is followed by the definitions of the argument relations. Different algorithms are then provided to train the behavioral model for different types of arguments.

2.1 Overview

Figure 1 illustrates the basic idea of how our intrusion detection system (IDS) is constructed. We regard two diverse software having *semantically-close functionalities* if they provide same services. Examples of such diverse software could be web servers like Apache and Lighttpd, or office software like Adobe PDF Reader and Foxit PDF Reader. Similar to existing diversity-based intrusion detection techniques, the framework in Figure 1 utilizes two diverse software providing semantically-close functionalities to build the behavioral model, base on the observations that these software cannot be successfully exploited by the same attack [15].

In this paper, we focus on building a normal-behavior model by extracting the function arguments of both applications. Since these applications provide semantically-close functionalities, there are semantic relations between the behaviors of these applications when they process the same input. **Such semantic relations will exhibit as the relations between the related function calls and their argument values.** For example, two web servers processing the same HTTP request need to access the same local file on the disk. Thus, consequently, there should be functions in both applications whose argument values contain the same file name. In the following, we will briefly introduce how our model captures the argument relations between the two diverse applications.

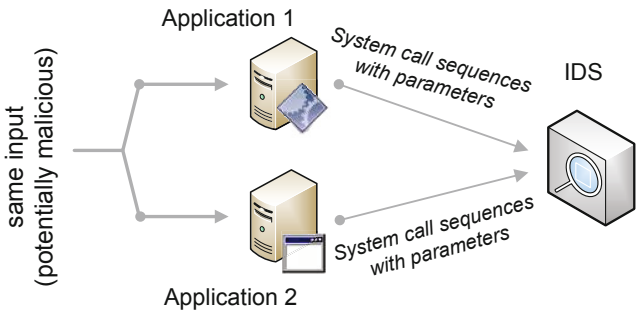

Fig. 1. Our diversity IDS framework

Once the argument relations are trained, they will be utilized to detect attacks that attempt to fool traditional IDS with erratic function arguments.

In the model of Figure 1, the same inputs, which are assumed to be free of attacks in the training phase, are passed to both of these applications (app_1, app_2). In order to process the input, each of these applications will invoke a series of system calls (for each input):

$$S_1 = \langle s_{1,1}, s_{1,2}, ..., s_{1,l_1} \rangle \qquad S_2 = \langle s_{2,1}, s_{2,2}, ..., s_{2,l_2} \rangle \tag{1}$$

Each system call $s_{i,j}$ has a vector of arguments. In the training phase, all information for each $s_{i,j}$ will be recorded by corresponding monitor module of $\mathsf{app_i}$, and is used to extract the information of the arguments. Specifically, in our model, each argument is identified by:

$$\mathsf{arg_{i,x}} \text{ where } i \in \{1, 2\},$$
$$x = \langle \mathsf{index},\ \mathsf{type},\ \mathsf{s\_name},\ \mathsf{callstack} \rangle.$$

In the above representation, i in $\mathsf{arg_{i,x}}$ indicates this argument appears in the trace of $\mathsf{app_i}$. index is the position of this argument in the corresponding system call, whose name is s_name; type is the type of the argument (e.g. *string* or *integer*); callstack stores the call stack information of the corresponding invocation of this particular system call.

In the training phase, we first obtain a pair of system call traces (S_1, S_2) for each input. With all pairs of the system call traces, we then get a set of argument pairs. For each argument pair ($\mathsf{arg_{1,p}}$, $\mathsf{arg_{2,q}}$), $\mathsf{arg_{1,p}}$ is an argument in app_1, which is identified by a unique set of $\langle \mathsf{index},\ \mathsf{type},\ \mathsf{s\_name},\ \mathsf{callstack} \rangle$ appearing in the training set, and $\mathsf{arg_{2,q}}$ is defined similarly. From the training data, we collect a set of value pairs $\mathsf{Value_{p,q}}$ for each argument pair, where $\mathsf{Value_{p,q}} = \{(\mathsf{v}_1, \mathsf{v}_2) | \mathsf{arg_{1,p}} = \mathsf{v}_1, \mathsf{arg_{2,q}} = \mathsf{v}_2\}$. According to $\mathsf{Value_{p,q}}$, we then produce a database of relations $\mathbb{R} = \{\langle \mathsf{arg_{1,p}}\ \mathsf{R}\ \mathsf{arg_{2,q}} \rangle\}$. This relation set $\mathbb{R}$ is finally utilized to detect whether there is any violation for each pair of parameter values. If the relation of a pair of parameter instances ($\langle \mathsf{arg_{1,p}} = \mathsf{v_x} \rangle$ and $\langle \mathsf{arg_{2,q}} = \mathsf{v_y} \rangle$) does not satisfy the corresponding $\langle \mathsf{arg_{1,p}}\ \mathsf{R}\ \mathsf{arg_{2,q}} \rangle$ in $\mathbb{R}$, the IDS will raise an alarm.

2.2 Relationships of the Arguments

In our model, we focus on two most common types of system call arguments – *string* and *integer*, the definitions of which follow the standard definition in programming language: a *string* is a sequence of zero or more characters followed by a `NULL` ("\0") character; while an *integer* is a numeric variable holding whole numbers.

We define binary relation R that captures the relationship between two system call arguments in the diverse applications. The relation between two arguments is expressed as $\langle \mathsf{arg_1}\ \mathsf{R}\ \mathsf{arg_2} \rangle$, where $\mathsf{arg_1}$ is a particular argument in the first application, and $\mathsf{arg_2}$ is a particular argument in the second application. Different sets of candidate relations are given to *string* and *integer* since these two argument types have different characteristics.

We provide the following basic relations for *string* arguments:

- **equal** captures equality relation of the given two arguments, e.g., the file name passed to an `open` system call in app_1 could be the same as the file name passed to another `open` (or `stat64`) system call in app_2.
- **samePrefix(n)** indicates that the two string arguments have the same prefix, the length of which is at least n. For example, if arg_1 = `"/home/usr/xyz"` and arg_2 = `"/home/usr/abc"`, then $\langle \mathsf{arg}_1\ \mathsf{samePrefix(10)}\ \mathsf{arg}_2 \rangle$ holds.
- **sameSuffix(n)** indicates that the two string arguments have the same suffix substring with length at least n.
- **contain** means that the second argument is a substring of the first argument.
- **partOf** is the reverse of **contain** relation, in which the first argument is a substring of the second argument.

Note that for the same pair of arguments, more than one of the above relations may hold. For example, if arg_1 = `"/home/configure.ini"` and arg_2 = `"/home/conf.ini"`, then both $\langle \mathsf{arg}_1\ \mathsf{samePrefix(10)}\ \mathsf{arg}_2 \rangle$ and $\langle \mathsf{arg}_1\ \mathsf{sameSuffix(4)}\ \mathsf{arg}_2 \rangle$ hold. The above five relations defined are sufficient to cover the binary relations of string arguments proposed in existing approaches, which are defined for modeling the binary relations of arguments in a single program, such as isWithinDir, hasSameDirAs, hasSameExtensionAs [2].

For *integer* arguments, we use a polynomial equation to represent the relation of the two arguments. That is, let $x = \mathsf{arg}_1$ and $y = \mathsf{arg}_2$ (or $x = \mathsf{arg}_2$ and $y = \mathsf{arg}_1$), the following equation holds:

$$y = c_m x^m + c_{m-1} x^{m-1} + ... + c_1 x + c_0 \tag{2}$$

For example, for the two `malloc` calls which create a memory region to store the `uri` string parsed from the same request, the parameter values of these two `malloc` could have the form $y = 1 \cdot x + c_0$. The value of c_0 may not be 0 because the internal structures which store the `uri` are different in these two programs. Note that in Equation (2), when $c_1 = 1$ and $\forall i \neq 1, c_i = 0$, then $\mathsf{arg}_1 = \mathsf{arg}_2$. In our model, this equal relation between numeric arguments is able to capture most relations of flag arguments (such as `O_RDONLY` and `O_RDWR`), because they usually appear as the same in the diverse software providing semantically-close functionalities.

Polynomial relation does not cover all the binary relations between two integer arguments, e.g., exponential relation or bitwise relation may also exist under some circumstances. In our current model, we only preserve polynomial relation for integer parameters as it is the most common relation we observed in real applications.

2.3 Training Algorithms

The training procedure can be generally divided into three stages: argument pair extraction, relation acquisition and relation refinement.

Argument Pair Extraction. In this first stage, our purpose is to extract a set of $\mathsf{Value_{p,q}}$ for each pair of $\langle \mathsf{arg_{1,p}}, \mathsf{arg_{2,q}} \rangle$. Each $\mathsf{Value_{p,q}}$ set will contain all the value pairs occurred in the whole training procedure. All the sets of $\mathsf{Value_{p,q}}$ will then be used to train the relation R between $\langle \mathsf{arg_{1,p}}, \mathsf{arg_{2,q}} \rangle$. The algorithm of extracting each pair of arguments and its corresponding values are given in Algorithm 1, after which a set $\mathbb{PV}$= $\{(\mathsf{arg_{1,p}}, \mathsf{arg_{2,q}}, \mathsf{Value_{p,q}})\}$ will be collected. This $\mathbb{PV}$ set will then be used as input in Algorithm 2 and Algorithm 3.

Algorithm 1. Argument-pair extraction

```
1: for each (S1, S2) pair in the training set do
2:   for each s1,j in S1 and each s2,k in S2 do
3:     if comparable(s1,j, s2,k) then
4:       for each arg1,p belonging to s1,j, and each arg2,q belonging to s2,k do
5:         v1 = value of arg1,p
6:         v2 = value of arg2,q
7:         if (arg1,p.type = arg2,q.type) then
8:           if (arg1,p, arg2,q, Value_p,q) already exists in PV then
9:             add (v1, v2) to Value_p,q if (v1, v2) ∉ Value_p,q
10:          else
11:            Value_p,q = {(v1, v2)}
12:            add (arg1,p, arg2,q, Value_p,q) to PV
13:          end if
14:        end if
15:      end for
16:    end if
17:  end for
18: end for
```

This step is critical to the rest of the training procedure. The amount of all the combinations of $\langle \mathsf{arg_{1,p}}, \mathsf{arg_{2,q}} \rangle$ could be huge, however, we only consider argument pairs which appear in `comparable` function calls (as shown in line 3 of Algorithm 1). We define *comparable* function calls as those who have the same function names or whose functionalities are semantically related. For example, system calls `open` and `stat64` are comparable, and library calls `malloc`, `calloc` and `realloc` are comparable. System calls like `setuid` and `open` are not comparable since their functionalities are not semantically related. Our current implementation of Algorithm 1 reads in a configuration file that specifies which function calls are comparable. This configuration file is carefully constructed according to the platform on which the target applications are running. Our current implementation only considers the Linux operating system with GNU C library.

Relation Acquisition. The next step is to learn the relations between each pair of arguments gained by Algorithm 1. Here we introduce two algorithms for learning the relations: Algorithm 2 is used to learn the relations between two *string* arguments; while Algorithm 3 is for *integer* arguments. We use $\emptyset$ to denote that there is no relation between two arguments ($\mathsf{arg_1}$ $\emptyset$ $\mathsf{arg_2}$).

Algorithm 2. String-relation learning

Require: set $\mathbb{PV}$.

1: **for** each $(\mathsf{arg}_{1,\mathsf{p}}, \mathsf{arg}_{2,\mathsf{q}}, \mathsf{Value}_{\mathsf{p,q}})$ in $\mathbb{PV}$ **do**
2: **if** $\mathsf{arg}_{1,\mathsf{p}}.\mathsf{type} = \mathsf{arg}_{2,\mathsf{q}}.\mathsf{type} = \textit{string}$ **then**
3: **for** each $(\mathsf{v}_1, \mathsf{v}_2)$ in $\mathsf{Value}_{\mathsf{p,q}}$ **do**
4: calculate $\mathsf{R} \in \{\mathsf{equal}, \mathsf{samePrefix(n)}, \mathsf{sameSuffix(n)}, \mathsf{contain}, \mathsf{partOf}, \emptyset\}$, which satisfies $\mathsf{v}_1\ \mathsf{R}\ \mathsf{v}_2$.
5: **if** $\mathsf{R} \neq \emptyset$ **then**
6: **for** each $\mathsf{R_c}$ that $\langle \mathsf{arg}_{1,\mathsf{p}}, \mathsf{R_c}, \mathsf{arg}_{2,\mathsf{q}} \rangle \in \mathbb{R}$ **do**
7: **if** R conflicts with $\mathsf{R_c}$ **then**
8: remove all $\langle \mathsf{arg}_{1,\mathsf{p}}, \mathsf{R_c}, \mathsf{arg}_{2,\mathsf{q}} \rangle$ in $\mathbb{R}$
9: add $\langle \mathsf{arg}_{1,\mathsf{p}}, \emptyset, \mathsf{arg}_{2,\mathsf{q}} \rangle$ to $\mathbb{R}$
10: **else**
11: add $\langle \mathsf{arg}_{1,\mathsf{p}}, \mathsf{R}, \mathsf{arg}_{2,\mathsf{q}} \rangle$ to $\mathbb{R}$
12: **end if**
13: **end for**
14: **else if** $\langle \mathsf{arg}_{1,\mathsf{p}}, \emptyset, \mathsf{arg}_{2,\mathsf{q}} \rangle \notin \mathbb{R}$ **then**
15: add $\langle \mathsf{arg}_{1,\mathsf{p}}, \emptyset, \mathsf{arg}_{2,\mathsf{q}} \rangle$ to $\mathbb{R}$
16: **end if**
17: **end for**
18: **end if**
19: **end for**

Note that there is an update procedure in the learning process of Algorithm 2 for the relation of $\mathsf{samePrefix(n)}$ and $\mathsf{sameSuffix(n)}$, which is not shown in the algorithm. Take $\mathsf{samePrefix(n)}$ for example, suppose the existing relation for $\mathsf{arg}_1, \mathsf{arg}_2$ in $\mathbb{R}$ is $\mathsf{samePrefix}(\mathsf{n}_{\mathsf{old}})$ and the new learnt relation is $\mathsf{samePrefix}(\mathsf{n}_{\mathsf{new}})$. The new relation of $\mathsf{arg}_1, \mathsf{arg}_2$ in $\mathbb{R}$ will be updated as $\mathsf{samePrefix}(\mathsf{min}(\mathsf{n}_{\mathsf{old}}, \mathsf{n}_{\mathsf{new}}))$.

Another important detail not shown in Algorithm 2 is that, a threshold N can be set for the relations $\mathsf{samePrefix(n)}$ and $\mathsf{sameSuffix(n)}$, to reduce the false positives caused by small n. During learning, if the calculated $n < N$, then set $\mathsf{R} = \emptyset$. And different N should be assigned for $\mathsf{samePrefix(n)}$ and $\mathsf{sameSuffix(n)}$. Also note that a set of confliction rules for the relations is needed in Algorithm 2 (at line 7). Generally, $\emptyset$ conflicts with other relations, and equal, $\mathsf{contain}$, partOf conflict with each other since the equal relation will always be verified first.

In Algorithm 3, the given order m should be at least 2, and should not be too large so as to avoid the overfitting problem. m can also be dynamically adjusted according to the size of each $\mathsf{Value}_{\mathsf{p,q}}$. However, the value of m should be at most $\mathsf{Value}_{\mathsf{p,q}}.\mathsf{size} - 1$ in order to have enough value pairs for solving the equation set and leave at least one value pair to verify the results.

The whole learning process is optimized by utilizing the $\emptyset$ relations. The $\mathbb{PV}$ set does not need to be fully computed before running Algorithm 2 and Algorithm 3. If $\langle \mathsf{arg}_{1,\mathsf{p}}\ \emptyset\ \mathsf{arg}_{2,\mathsf{q}} \rangle$ already appears in $\mathbb{R}$, then the remaining instances of $\langle \mathsf{arg}_{1,\mathsf{p}}, \mathsf{arg}_{2,\mathsf{q}} \rangle$ do not need to be added into $\mathbb{PV}$. The $\emptyset$ relations will be dropped at the end of the training.

Algorithm 3. Integer-relation learning

Require: set $\mathbb{PV}$, order m.

for each $(\mathsf{arg}_{1,\mathsf{p}}, \mathsf{arg}_{2,\mathsf{q}}, \mathsf{Value}_{\mathsf{p,q}})$ in $\mathbb{PV}$ **do**
  if $\mathsf{arg}_{1,\mathsf{p}}.\mathsf{type} = \mathsf{arg}_{2,\mathsf{q}}.\mathsf{type} = \textit{integer}$ **then**
    if $\mathsf{Value}_{\mathsf{p,q}}.\mathsf{size} < m$ **then**
      add $\langle \mathsf{arg}_{1,\mathsf{p}}, \emptyset, \mathsf{arg}_{2,\mathsf{q}} \rangle$ to $\mathbb{R}$
    else
      use the first m pairs of $(\mathsf{v}_1, \mathsf{v}_2)$ in $\mathsf{Value}_{\mathsf{p,q}}$ to solve the equation set of Equation (2) to get $(c_m, ..., c_0)$, for both $(x = \mathsf{arg}_{1,\mathsf{p}}, y = \mathsf{arg}_{2,\mathsf{q}})$ and $(x = \mathsf{arg}_{2,\mathsf{q}}, y = \mathsf{arg}_{1,\mathsf{p}})$.
      if the equation set is solvable **then**
        $\mathsf{R} = \{x, y, (c_m, ..., c_0)\}$
        for each $(\mathsf{v}_1, \mathsf{v}_2)$ left in $\mathsf{Value}_{\mathsf{p,q}}$ **do**
          if Equation (2) does not hold **then**
            $\mathsf{R} = \emptyset$
          end if
        end for
        add $\langle \mathsf{arg}_{1,\mathsf{p}}, \mathsf{R}, \mathsf{arg}_{2,\mathsf{q}} \rangle$ to $\mathbb{R}$
      else
        add $\langle \mathsf{arg}_{1,\mathsf{p}}, \emptyset, \mathsf{arg}_{2,\mathsf{q}} \rangle$ to $\mathbb{R}$
      end if
    end if
  end if
end for

2.4 Model Refinement

In this subsection, we include an additional training phase to refine the relations we have obtained by the above algorithms. The relations $\mathbb{R}$ gained by using previous algorithms are patterns on the values we observed. However, certain trained relations may be due to the coincidence in the training data set, which could cause false alarms in detection. Thus, it will be better if we can remove those trained patterns in $\mathbb{R}$ which are not caused by the semantic relations between the two diverse applications.

However, it is not an easy task to validate the semantic relations of arguments and refine the trained model. Even with the source code, it is difficult for a human to capture the exact semantic meaning of a given function in a complex application. Thus, to automatically capture the semantic meanings of functions without the source code is an even harder problem. One way of learning the semantic relations between arguments is to use taint analysis [22]. Since the semantics of different set of function calls vary a lot, the detailed method of carrying out taint analysis needs to be customized accordingly. It is difficult to design a universal solution to perform the taint analysis for all the function calls.

In our current work, we develop a method of mapping memory management library calls (such as `malloc`, `free`, `realloc`, etc.) of two diverse web servers, according to the semantics gained by taint analysis. The basic idea is as follows: First of all, by tainting the request stream sent from client, we gain the

knowledge that which portions of the request are mapped to which heap memory regions. Since these memory regions are created by the corresponding memory library calls, each library call can be correlated with a certain portion of the request. We mapped the two memory library calls (e.g., one `malloc` in Apache and one `calloc` in Lighttpd) whose memory regions store the same part of the request (e.g. the `uri`). We then preserve the argument relations that belong to the mapped library calls, and remove other unmapped relations from $\mathbb{R}$. The implementation detail is given in Section 3, the effect of such refinement will be further evaluated in Section 4.

2.5 Detection

After the relation set $\mathbb{R}$ is trained, the detection phase is quite straightforward. During detection, for each argument pair $(\mathsf{arg}_{1,\mathsf{p}}, \mathsf{arg}_{2,\mathsf{q}})$ appears in $\mathbb{R}$, each instance of $(\mathsf{arg}_{1,\mathsf{p}} = \mathsf{v}_\mathsf{x}, \mathsf{arg}_{2,\mathsf{q}} = \mathsf{v}_\mathsf{y})$ will be tested. If an instance does not satisfy the corresponding $\langle \mathsf{arg}_{1,\mathsf{p}}\ \mathsf{R}\ \mathsf{arg}_{2,\mathsf{q}} \rangle$ in $\mathbb{R}$, the IDS will raise an alarm. Although the complexity of the training is relatively high, the detection only involves simple and fast computation. The main cost of detection depends on the cost of monitoring and logging the function calls.

3 Implementation

We have implemented our approach on Ubuntu 8.04 (Linux kernel 2.6.24). The implementation consists of two online components and an offline component.

The two online components are both monitor modules (referred to as tracer), one of which is used to trace system calls, the other is used to trace library calls of the monitored programs. For the system call tracer, we utilize `ptrace` to intercept each system call made by the monitored program and log the following information: (a) the PC value from where the system call was invoked, (b) values of arguments, and (c) the call stack information which contains a set of absolute return addresses. For the library call tracer, we modify the GNU C library (glibc) under Ubuntu to output similar information for a selected set of library calls. Since the `backtrace` method cannot be used within the implementation of some library calls such as `malloc`, we implement our own `backtrace` method in the glibc to log the call stack information.

Each time when the monitored program starts, all the base addresses of its loaded shared libraries are also recorded, which is retrieved from corresponding `/proc/[pid]/maps`. These addresses will be used to convert the absolute addresses in the call stack recorded by the tracer to relative addresses, in the form of `[libname+offset]`. By having relative call stacks, we are able to identify the same instance of function call across different runs of the same program.

The offline component of our implementation includes the parsers of the logged traces and the training module that implements the algorithms in Section 2. As mentioned earlier, a configuration file is also provided to the training module, which specifies the function calls that are comparable. The implementation of the offline component is about 3.5K LOC.

For the model refinement part in the training, we utilize TEMU [1] to carry out the taint analysis. Web server programs running in TEMU are provided with tainted request stream and tainted local disk files, and the instructions of the monitored web server will be recorded when processing each request. The recorded instruction traces are then translated by the `trace_reader` tool in Vine [1] and used as inputs to the trace parsers we implemented. According to the taint information in the trace files, our trace parser will be able to extract the information that each memory library calls is related to which part of the request stream (or is related to which file on the disk). Then two library calls (in two diverse servers) which are related to the same part of the request (or the same local file) are recorded as the mapped library calls as mentioned in the previous section. This TEMU trace parser is around 1K LOC.

4 Evaluation

In this section, we first investigate the effectiveness of our approach in detecting real attacks and then analyze the false alarm rates. Performance overheads for intrusion detection are also discussed. All experiments are conducted under Ubuntu 8.04 and the training and testing are performed in offline mode.

4.1 Detection Effectiveness

Since the code injection attacks have been extensively addressed in prior research [7,8,9,13,14,19,23], we focus on evaluating the detection effectiveness of our model against attacks on security-critical data utilizing erratic arguments. Table 1 lists the set of attacks tested in our evaluation. The first two attacks in Table 1 are detectable by our approach since they both violate the *string* argument relations trained in our model, while the other two attacks in Table 1 violate the *integer* argument relations.

Table 1. Selected Non-control-flow Attacks

Reference	Vulnerable Program	Attack Description	Alternative Program	Detected? (type)
S.Chen et al. [4]	Ghttpd	stack overflow to overwrite filename data	Null-httpd	Yes (string)
S.Chen et al. [4]	Null-httpd	heap overflow to corrupt cgi-bin configuration string	Ghttpd	Yes (string)
S.Chen et al. [4]	Wu-ftpd	format string attack to overwrite userid data	Pure-ftpd	Yes (integer)
CVE-2008-4298	Lighttpd	memory leak via duplicate request headers	Cherokee httpd	Yes (integer)

Detection of Anomalous String Arguments.

The first attack in Table 1 exploits a stack overflow vulnerability in Ghttpd's logging function [4], which occurs in the following code fragment in function `serverconnection()`:

```
1: if (strstr(ptr, "/.."))
2:   reject the request;
3: log(...);
4: if (strstr(ptr, "cgi-bin"))
5:   execve(ptr, ...)
```

In the above code, `ptr` is a char pointer to the string of URL requested by a remote client. The first two lines in the code are used to check the absence of "`/..`" in the URL, before the CGI request is parsed and handled in line 4–5. The stack buffer overflow vulnerability is in function `log()`, where a long user input string can overrun a 200-byte stack buffer. Chen et al. [4] managed to construct a stealthy attack which changes `ptr` to point to a string `cgi-bin/../../../../bin/sh` by exploiting the vulnerability in `log()`. Their attack neither injects code nor alters the return address, thus, it is difficult to be detected by most of existing models.

Our approach is able to detect this attack. During training, our model learns the `equal` relation between the first parameter of `execve` in Ghttpd and the parameter of corresponding `execve` in Null-httpd (in function `cgi_main()`). Since this relation is later violated when this attack has successfully changed the value of `ptr` in Ghttpd, an alarm is raised by the IDS.

Although this attack is also detectable by the dataflow model [2], their mechanism is different. Their system first learns that all files executed at line 5 should be within the `"cgi-bin"` directory. The attack is detected when it accesses a file outside this directory. However, such `isWithinDir` [2] relation (trained by monitoring the program itself) may not be sufficient in practical scenarios. For example, in typical business applications, files under the same directory may have different access policies. A user x is only allowed to execute program A under the directory, but not program B. Due to the overflow attack, adversary with the privilege of user x is able to gain the access to program B. Under such a scenario, the `isWithinDir` relation will not be able to detect such attacks since all the programs are under the same directory, while our model is still able to detect attacks in cases like these.

The second attack in Table 1 targets on a heap overflow vulnerability exists in Null-httpd. This vulnerability is triggered when a special POST command is received by the server. This vulnerability can be used to corrupt the CGI-BIN configuration of Null-httpd and will result in root compromise without executing any external code. In the attack illustrated by Chen et al. [4], two POST commands are issued to precisely overwrite four characters in the CGI-BIN configuration so that it is changed from `"/usr/local/httpd /cgi-bin\0"` to `"/bin\0"`. After the corruption, `/bin/sh` can be started as a CGI program and any shell command can be sent as the standard input to `/bin/sh`.

This attack cannot be easily detected by control-flow schemes [7,8,9,13,14,19,23], and is not addressed by the dataflow scheme [2]. However, our diversity model is able to detect such an intrusion due to the same reason in the first attack – the equal relation (of the first parameter of the two `execve` calls in Null-httpd and in Ghttpd) learnt during training, is violated when Null-httpd is exploited.

Note that although both of these two servers (Ghttpd and Null-httpd) have vulnerabilities, we can still use them together to build our diversity detection model because their vulnerabilities are not exploitable by the same attack code. In general, the probability that the same vulnerability exists in two diverse applications providing semantically-close functionalities is very low [15].

Detection of Anomalous Integer Arguments.

The third attack in Table 1 exploits a format string vulnerability in Wu-ftpd. The vulnerable code fragment is within the `getdatasock()` function:

```
1: seteuid(0);
2: setsockopt( ... );
   ...
3: seteuid(pw->pw_uid);
```

The above function is invoked when a user issues data transfer commands, such as downloading or uploading a file. It requires root privilege in order to perform the `setsockopt()` operation. Thus, the privilege is temporarily escalated using `seteuid(0)` and then changed back by the second `seteuid()`. The data structure `pw->pw_uid` is a cached copy of the user ID saved on the heap. The attack proposed in [4] exploits the format-string vulnerability to change `pw->pw_uid` to 0, which maintains the root privilege for the attacker so that arbitrary files can be uploaded and downloaded by the attacker as a root user.

Our model detects this attack when monitoring Wu-ftpd together with Pure-ftpd. Since the two servers have the same configurations, the parameter of `seteuid()`[1] function call on line 3 in Wu-ftpd always has the same value as the parameter of the `seteuid()` calls in function `doport3()` in Pure-ftpd. These *integer* parameter relations are violated when the adversary overflow the heap to change `pw->pw_uid` to 0.

The fourth attack in Table 1 exploits a memory leak vulnerability exists in Lighttpd. When a duplicated field appears in a request header (e.g., "`User-Agent :Mozilla/4.0`" and "`User-Agent:MSIE/8.0`" both appear in the header), the `http_request_parse()` method in Lighttpd will allocate a memory region to store the content of the second field (i.e., `MSIE/8.0`), but will not recycle this resource afterwards. An adversary can utilize this vulnerability to consume the memory of the server running Lighttpd by sending many requests with duplicate fields (with a maximum field length of 2KB).

[1] The underlying system call invoked is setresuid32().

Such Denial-of-Service attack cannot be directly detected by the existing approaches which train on a single server, especially when the total memory consumed is not large enough to cause any exception. The difficulty comes from the memory management behaviors of these web servers. For the most commonly used servers (such as Apache, Lighttpd, etc.), the allocated memory will be reused in processing the following requests and never be explicitly freed. Thus, for both normal request and attack request processing, only memory allocation methods (such as `malloc`, `realloc` ...) are observed, no deallocation method (such as `free`) will appear in the library call sequences obtained. This makes it difficult for an IDS to precisely model the memory behaviors, as it requires simulating the complex internal memory management of these server applications.

Our diversity IDS is able to learn the integer argument relations of the corresponding memory allocation calls in the two servers monitored. To be specific, the IDS learns that 16 pairs of the parameter values to the `malloc` and `realloc` calls of Lighttpd and Cherokee servers are equal or have fixed difference (which is actually due to the size difference of the internal structures in these two servers). In the detection phase, the IDS detects the memory leak attack immediately when the attack request causes one of Lighttpd's `malloc` parameter to increase (in `buffer_copy_string_len()` invoked by `http_request_parse()`), which violates the integer relations that have been trained in the model.

4.2 False Alarm Analysis

There are three pairs of programs in Table 1. All of them are used to evaluate the false alarm rates of our approach, as shown in Table 2. Two pairs of them are http servers (Lighttpd and Cherokee, Ghttpd and Null-httpd), which are configured to hold the same content of the web site of our university. In the training phase, the two web servers in the same pair are provided with the same series of requests (10K requests) obtained from the real log of our university's web server. In the detection phase, another set of requests (50K requests) from the logs are sent to these servers to evaluate the false alarm rates. Applications in the third pair are FTP server programs (Wu-ftpd and Pure-ftpd). Since we do not have the access to the log of large amount of real FTP requests, we configure these two FTP servers to hold the files downloaded from GNU FTP[2], and simulate the requests by randomly issuing commands (such as `put`, `get`, `dir`, `passive`, `type`, etc.) for random files or directories on the servers.

We construct two different experiments to test our false alarm rates (as shown in Table 2 and Table 3). The first experiment only focuses on monitoring the system calls and their arguments so that it can be compared with existing approaches which also utilize system call arguments [2,17] (e.g., the result of the dataflow model [2] shows the false positive rate of the tested HTTP server is 64.12×10^{-5}, and the rate for SSH server is 0.02×10^{-5}). Note that the rates shown in Table 2 are "raw" false alarm rates, i.e., the fraction of system calls that caused violations, without combining the same type of violations. For

[2] GNU Software FTP server, `ftp.gnu.org/gnu`

Table 2. False alarm rate

Diverse Programs		Training Trace # of **Sys calls** ($\times 10^5$)	Detection Trace # of **Sys calls** ($\times 10^5$)	False alarm rates ($\times 10^{-5}$)
Pair 1	Lighttpd	2.29	10.90	0.826
	Cherokee httpd	5.19	24.35	
Pair 2	Ghttpd	7.24	39.51	1.948
	Null httpd	20.62	98.57	
Pair 3	Wu ftpd	10.78	54.15	0.617
	Pure ftpd	4.37	12.96	

Table 3. Model refinement by taint analysis

Programs	Training Trace # of **Lib calls** ($\times 10^5$)	Detection Trace # of **Lib calls** ($\times 10^5$)	False alarm rates ($\times 10^{-5}$)	
			Original	After Refine
Lighttpd	2.31	11.06	5.286	1.762
Cherokee	0.46	2.27		

example, the false alarm rate for Lighttpd in Table 2 is 0.826×10^{-5}, which means that one false alarm will be raised for every 100K system calls processed. This indicates that one out of 10K requests will cause false alarms, as on average 10.9 system calls are invoked to process one request for Lighttpd.

The results show that the second pair of applications have much higher false alarm rate than the other two pairs, as in Table 2. We investigated the reason for this higher false alarm rate, and found that this is due to the fact that during the training, there are several coincident contain relations for the string arguments between Ghttpd and Null-httpd, which are violated in the detection phase for benign requests. Our current implementation of the training algorithm regards two string arguments as contain as long as their values satisfy this relation, even if these pair of arguments only appear once in the training. However, some rules in the training phase could be added to further decrease the false alarm rate. For example, any string relations should have at least two instances of value pairs in the training phase so that one instance of values is used to set up the relation and other values can be used to validate the relation in the training (and any argument pairs which only have one instance should be regarded as $\emptyset$ relation in $\mathbb{R}$). Such modification could reduce the false positives of our model but should be carefully designed so that it would not decrease the detection capability as well. Investigation on this trade-off is left as future work.

In the second experiment (as shown in Table 3), we investigate the false positive rate when our model monitors the memory management library calls of the diverse applications. Note that different from Table 2, only library calls are considered in Table 3. We further investigate the effectiveness on false positive reduction by refining our model using taint analysis. The result shows that after removing the library call argument patterns which are not mapped by the semantic relations, the false positive rate decreases. It is possible to refine the

Table 4. Program size and model size

Programs		Program Size (Kbytes)	String Relations	Integer Relations
Pair 1	Lighttpd	767.9	143	367
	Cherokee httpd	1165.7		
Pair 2	Ghttpd	43.6	120	342
	Null httpd	34.3		
Pair 3	Wu ftpd	385.3	171	496
	Pure ftpd	87.8		

relations of other arguments by using taint analysis. However, since the semantics of different set of library/system call arguments vary, taint analysis needs to be carefully customized accordingly.

4.3 Performance Overheads

Table 4 shows the size of the programs used in our evaluation, along with the model sizes in terms of the number of relations learnt. Note that the sizes of the programs in the first pair include some of their own shared libraries. This is because part of the functionalities of these servers are compiled as shared libraries in default (e.g., many of the commonly used functions in cherokee are compiled in `libcherokee-base.so` and `libcherokee-server.so`), which is different from standalone programs. It can be seen from the table that the size of our models are relatively small compared to the sizes of the programs.

We also studied the time cost of our model for both learning and detection phases, which is illustrated in Table 5. The original size of the training traces were between 110MB and 526MB, consisting of 0.2 to 2 million system calls. As shown in Table 5, we measure the performance overheads of monitoring the system calls and library calls, which is the dominate overhead during detection. It shows that the overheads of monitoring system calls could be quite high for web servers (up to 83.4%). The overhead is mainly due to our system call tracer. As explained in Section 3, our monitor module utilizes `ptrace` for system call interception with our own implementation of the `backtrace` which records the call stack information of each system call. Similar overhead was also reported by existing approach [2] using `ptrace`. This cost can be reduced to less than 6% [9], by a kernel implementation of the interceptor.

Table 5. Training time and detection overhead

Programs	Training time	Detection Overheads	
		Monitoring sys calls	Monitoring lib calls
Lighttpd & Cherokee	93.8 sec	29.10%	18.38%
Ghttpd & Null-httpd	1620.9 sec	83.39%	11.41%
Wu-ftpd & Pure-ftpd	2091.3 sec	17.56%	1.37%

5 Related Work

In this section, we summarize the related work from two perspectives: one is traditional intrusion detection schemes, the other is diversity-based detection schemes.

Traditional intrusion detection techniques [5,7,8,9,13,14,19,23,26] mainly focus on utilizing only system call sequences to detect code injection attacks. Recent works [2,17,18,20,24] further incorporate system call argument information to defend against attacks which do not modify control flows. However, these approaches have difficulties in deciding which legitimate argument value is really benign, when multiple legitimate values appear in the training phase.

Early works on software diversity construct intrusion tolerance systems [3,21] with software providing semantically-close functionalities. This architecture is then utilized for developing diversity-based intrusion detection techniques [6,10,11,16,25]. Most of these techniques use Commercial Off-The-Shelf (COTS) software to build the detection models. Among those schemes, the techniques proposed by Just et al. [16] and Totel et al. [25] are output voting schemes, which only compare the final outputs (HTTP status codes and files) of the diverse software to detect intrusions. However, as many of the intrusions may not result in observable deviation in the responses of those server software, such intrusions can evade detections of these techniques.

Behavioral Distance model by Gao et al. [10,11] was later proposed to defend against stealthy attacks which are not addressed by both the output voting schemes and traditional intrusion detection techniques which only monitor single application. However, since hidden Markov model used in their scheme (to train the normal-behavior profiles of the system call sequences) is only able to handle finite states, their model cannot be simply extended to detect attacks utilizing erratic arguments.

Our approach is the first work that captures underlying semantic correlation of the argument values in diverse programs. Our model gains more accurate context information compared to existing schemes. Such context information is critical in detecting sophisticated attacks on security-critical data utilizing erratic arguments. When deployed, our model can be combined with the existing system call sequence or control flow models to defend against a wider range of attacks.

6 Conclusions

In this paper, we propose an anomaly detection model to detect erratic-argument attacks which are recognized as normal inputs by the existing techniques. Our approach utilizes the function arguments of two diverse applications which provide semantically-close functionalities. Different from existing techniques, our model learns the relations of the function arguments between the two applications, which naturally captures more accurate context information. In the evaluation, we show that our model is able to detect real attack manipulating the value of

erratic arguments, with a moderate false alarm rate. The main limitation of our scheme is the additional cost on the management of diverse software. However, such a cost could be negligible for some existing fault-tolerant system where diverse software have already been deployed to prevent simultaneous failure.

References

1. TEMU and Vine. The BitBlaze Dynamic Analysis Component, `http://bitblaze.cs.berkeley.edu`
2. Bhatkar, S., Chaturvedi, A., Sekar, R.: Dataflow anomaly detection. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy, pp. 48–62 (2006)
3. Chen, L., Avizienis, A.: N-version programming: A fault-tolerance approach to reliability of software operation. In: Digest of 8th International Symposium on Fault-Tolerant Computing (FTCS), pp. 3–9 (June 1978)
4. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: Proceedings of the 14th Conference on USENIX Security Symposium, p. 12 (2005)
5. Lam, L.C., Chiueh, T.-c.: Automatic Extraction of Accurate Application-Specific Sandboxing Policy. In: Jonsson, E., Valdes, A., Almgren, M. (eds.) RAID 2004. LNCS, vol. 3224, pp. 1–20. Springer, Heidelberg (2004)
6. Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., Hiser, J.: N-variant systems: a secretless framework for security through diversity. In: Proceedings of the 15th Conference on USENIX Security Symposium (2006)
7. Feng, H.H., Kolesnikov, O.M., Fogla, P., Lee, W., Gong, W.: Anomaly detection using call stack information. In: Proceedings of the 2003 IEEE Symposium on Security and Privacy (2003)
8. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A sense of self for unix processes. In: Proceedings of the 1996 IEEE Symposium on Security and Privacy, p. 120 (1996)
9. Gao, D., Reiter, M.K., Song, D.: Gray-box extraction of execution graphs for anomaly detection. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, pp. 318–329 (2004)
10. Gao, D., Reiter, M.K., Song, D.: Behavioral Distance for Intrusion Detection. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 63–81. Springer, Heidelberg (2006)
11. Gao, D., Reiter, M.K., Song, D.: Behavioral Distance Measurement Using Hidden Markov Models. In: Zamboni, D., Kruegel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 19–40. Springer, Heidelberg (2006)
12. Gao, D., Reiter, M.K., Song, D.: Beyond output voting: Detecting compromised replicas using HMM-based behavioral distance. IEEE Transactions on Dependable and Secure Computing (TDSC) (July 2008)
13. Ghosh, A.K., Schwartzbard, A.: A study in using neural networks for anomaly and misuse detection. In: Proceedings of the 8th Conference on USENIX Security Symposium, p. 12 (1999)
14. Giffin, J.T., Jha, S., Miller, B.P.: Efficient context-sensitive intrusion detection. In: Proceedings of the Network and Distributed System Security Symposium (2004)
15. Han, J., Gao, D., Deng, R.H.: On the Effectiveness of Software Diversity: A Systematic Study on Real-World Vulnerabilities. In: Flegel, U., Bruschi, D. (eds.) DIMVA 2009. LNCS, vol. 5587, pp. 127–146. Springer, Heidelberg (2009)

16. Just, J.E., Reynolds, J.C., Clough, L.A., Danforth, M., Levitt, K.N., Maglich, R., Rowe, J.: Learning Unknown Attacks - A Start. In: Wespi, A., Vigna, G., Deri, L. (eds.) RAID 2002. LNCS, vol. 2516, pp. 158–176. Springer, Heidelberg (2002)
17. Kruegel, C., Mutz, D., Valeur, F., Vigna, G.: On the Detection of Anomalous System Call Arguments. In: Snekkenes, E., Gollmann, D. (eds.) ESORICS 2003. LNCS, vol. 2808, pp. 326–343. Springer, Heidelberg (2003)
18. Maggi, F., Matteucci, M., Zanero, S.: Detecting intrusions through system call sequence and argument analysis. IEEE Transactions on Dependable and Secure Computing (TDSC) 7, 381–395 (2010)
19. Michael, C.C., Ghosh, A.: Simple, state-based approaches to program-based anomaly detection. ACM Transactions on Information and System Security (TISSEC) 5(3), 203–237 (2002)
20. Provos, N.: Improving host security with system call policies. In: Proceedings of the 12th Conference on USENIX Security Symposium, p. 18 (2003)
21. Reynolds, J., Just, J., Lawson, E., Clough, L., Maglich, R.: The design and implementation of an intrusion tolerant system. In: Proceedings of the 2002 International Conference on Dependable Systems and Networks, DSN (2002)
22. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, pp. 317–331 (2010)
23. Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A fast automaton-based method for detecting anomalous program behaviors. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, p. 144 (2001)
24. Tandon, G., Chan, P.: Learning rules from system call arguments and sequences for anomaly detection. In: Workshop on Data Mining for Computer Security (2003)
25. Totel, E., Majorczyk, F., Mé, L.: COTS Diversity Based Intrusion Detection and Application to Web Servers. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 43–62. Springer, Heidelberg (2006)
26. Wagner, D., Dean, D.: Intrusion detection via static analysis. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, p. 156 (2001)