Dataflow Anomaly Detection*

Sandeep Bhatkar Abhishek Chaturvedi R. Sekar
Department of Computer Science
Stony Brook University, Stony Brook, NY 11794
{sbhatkar, abchatu, sekar}@cs.sunysb.edu

Abstract

Beginning with the work of Forrest et al, several researchers have developed intrusion detection techniques based on modeling program behaviors in terms of system calls. A weakness of these techniques is that they focus on control flows involving system calls, but not their arguments. This weakness makes them susceptible to several classes of attacks, including attacks on security-critical data, race-condition and symbolic link attacks, and mimicry attacks. To address this weakness, we develop a new approach for learning dataflow behaviors of programs. The novelty in our approach, as compared to previous system-call argument learning techniques, is that it learns temporal properties *involving the arguments of different system calls, thus capturing the flow of security-sensitive data through the program. An interesting aspect of our technique is that it can be uniformly layered on top of most existing control-flow models, and can leverage control-flow contexts to significantly increase the precision of dataflows captured by the model. This contrasts with previous system-call argument learning techniques that did not leverage control-flow information, and moreover, were focused on learning statistical properties of individual system call arguments. Through experiments, we show that temporal properties enable detection of many attacks that aren't detected by previous approaches. Moreover, they support formal reasoning about security assurances that can be provided when a program follows its dataflow behavior model, e.g.,* `tar` *would read only files located within a directory specified as a command-line argument.*

1. Introduction

Beginning with the work of Forrest et al [15], several researchers have developed intrusion detection techniques based on modeling behaviors of programs in terms of system call sequences [26, 21, 11, 31, 7, 14, 9, 19]. Approaches described in [15, 33, 34, 21, 26, 7, 9] learn models from program behaviors observed during a *training phase*, which is generally assumed to be free of attacks. Other approaches use static analysis of source code [31, 19] or binary [14, 6]. A variety of representations have been used for these models, beginning with strings [33, 34], to finite-state automata [26, 31, 21] and push-down automata [7, 14, 19, 9].

All of the above techniques have been shown to be effective against the most common types of attacks, which involve *execution of foreign code.* In addition, learning-based techniques are effective against attacks that exercise unusual code paths, e.g., a code path that is taken when a maliciously crafted input is processed by vulnerable code in a (buggy) server. However, an important weakness of the above approaches is their singular focus on control flows, with little emphasis on *data flows* involving system call arguments. This makes them susceptible to several classes of attacks:

- *Non-control-flow hijacking attacks.* [5] demonstrates attacks on several common servers that target security-critical data, e.g., variables that store the userid corresponding to an FTP client, the directory that contains all allowable CGI scripts for a web server, and so on.
- *Race condition attacks* [2] exploit TOCTOU errors, where the resource referenced by a name has changed between the time of check and time of use. Race attacks do not change the system calls made by a victim process, but only change the interpretation of their operands.
- *Mimicry attacks [32].* These are evasion attacks, where an attacker modifies an attack so that it closely mimics program's behavior as seen by the intrusion detection system (IDS). Recent work [32, 10, 17] has shown that such attacks can largely be generated in an automated way.

To detect these attacks, it is necessary to reason about system call arguments. Research in this direction has so far been focused on learning statistical properties of each system call argument in isolation [18, 28]. In contrast, we present an intrusion detection technique that is based on learning *temporal properties* involving arguments of different system calls, thus capturing the flow of security-sensitive data through the program. Specifically, we make the following contributions in this paper:

- We formulate a notion of *dataflow* properties of programs. Since properties are defined in terms of externally observable events (specifically, system calls), our formula-

*This research is supported in part by an ONR grant N000140110967, NSF grants CCF-0098154, CCR-0205376 and CNS-0208877, by Computer Associates, and NYSTAR.

tion cannot reason about *actual* dataflows that take place in a program; instead, it hypothesizes the flows that may be present, based on relationships observed between the parameters of different system calls. Our formulation is decoupled from control-flow models, but can leverage control-flow context to significantly improve the precision of dataflow models. Dataflow properties are categorized into *unary* relations that involve properties of a single system call argument, and *binary* relations that involve arguments of two different system calls.

Binary relations turn out to be powerful and versatile. They enable models to be parameterized, e.g., they can capture how a system call argument is derived from a command-line parameter or an environment variable. They enable detection of several recently reported stealthy attacks that would be undetectable without them. Binary relations also support formal reasoning about nontrivial security properties of programs.

- We present an efficient algorithm that can be layered on top of existing techniques for learning control-flow behaviors of programs. This algorithm provides a uniform way to enhance the precision of existing methods, including the N-gram [15], FSA [26], VtPath [7] and the execution graph [9] methods.
- We present a detailed experimental evaluation of the technique, establishing the following benefits:
  - *Attack detection.* We selected several attacks that are hard to detect using existing control-flow models, and show that they can be detected using our technique. Of particular significance in this regard are race attacks, which, to the best of our knowledge, have not been detected by previous program behavior based anomaly detection technique. Moreover, our technique can detect sophisticated attacks on security-critical data. We argue that attack detection is based on essential characteristics of the attack, and not the result of nonessential artifacts that can be easily changed.
  - *Formal assurances about security.* We show that the models are precise enough that interesting dataflow properties can be proved about them. This means that we can assert that we can detect *any attack* that violates these properties. We believe that ours is the first anomaly-based intrusion detection technique that can provide formal assurances involving substantive security properties for large programs.
  - *Model precision.* The precision of model in terms of *average branching factor* [31] improves by a factor between 10 to 10^5.
  - *False alarms.* False alarm rate for intrusion detection increased by a factor of 2 to 4 as a result of capturing argument information.

Paper Organization The rest of the paper is organized as follows. In Section 2, we define data flow properties that can be incorporated into control-flow behavior models. Section 3 describes an algorithm to learn dataflow relationships. Section 4 provides details on implementation. In Section 5, we evaluate the effectiveness of our approach. Section 6 discusses related work, followed by concluding remarks in Section 7.

2. Defining Data Flow Behavior

2.1. Events, Traces and Behavior Model

We formalize program behaviors in terms of externally observable events generated by a program. Since our interest is mainly confined to system call events in the rest of this paper, we will use the terms "event" and "system call" interchangeably. We begin with a series of definitions:

- An *execution trace* (or simply, a *trace*) for a program P, denoted $T(P)$, is the sequence of all the system calls executed by P during its execution. A trace typically includes information about system call arguments, and/or information about the program's runtime environment, e.g., the program location from where a system call is made ("PC" information).
- A *system call tracer* (or simply, a tracer) is responsible for intercepting and recording system calls made by P, thus generating $T(P)$.
- The *trained behavior* of P is the set $\mathcal{T}(P)$ of all traces generated by P during its training runs.
- A *behavior model* for P is an automaton that accepts traces. (Strictly speaking, a model accepts *prefixes* of traces, but for simplicity, we will say that the model accepts T.)

Given a training trace set $\mathcal{T}$, a model for these traces captures some of the essential properties of these traces. The properties captured will vary across different methods, as discussed next.

2.2. Behavior Models from Previous Research

- *N-gram method:* The property captured by this method is given by the set of all the substrings of $\mathcal{T}$ of length N. Only event names are captured in the model, but not the arguments. The model will accept another trace T', provided all of its substrings of length N are found in the training set $\mathcal{T}$.
- *FSA method:* For the FSA method, events are annotated with program location information. An event e invoked from location L is denoted $e@L$. The FSA method captures successor relationship between events within each trace in $\mathcal{T}$. A new trace T' will be accepted by the model if and only if every successive event pair $(e_1@PC_1, e_2@PC_2)$ in T' is also present in some trace

in $\mathcal{T}$. In addition, the first (last) event in T' must appear as the first (last) event in some trace in $\mathcal{T}$.

- *VtPath method:* This method also learns successor relationships similar to the FSA method, but it takes into account all of the function calls that were active at the time of invocation of the system call, as opposed to just the location of a system call. This call stack information is compactly summarized using a *VtPath*, which captures the difference between the call stacks observed at the time of occurrence of e_1 and e_2.
- *Execution graph method:* This method captures a further generalization of the relationship learnt by the VtPath method. In particular, suppose that there is a pair $(e_1, \mathbf{S}_1)$ and $(e_2, \mathbf{S}_2)$ of events in a trace $T \in \mathcal{T}$, where $\mathbf{S}_1, \mathbf{S}_2$ denote the return addresses on the stack at the point of invocation of these events. From this information, this method infers a sequence of calls, returns, and intra-procedural transitions that must have occurred in the program P in order to generate this sequence of events and call-stack information. Now, any trace T' whose event pairs can be constructed by composing the program transitions observed from all event pairs in $T \in \mathcal{T}$ are accepted by the model.

From the above description, we conclude that the above four methods capture:

(a) *all-trace* properties, i.e., properties that hold for every execution trace, and

(b) sequencing relationships among events, i.e., *control-flow properties* of programs.

Below, we describe how these methods can be extended to incorporate dataflow properties.

2.3. Dataflow Properties

By dataflow properties, we still refer to properties of execution traces, but these properties relate to the values of event argument data and their flow from one system call to a subsequent one. A natural approach for capturing properties of event arguments is using sets. For instance, we can use a set to specify all possible file name arguments to `open` system calls in any execution trace for a program P. Note that this approach combines information about all `open` system calls that appeared in any trace. Such a combination can lead to significant (and needless) losses in accuracy. To illustrate this, consider the following example program:

```
L1: fd1 = open("/etc/passwd", O_RDONLY);
... /* perform authentication */
L2: fd2 = open("/tmp/out", O_RDWR);
```

If we combined the information about all open operations together in the dataflow model, then a trace that corresponds to opening `/etc/passwd` file at line L2 will be accepted, while it is clearly inconsistent with normal program behavior. To improve accuracy, it is hence desirable to partition the `open` system calls into subsets, and capture properties of each subset separately. The question then is how to do this partitioning.

Making control-flow context available for learning dataflow properties. We make the important observation that control-flow models already distinguish among different occurrences of the same event in a trace. For instance, in the above example, the FSA model distinguishes between the two `open` system calls based on the PC value. More generally, the approaches described above use some sort of an automaton model: the N-gram model uses string-matching automata to recognize N-grams, while the FSA approach uses a finite-state automaton. The VtPath model can be thought of as recognizing VtPaths, which are strings over program locations. The execution graph method uses a push-down automaton, and hence its control state, possibly with some (bounded) information from the stack, can be used to distinguish event occurrences.

We use the term *control-flow context* to refer to the event context information that can be provided by a control-flow model. We use *labeled traces* to encode control-flow context into traces. This approach allows us to decouple dataflow properties from control-flow models.

Control-context is encoded in traces by giving names for event arguments: if the same event appears in two places in a labeled trace, and these two instances correspond to the same control-flow context, then (and only then) their arguments should have the same name. An example labeled trace is:

```
open@L1 X = "/etc/passwd" Y = "read"
open@L2 Z = "/tmp/out"    W = "write"
```

2.4. Possible Dataflow Relationships

Similar to control-flow models, our focus is on learning *all-trace* dataflow properties. These properties will be formulated as relationships on event arguments. It is natural to specify these relationships by referring to argument names in a labeled trace. We limit our attention to unary and binary relations on event arguments in this paper, as they can be learnt more efficiently, and seem adequate for our purposes.

Unary Relations. Unary relations capture properties of a single argument. They can all be represented using the form $X\ \mathbf{R}\ c$, where X is an argument name, $\mathbf{R}$ denotes a relation, and c is a constant value. Examples of unary relations include:

- **equal** relationship is applicable to all types of arguments. For instance X **equal** v indicates that the value of argument X is always v.
- **elementOf** relation is used to capture the fact that an argument can take one of several values. X **elementOf** $\mathcal{S}$ indicates that X can take any of the values in the set $\mathcal{S}$.

- **subsetOf** is a generalization of **elementOf**, and is used when an argument can take multiple values, all of which are drawn from a set. For instance, M **subsetOf** {`RD, WR`} represents the fact that M is a set-valued argument whose value is a subset of the set {`RD, WR`}.
- **range** is a relation involving integer arguments. It is characterized by a lower and upper bound, e.g., X **range** $(0, 2)$ indicates that the argument X can take values in the range $0 - 2$.
- **isWithinDir** relation is used to capture the fact that a file name argument is contained within a specified directory. For example, if X has the value `"/home/user/xyz"`, we can state X **isWithinDir** `"/home/user/"`.
- **hasExtension** relation is used to specify the fact that a file name argument has certain extensions, e.g., if X takes values from the set {`a.doc, b.doc, c.txt`}, then X **hasExtension** {`"doc", "txt"`} holds.

Some of the unary relations mentioned above incorporate some approximations. For instance, **elementOf** is an example of a relation that doesn't need to make approximations. If a variable X is observed to have k distinct values $v_1, ..., v_k$ across all traces, this can be captured as $X \in \{v_1, ..., v_k\}$. Approximations become necessary when the sets become large — with integer- or string-valued variables, sets can be unbounded in size. In this case, approximations such **range** or **isWithinDir** may be used.

We note that previous works on system call argument learning [18, 28, 24] were all focused on unary relations. The main difference with our approach is that we suggest the use of control-flow contexts to support more accurate learning. As a result, our approach learns properties that hold across all occurrences of a system call made from the same control context, whereas the previous approaches learn properties that hold across all occurrences of a system call, regardless of control context.

Binary Relations. Binary relations capture relationships between two event arguments. These may be arguments of the same event, or arguments of different events. Our focus is mainly on the latter, since such relationships naturally capture the flow of data from the arguments of one system call to another.

Binary relations can in general be represented using sets in a manner analogous to unary relations. However, such an approach will limit the method to finite (small) relations, or require approximations that lose important information. In practice, we often need to represent some types of relations over large domains without significant loss of information. Examples of such relationships include:

- **equal** captures equality between system call operands, e.g., the file descriptor returned by an open operation equals the first argument of a subsequent write operation.
- **isWithinDir** indicates that one file name argument is within the directory named by the other argument.
- **contains** is the reverse operation of **isWithinDir**, in which the second file name argument is within the directory named by the first argument.
- **hasSameDirAs** indicates that two arguments have a common base directory, e.g., if $X =$ `"/home/user1/xyz"` and $Y =$ `"/home/user2/abc"`, X **hasSameDirAs** Y and Y **hasSameDirAs** X hold, with the common directory being `"/home/"`.
- **hasSameBaseAs** indicates that two file names have the same base, e.g., if $X =$ `"somefile.txt"` and $Y =$ `"somefile.doc"` then X **hasSameBaseAs** Y (as well as Y **hasSameBaseAs** X) hold, with the common base being `"somefile"`.
- **hasSameExtensionAs** indicates that two file names have the same extension. For example, if X has the value `"somefile.txt"` and Y has the value `"someotherfile.txt"`, X **hasSameExtensionAs** Y (as well as Y **hasSameExtensionAs** X) holds, where the common extension is `"txt"`.

In the next section, we describe how such relations can be formulated and learnt.

Interpretation of Binary Relations. Unary relations were defined to apply to all occurrences of an event argument in any trace. However, this interpretation is unsuitable for binary relations, since a relation of the form X **equal** Y cannot hold with such an interpretation unless both X and Y have only one possible value in all traces. Hence we use an alternative interpretation that pairs each occurrence of X in a trace with a single occurrence of Y. A natural way to do this is to pair X and Y occurrences that are closest to each other.

Definition 1 (Lifting a relation R to a trace.) *Given a binary relation* **R** *on event arguments X and Y, we can lift* **R** *to a trace T, denoted X $\mathbf{R}_T$ Y, which holds iff, for each occurrence of X and its closest preceding occurrence of Y in T, X $\mathbf{R}$ Y holds. X $\mathbf{R}_{\mathcal{T}}$ Y holds iff X $\mathbf{R}_T$ Y holds $\forall T \in \mathcal{T}$.*

Consider a labeled trace T of the form

$$Y = 1, Z = 2, X = 1, Y = 2, X = 2.$$

We can say that X **equal**$_T$ Y, but not Y **equal**$_T$ X. Also, Y **equal**$_T$ Z, but not vice-versa. Now, consider another trace, and the **isWithinDir** relationship:

$Y =$`"/tmp"`, $X =$`"/tmp/f1"`, $X =$`"/f2"`,
$Y =$`"/var"`, $X =$`"/var/g1"`, $X =$`"/g2"`

It is clear that X **isWithinDir**$_T$ Y does not hold since the second occurrence of X does not satisfy **isWithinDir** when compared with the first Y. Note, however, that X **isWithinDir** Y holds if we ignore all but the first occurrence of X among a series of occurrences of X without

```
1. int main(int argc, char **argv) {
2.   source_dir = argv[1];  target_file = argv[2];
3.   target_fd = open(target_file, WR);
4.   push(source_dir); // uses a global stack
5.   while (dir_name = pop()) != NULL) {
6.       dir = opendir(dir_name);
7.       foreach (dir_entry ∈ dir) {
8.           if (isdirectory(dir_entry))
9.               push(dir_entry);
10.          else {
11.              source_fd = open(dir_entry, RD);
12.              read(source_fd, buf);
13.              write(target_fd, buf);
14.              close(source_fd);
15.          }
16.      }
17. }
18. close(target_fd);
19. exit(0);
20. }
```

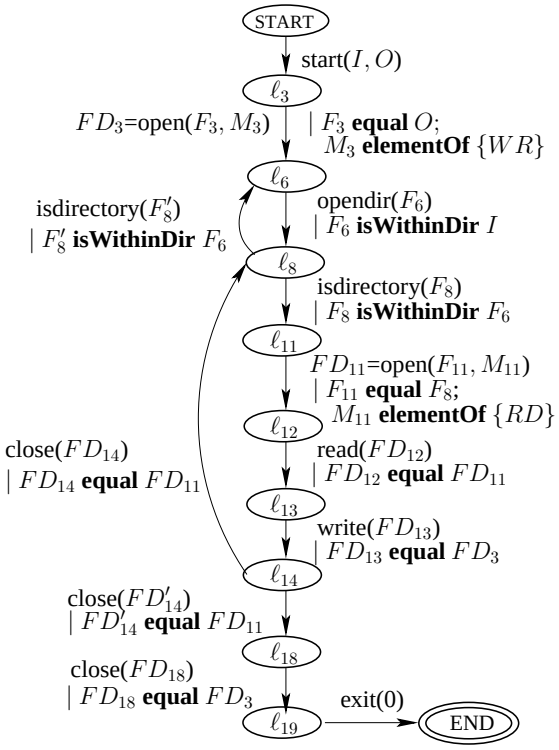

Figure 1. Example program *simpletar* (left) and its model with argument information (right).

an intervening Y. This motivates a different flavor of the relation $\mathbf{R}_T$ that we denote using $\mathbf{R}'_T$.

Definition 2 *X $\mathbf{R}'_T$ Y holds iff X $\mathbf{R}$ Y holds for each pair X, Y in T without an intervening X or Y.*

Finally, consider the trace $X = 1, Y = 0, X = 2, Y = 1, X = 3, Y = 2, ...$ Clearly, the value of Y equals the value of the last but one preceding X. To accommodate this, we define another variation $\mathbf{R}^n_T$:

Definition 3 *X $\mathbf{R}^n_T$ Y holds iff X $\mathbf{R}$ Y holds for each occurrence of X and its $n+1$th preceding occurrence of Y.*

Note that $\mathbf{R}_T$ is the same as $\mathbf{R}^0_T$. In the above example trace, Y **equal**^1_T X.

2.5. Motivating Example

We illustrate the concepts developed above with an example program that is a highly simplified version of the `tar` program. Figure 1 shows this *simpletar* program, which takes a command-line argument describing the source directory, and another command-line argument that specifies the name of the archive. It traverses the directory, which may contain subdirectories, and copies all the files into the archive. For simplicity, we ignore many aspects of archiving such as maintaining file boundaries, directory structures, and so on. In addition, we have abstracted away some details such as the use of `lstat` system call, and replaced them with more descriptive names such as `isdirectory`. In the example, all system calls are underlined.

Figure 1 also shows an abstract version of FSA model learnt for *simpletar*. States in the model are labeled with ℓ_n where n is the line number. Transitions are labeled with system call and argument information. Argument names are based on the nature of the argument and the transition they are associated with, e.g., a file descriptor argument to the `close` system call made on the transition from state ℓ_{14} to ℓ_8 is labeled as FD_{14}, while FD'_{14} refers to the same argument when the transition is from ℓ_{14} to ℓ_{18}. Relationships are shown as annotations on the transitions.

To illustrate how some of these relationships are learnt, we consider an execution trace generated when the program is run to archive the directory `/opt/proj` into a tarball `/tmp/proj.tar`. The operations of the trace are shown in the first column of Figure 2. The second column shows corresponding control-flow transitions learnt in the FSA. The third column shows values of system call arguments, and the fourth column shows some of the relationships learnt. In the generation of the trace, we have introduced a synthetic event `start` to capture command line arguments, and converted all the files names into absolute path names. Some of the relationships, such as those capturing absolute values of the file descriptors, are not shown in the above example. In addition, absolute values of various file arguments will be learnt from the above trace, but these are not shown

Operation Traces	Control-Flow Transition	Argument Values	Satisfied Data-Flow Property
Program started with arguments "/opt/proj","/tmp/proj.tar"		$\{I$ = "/opt/proj", O = "/tmp/proj.tar"$\}$	
ℓ_3 : open("/tmp/proj.tar", WR)=3	$start \rightarrow \ell_3$	$\{F_3$ = "/tmp/proj.tar", M_3 = WR, $FD3 = 3\}$	F_3 **equal** O, M_3 **elementOf** {WR}
ℓ_6 : opendir("/opt/proj")	$\ell_3 \rightarrow \ell_6$	$\{F_6$ = "/opt/proj"$\}$	F_6 **isWithinDir** I
ℓ_8 : isdirectory("/opt/proj/README")	$\ell_6 \rightarrow \ell_8$	$\{F_8$ = "/opt/proj/README"$\}$	F_8 **isWithinDir** F_6
ℓ_{11} : open("/opt/proj/README", RD)=4	$\ell_8 \rightarrow \ell_{11}$	$\{F_{11}$ = "/opt/proj/README", M_{11} = RD, $FD_{11} = 4\}$	F_{11} **equal** F_8 M_{11} **elementOf** {RD}
ℓ_{12} : read(4)	$\ell_{11} \rightarrow \ell_{12}$	$\{FD_{12} = 4\}$	FD_{12} **equal** FD_{11}
ℓ_{13} : write(3)	$\ell_{12} \rightarrow \ell_{13}$	$\{FD_{13} = 3\}$	FD_{13} **equal** FD_3
ℓ_{14} : close(4)	$\ell_{13} \rightarrow \ell_{14}$	$\{FD_{14} = 4\}$	FD_{14} **equal** FD_{11}
ℓ_8 : isdirectory("/opt/proj/src")	$\ell_{14} \rightarrow \ell_8$	$\{F'_8$ = "/opt/proj/src"$\}$	F'_8 **isWithinDir** F_6
ℓ_6 : opendir("/opt/proj/src")	$\ell_8 \rightarrow \ell_6$	$\{F_6$ = "/opt/proj/src"$\}$	F_6 **isWithinDir** I
ℓ_8 : isdirectory("/opt/proj/src/a.c")	$\ell_6 \rightarrow \ell_8$	$\{F_8$ = "/opt/proj/src/a.c"$\}$	F_8 **isWithinDir** F_6
ℓ_{11} : open("/opt/proj/src/a.c", RD)=4	$\ell_8 \rightarrow \ell_{11}$	$\{F_{11}$ = "/opt/proj/src/a.c", M_{11} = RD, $FD_{11} = 4\}$	F_{11} **equal** F_8 M_{11} **elementOf** {RD}
ℓ_{12} : read(4)	$\ell_{11} \rightarrow \ell_{12}$	$\{FD_{12} = 4\}$	FD_{12} **equal** FD_{11}
ℓ_{13} : write(3)	$\ell_{12} \rightarrow \ell_{13}$	$\{FD_{13} = 3\}$	FD_{13} **equal** FD_3
ℓ_{14} : close(4)	$\ell_{13} \rightarrow \ell_{14}$	$\{FD'_{14} = 4\}$	FD'_{14} **equal** FD_{11}
ℓ_{18} : close(3)	$\ell_{14} \rightarrow \ell_{18}$	$\{FD_{18} = 3\}$	FD_{18} **equal** FD_3
ℓ_{19} : exit(0)	$\ell_{18} \rightarrow \ell_{19}$		

Figure 2. A sample trace of the program in Figure 1 and the observed argument relationships.

here, since these values will vary when trained with many different traces, and will eventually be discarded.

From the above example, it is evident that binary relationships involving file names and file descriptors are very useful, as they allow us to track many interesting dataflow properties of the traces. It is interesting to note that some of these relationships arise due to the properties of the program environment, rather than the program itself. For instance, the relationship **isWithinDir** between F_8 and F_6 exists due to the fact that in the file system, the absolute path name of a file has the name of its parent directory as a prefix. A static analysis technique will have a hard time extracting such relationships, since it is obviously impossible to infer the semantics of the file system from this program.

3. Learning Argument Relationships

In this section, we describe our algorithms for learning the relationships described in the last section. These algorithms take labeled traces as inputs, and output the relationships that hold in all traces.

3.1. Learning Unary Relations

Learning unary relationships is straight-forward. With each event argument, the algorithm maintains a list of all the values encountered in all the traces. If the number of values exceeds a threshold, then the algorithm approximates the set. The kind of approximation that is appropriate for each event argument must be externally specified, through a configuration file. A few lines from this configuration file are shown in Figure 3. The first line specifies that for event arguments that represent file modes, up to 4 distinct values should be remembered. (Other parts of the configuration file specify which event arguments represent what type of data.) Beyond that, the values are to be approximated using a bit-or operation. The second line is applicable to file or directory arguments. A maximum of 10 distinct values are to be remembered, after which they are to be approximated using a common prefix operation. This line also states that for file names that match "/etc/*" or "/lib/*", the name should be remembered, regardless of the size of the set. The last line states that for file and socket descriptors, a maximum of 4 distinct values should be remembered, after which only the range of values is to be kept.

value kind=[MODE] **approx**=BIT_OR **max**=4;
value kind=[PATH] **approx**=PREFIX **max**=10
must=["/etc/*","/lib/*"];
value kind=[FD, SD] **approx**=RANGE **max**=4;

Figure 3. A sample configuration file.

Runtime and Storage Requirements. Unary relations can be learnt in $O(N)$ time, where N is the length (measured in terms of number of bytes) of the labeled trace, provided we restrict ourselves to simple approximation operations on strings such as the longest common prefix (LCP). Note that LCP can be computed quickly if the strings are represented using a trie [22]. Construction of tries takes time that is linear in the size of input strings. Computing the common prefix takes no longer than the length of the shortest string in the trie. Since at least one insertion will take place before a second invocation of the common prefix operation, it can be shown that the total time taken to maintain the LCP information is bounded by the total length of all strings in the input trace, which, in turn, is bounded by the trace length.

Storage requirements are dictated by (a) the total number of distinct argument names in the traces, and (b) the max-

```
1. procedure LearnRelations(EvArg X, Value V) {
2.     Y = ValTable.lookup(V, R);
3.     CurRels[R][X] = CurRels[R][X] ∩ Y;
4.     Y_n = Y ∩ NewArgs(X);
5.     CurRels[R][X] = CurRels[R][X] ∪ Y_n;
6.     ValTable.update(X, V);
7. }
```

Figure 4. Relationship learning algorithm

imum size of sets before an approximation is performed. Note that due to the way we generate labeled traces, (a) is bounded by the size of the control-flow model for N-gram, FSA and VtPath methods. The key point here is that even if the size of the traces is increased without a bound, the storage requirements are still going to be bounded.

3.2. Learning Binary Relations

Binary relations are learnt using the procedure LearnRelations shown in Figure 4. This procedure is invoked repeatedly for each event in the input trace, and for each argument to this event. LearnRelations is parameterized with respect to a relation $\mathbf{R}$, and is designed to learn $\mathbf{R}_T$. (Modifications to learn $\mathbf{R}'_T$ and $\mathbf{R}^k_T$ are discussed subsequently.) It takes two arguments: the current argument name X, and its current value V. It uses two global data structures: (a) ValTable, which is used to store the values of the most recent occurrences of all event arguments, and (b) CurRels, which is indexed by a relation $\mathbf{R}$ and an argument name X, and stores the set $\mathcal{Y}_{cur}$ of all arguments Y_c such that $X\ \mathbf{R}_{T_p}\ Y_c$ holds for the prefix T_p of current trace up to, but *not* including X. Both data structures are initialized to be empty at the beginning of the algorithm.

At line 2, ValTable is looked up to identify the set $\mathcal{Y}$ of all arguments Y such that $X\ \mathbf{R}\ Y$ holds for the most recent value of Y. In the next step, arguments Y_d in $\mathcal{Y}_{cur}$ that aren't in $\mathcal{Y}$ are deleted, as the latest occurrence of Y_d didn't possess the specified relationship with X.

Line 4 is designed to handle event arguments Y_n whose first appearance in the current trace occurred after the previous occurrence of X. We rely on a function NewArgs to identify such arguments. Note that for such Y_n, the condition characterizing $X\ \mathbf{R}_{T_p}\ Y_n$ holds vacuously. At the same time, the relationship has not actually been verified to hold even once, so Y_n does not appear in CurRels$[\mathbf{R}][X]$. Therefore, such event arguments are explicitly added to CurRels$[\mathbf{R}][X]$ at line 5. Finally, ValTable is updated at line 6 with the latest value of X.

A slight generalization of this algorithm is necessary when dealing with relations that take an additional constant parameter. For instance, consider the relation **isWithinDir** such that $X\ \mathbf{isWithinDir}\ Y$ iff $X = Ys$, i.e., X is obtained by adding a suffix s to Y. In this case, the suffix s may change ("shrink") as we examine more Y, X pairs. For example, with a trace $Y = abc, X = abcde$, s has the value de, but when an additional pair of X, Y values are added, as in the trace $Y = abc, X = abcde, Y = efg, X = efgd$, s becomes $d*$. To support such relationships, a refinement of **isWithinDir**, which modifies the parameter s, will be needed at step 3.

Learning $\mathbf{R}'_T$ requires a slight change to the algorithm. Specifically, at step 3, we do not delete some variables from CurRels$[\mathbf{R}][X]$ even when they aren't in $\mathcal{Y}$. This exception is made for those variables Y_e that haven't appeared on the trace since the previous occurrence of X. This can be easily checked by associating timestamps for arguments, and checking that the last timestamp of Y_e is less than the timestamp of the previous occurrence of X. Learning $\mathbf{R}^k_T$ requires another kind of change: rather than deleting old values of an event argument X each time a new value is seen, we retain the k most recent values of X.

Runtime and Storage Requirements. Below, we discuss the runtime and storage requirements for learning a single relation. The LearnRelations algorithm is invoked $O(M)$ times, where M is the total number of events in the trace. (This assumes that each event has $O(1)$ arguments.) If a hash table representation is used, and $\mathbf{R}$ is the equality relation on integral types, then line 2 of the algorithm can be completed in $|\mathcal{Y}|$ time. If $\mathbf{R}$ is an operation such as a **isWithinDir**, **contains**, or **hasSameDir**, then this step can be completed in time proportional to $|\mathcal{Y}| + length(V)$ by maintaining ValTable as a trie. Adding this over the M iterations of LearnRelations, we arrive at

$$\sum_{i=1}^{M} |\mathcal{Y}| + \sum_{i=1}^{M} length(X_i)$$

where X_i denotes the ith argument in the trace. This expression can be simplified to $M * |\mathcal{Y}| + N$. Noting that $|\mathcal{Y}|$ is bounded by the number distinct event arguments, which in turn is bounded by the size S of the FSA, we get a bound of $O(SM + N)$ worst-case runtime for the approach, when FSA method is used.

In the previous paragraph, we computed the runtime for executing line 2 of LearnRelations. It can be easily seen that the complexity of line 3 and 4 are bounded by the size of $\mathcal{Y}$. Also, since the size of $\mathcal{Y}_n$ is smaller than that of $\mathcal{Y}$, the runtime of line 5 is also bound by $|\mathcal{Y}|$. Finally, the update of the value table takes time bounded by $length(V)$ for integer and string relations. Thus, the runtime of all other steps is bounded by the runtime of line 2, and hence the overall complexity of $O(SM + N)$.

The above argument is based on the worst-case size of $\mathcal{Y}$. By retracing the arguments, it is easy to see that we can replace S by the average size A of $\mathcal{Y}$, and the complexity argument would still hold. In our experiments, we have ob-

served that A is relatively small (less than 10), while S is much larger – of the order of hundreds. Moreover, N was about a hundred times larger than M. As a result, the factor N dominates in practice over SM. This leads to a practically efficient algorithm that takes time linear in the size of the input trace file length.

It is easy to see that the storage requirements for learning binary relations is dependent on the number of distinct event arguments. A worst-case storage complexity that is quadratic in S can be established, but in practice, we find that the storage requirements are more or less linear in S.

Dealing With Multiple Traces. The obvious approach for handling multiple traces is to process each trace using LearnRelations, one after the other. The global data structures need to be appropriately reinitialized between any two traces. In particular, CurRels table is not reinitialized between traces. ValTable is cleared at the end of each trace. Finally, a slight modification is necessary to the definition of NewArgs(X). It will include only those event arguments Y such that no occurrence of Y has preceded an occurrence of X in the current trace up to X, or in any of the previously processed traces.

Specifying Relations of Interest. Our algorithm is designed to support common binary relations such as **equal**, and file name/path related relations such as **isWithinDir**, **contains**, **hasSameDirAs**, **hasSameBaseAs**, and **hasSameExtensionAs**. We limit relationship learning within arguments that represent the same kinds of objects, e.g., a relationship is learnt among file descriptors, but none is learnt between a file descriptor and a userid. The relations of interest are specified using a configuration file, a section of which is shown below:

relation	equal	**kind**=[FD, UID, PATH]
relation	isWithinDir	**kind**=[PATH]
relation	hasSameDirAs	**kind**=[PATH]

The first line specifies that for arguments representing file descriptors, userids and file names, equality relationship should be learnt. The second and third lines specify that for file name arguments, **isWithinDir** and **hasSameDirAs** relations are of interest.

Observe that our model learning algorithm discounts the possibility of coincidental relationships, in the sense that any relationship that *appears* to hold is assumed to be a real, meaningful relationship. This assumption works well if the probability of accidental relationships is rather small. However, for file descriptors, which typically range over a small number of distinct values, this assumption does not hold. As a result, meaningless relations are often learnt, such as the relationship between a file descriptor argument of `read` and an `open` of an entirely different file that was opened and closed long ago. To address this problem, we specify that when a `close(fd)` occurs, no more relationships involving that `fd` should be learnt. This is specified in the configuration file using a "terminate" flag with the `close` operation.

Another related problem involving file descriptors is that a number of "useless" relationships are learnt. Suppose that a program opens a file at location ℓ_0 and then performs read operations on this file from locations ℓ_1,..,ℓ_n. Let X_0, X_1,..,X_n be the corresponding argument names. Our algorithm will learn that X_n has equality relationships with X_0 through X_{n-1}, that X_{n-1} has equality relationships with X_0 through X_{n-2}, and so on. From a security perspective, it is clear that a relationship between X_i and X_0 is useful, for $0 < i \leq n$, since these relationships associate a `read` operation with the name of the file being opened. However, it does not seem to be useful to learn a relationship between X_i and X_j where $i, j \neq 0$. Such useless relationships can be specified in the configuration file, and the learner avoids learning them. Our current implementation considers a relationship between two file descriptor arguments X and Y to be useful only when Y corresponds to the return value of an open operation.

4. Implementation

We have implemented our approach on RedHat Linux 7.3. The implementation consists of an online and an offline component. The online component is a *tracer*, which uses `ptrace` mechanism to trap each system call made by a monitored process and logs the following information: (a) the program location from where the system call was invoked, (b) values of significant arguments, and (c) the return code for a system call. The tracer incorporates some knowledge of the argument data that is useful, e.g., filenames are logged, but not the buffer arguments of system calls such as `read`. The tracer performs some normalization of system call argument values, e.g., converting filenames into a canonical form obtained by resolving symbolic links and occurrences of "." and "..". For some system calls, additional information is logged, e.g., inode information for `open` and `stat` calls, and IP address and port information for sockets.

The tracer records program parameters such as open file descriptors, command-line arguments and environment variables. These parameters are provided as arguments to "synthetic events" generated by the tracer. For open file descriptors, a synthetic `open` event is introduced that records the name of the file (or network endpoint) involved. For environment variables, the synthetic event name is derived from the name of the environment variable. For command-line arguments, the name is derived from the position of the argument, i.e., the 5th command-line argument is provided as an argument to an event named `arg5`. A more meaningful event name that is based on the function of an argument (rather than its position) can be generated us-

Reference	Program	Attack description	Detected?
S. Chen et al. [5]	WU-FTPD	format string attack overwrites userid data	Yes (B)
S. Chen et al. [5]	Netkit Telnetd	heap overfbw to corrupt name of execve'd program	Yes (B, U)
S. Chen et al. [5]	GHTTPD	stack overfbw to overwrite fi lename data	Yes (U)
CVE-2000-0915	Fingerd	read arbitrary fi le by symlinking `.plan` to it	Yes (B)
CVE-2002-0435	GNU rm	race condition	Yes (B)
H. Chen et al. [4]	Synthetic	causing fi le open to return stderr descriptor	Yes (U)

Figure 5. Attacks used in effectiveness evaluation. In the detection column, "B" and "U" respectively indicate that the attack was detected as a violation of a binary relationship or a unary relationship.

ing an application-specific plugin to the tracer that maps command-line arguments into appropriately named event sequence. Such application-specific plugins can also be used to obtain other parameters to a program, e.g., values specified in a configuration file.

The offline component consists of a log file parser, which reconstructs the system call events and feeds them into the learning module. The operation of the learning module is controlled using a configuration file as described earlier. The implementation of this module follows the description in the last section. Our experiments make use of only the $\mathbf{R}_T$ relation, but not the other variants $\mathbf{R}'_T$ and $\mathbf{R}^k_T$.

5. Evaluation

In this section, we first study the effectiveness of our approach in attack detection (Section 5.1), followed by an analysis of false alarm rates (Section 5.2). We then study the precision of models using the branching-factor metric proposed in [31]. Finally, performance overheads for intrusion detection are discussed in Section 5.4.

5.1. Detection of Attacks

Note that in principle, a range of attacks are detectable by our approach. One way to establish its effectiveness is to select some of the most visible exploits from US-CERT or CVE, and demonstrate their detection. However, this would not be very helpful because we would primarily be testing with very easy-to-detect attacks such as code-injection attacks that alter control flows in obvious ways. Indeed, many of those attacks can be detected by control-flow models. However, the real problem is that an attacker can easily adapt his attack to evade detection by these techniques. The main advantage of our approach is that due to the improved precision offered by it, it can block such stealthy attacks designed to evade existing IDS. To establish this, we:

- *Demonstrate detection of a collection of stealthy attacks.* (See Section 5.1.1.) We chose the set of attacks shown in Figure 5. Some of these attacks were designed to evade techniques focused on control-flow hijack attacks. Another category of attacks we studied was that of race conditions and symbolic link attacks, a category that has attracted techniques that were specifically designed for it. Previous program behavior based anomaly detection techniques don't detect them, since in their view, the behavior of the program hasn't changed at all.
- *Formally establish, using automated verification techniques, that certain attacks would never succeed.* We can show that if a program exhibits the behavior specified by its model, then it would provide some safety guarantees, *regardless of any attack mounted against it.* This is a unique feature of our technique and is outlined in Section 5.1.2.

5.1.1 Detection of Stealthy Attacks

For each of the attacks shown in Figure 5, we obtained the corresponding exploit from the Internet or developed it ourself. We ran the exploits and verified that the attacks were successful. Then we trained the programs involved with benign data. We then reran the attack, and verified that each attack caused an anomaly. We investigated the anomaly to ensure that the anomaly was a direct consequence of the attack, and not an artifact that resulted from poor training data, or nonessential aspects of the attack.

Figure 5 shows whether attacks were detected as a violation of a binary relation (B) or a unary relation (U). In half the cases, the essential feature of an attack was the violation of a binary relationship. This means that previous approaches, which did not rely on binary relations, won't be able to detect these attacks. Of the remaining three attacks, the last attack in the table requires control-flow context to be combined with argument value, and hence we believe that it won't be detected by previous techniques, since they did not leverage control-flow information. The remaining two attacks could be detected by some previous approaches for argument learning. Below, we describe each of the attacks in more detail.

WU-FTPD: Corruption of user identity data [5]. This attack exploits wu-ftpd format string vulnerability [3]. It involves the following code in `getdatasock()` function:

```
L1:  seteuid(0);
     setsockopt(...);
     ...
L2:  seteuid(pw->pw_uid);
```

In the above code, `setsockopt()` operation requires root privilege. For this reason, the privilege is temporarily escalated using `seteuid(0)`, and then dropped afterwords using `seteuid(pw->pw_uid)`. The attack exploits the format-string vulnerability to change `pw->pw_uid` to 0. Therefore, the root privilege is maintained even after second `seteuid()` call, which allows the remote attacker to upload and download arbitrary files as a root user.

In our experiments, we did not use the actual format-string attack, but simulated it by instrumenting the code to change value of `pw->pw_uid` under attack scenario. Our implementation detected this attack because it learns an equality relation between the argument of `seteuid` at line L2 and another setuid call appearing in the function `pass()`, which is invoked when the user first logs in. This equality relation is violated in an attack. It is important to detect this attack as a violation of this relationship: an alternative, such as raising an alarm when the absolute value of `seteuid` argument at `L2` is zero, would raise a false alarm when `root` uses this server.

Netkit Telnetd: Corruption of filename to be executed [5]. At the beginning of each client connection, the telnet daemon authenticates its user with an external program. The name of this program is stored in a variable `loginprg`. In this attack, a heap overflow vulnerability is used to overwrite this variable with the value `/bin/sh`, so that a subsequent authentication attempt by a user will result in a root shell. We simulated this attack in the similar manner as the previous attack.

With typical configurations of `telnetd`, `loginprg` always has the value `/bin/login`. In this configuration, it is easy to detect the attack as a violation of the value normally observed as the argument of `execve`. It is more interesting to note that our models are successfully able to handle atypical configurations as well, where `telnetd` may be invoked with different command line parameters specifying different authentication programs. In this case, recall that the tracer introduces a synthetic event to record the command line argument. The model captures the relationship between this argument and the argument of `execve`. In this manner, a model that is produced in an environment using a login program `x` can be deployed in another environment with a login program named `y`, and still be able to detect this attack.

GHTTPD: Directory traversal by corrupting filename [5]. A stack overflow in GHTTPD web server can be used to evade path name checks, and execute an arbitrary program [5]. Attack occurs in the following code fragment in `serverconnection` function:

```
     if (strstr(ptr, "/.."))
         return ... //reject request
     Log(...);
L1:  if (strstr(ptr, "cgi-bin")) execve(ptr, ...)
```

Variable `ptr` is a pointer to a text string of the URL requested by a remote client. The function `serverconnection()` checks the absence of `"/.."`, and the presence of `"cgi-bin"` in the URL before the CGI request is handled. The purpose of these checks is to ensure that only programs in the CGI-BIN directory are executed by the server. The function `Log()` has a buffer overflow vulnerability which is exploited to change `ptr` to point to a string `/cgi-bin/../../../../bin/sh` (details can be found in [5]). The subsequent check `strstr(ptr, "/cgi-bin")` is successful and spawns a shell. We used an actual buffer overflow to produce this attack.

Our system learns that the common prefix of all files executed at `L1` is the CGI-BIN directory, i.e., F **isWithinDir** CGI-BIN directory, where F is the file name argument of `execve`. Since this condition is violated by the above attack, our approach was able to detect it.

Fingerd symlink vulnerability. Some programs assume that file names given to them are regular names and do not contain symbolic links. Attacks can be crafted by violating this assumption. We describe an example of symlink vulnerability in old versions of BSD `fingerd` [8]. This server uses a local `finger` client program to serve remote requests. The server runs with root privileges, and executes the client without dropping these privileges. This allows the following attack: a user can create a symbolic link called `.plan` in his home directory that points to a file readable only to root (e.g., the shadow password file). Now, by running a finger on himself from a remote site, he can see the contents of this file.

The vulnerability arises in the following code snippet in `show_text()` function, which verifies the presence of a file to be shown, but does not check if it is a symbolic link.

```
     if (lstat(tbuf, &sbuf1))  return 0;
L1:  fd = open(tbuf, O_RDONLY); ...
     fp = fdopen(fd, "r");  ...
```

As essential aspect of this attack is that the file name that is actually read isn't within the directory of the user. This is detected in our approach as a violation of the relationship between the command-line argument, which specifies the name of the user to be fingered, and the directory of the filename opened at `L1`. (Recall that we resolve symbolic links in filenames before using them for learning or detection.) The attack could potentially be detected by observing that the resolved filename is something other than `.plan`, but this would raise a false alarm if the user were to use the symbolic link in a benign way, say, by linking `.plan` to another file named `schedule`.

Race condition attacks. Race conditions in file access occur when applications incorrectly assume that a sequence of operations on files is atomic. The prototypical example is that of a setuid-to-root program using `access` system call

to check if its real user $ruid$ has access to a file `f`, and then using `open(f)` to open it. In between the two calls, an attacker (who is typically the real user) can change `f`, so that the `access` call will succeed, but by the time `open` is executed, `f` points to a file that isn't accessible to $ruid$. Race condition attacks are among the hardest attacks to detect, and this has led to the development of detection techniques specifically targeting them [29, 16, 20, 30]. It is interesting to note that without any specialized effort, our approach can detect them.

To demonstrate the ability of our approach to detect real-world race attacks, we selected race condition in `rm` [25] from GNU file utilities package. The attack exploits the fact that `rm` descends into a subdirectory using `chdir`, and then ascends out of this subdirectory using `chdir("..")`. In the window of time between the descend and ascend operations, an attacker can move the subdirectory higher up. This will result in the second `chdir` operation going out of the directory on which it was invoked. For instance, consider an operation `rm -r /tmp/a/`, where `a` contains a subdirectory `b`. When `rm` descends into `/tmp/a/b`, the attacker can rename `/tmp/a/b` to `/tmp/b`. Now, when `rm` executes a `chdir("..")`, it will go into `/tmp`, and will start deleting all files in `/tmp`, which is different from the original intent of removing the subdirectory `/tmp/a`. In this attack, typically the `rm` will be invoked by root to clean up some directories of `/tmp`, while the attacker has write permission on the subdirectory `/tmp/a`.

In our experiments, we inserted sleep command in the `rm` program to obtain a sufficient time window to launch the actual attack. For the `rm` program, our implementation learnt a relationship between its command-line argument and all of the arguments to `unlink` and `rmdir` system calls made by it: namely, that the arguments to these system calls should be within the directory name given by the command-line argument. This relationship was violated during the attack, and hence it was detected. Other types of race conditions can also be detected as violations to path relationships. For a more robust detection technique, one can rely on inode numbers instead of filenames obtained using *realpath*.

Attacks on file descriptors. Programs may make assumptions about the meanings of file descriptors, e.g., that descriptor 2 corresponds to `stderr`. An example setuid program with such a vulnerability is described in [4]:

```
     fd = open("/etc/passwd");
     str = read_from_user();
L1:  fprintf(stderr,
        "The user entered:\n%s\n", str);
```

If the attacker `execve`'d this program after closing `stderr`, then a open of `"/etc/passwd"` will return file descriptor 2, and subsequently, the `fprintf` will have the effect of writing user provided data into the password file. This attack is detected as a violation of unary relationships learnt on file descriptors.

5.1.2 Verifying Security Properties Using Models

Note that if a security policy P can be statically verified with respect to a model M learnt by our technique, then one can be assured that an intrusion detection system based on M will detect any attack that violates P. Clearly, it would be beneficial if one can make such deterministic assertions about an anomaly detection system.

For verification, security policies are expressed as an extended finite-state automaton, i.e., a finite-state automaton that can remember a finite number of values such as file names. Technically, these automata capture negations of safety properties, so they accept traces that violate the desired security property. The models are also extended finite-state machines accepting normal execution traces. The verification then amounts to taking the intersection of the property and model automata, and checking if the language accepted by this automata is nonempty. If so, then the property is violated. The verifier is written in XSB Prolog [35], a system well-suited to writing verification tools. The focus of this section is on the results of verification rather than the verification process, so we omit the technical details of this process.

Following are three of the properties that we actually verified for `tar`, `gzip` and `find`:

- `find` *executes only those programs that are specified using a "*`-exec`*" command-line option.* To verify this property, we need an application-specific command-line parser to recognize the parameter following "`-exec`" switch and generate a corresponding synthetic event. The property itself states that the first argument to any `execve` made by `find` is equal to the "`-exec`" argument. Since an equality relationship is learnt in the model involving the "`-exec`" parameter and the argument to `execve`, this property is easily verified.
- *All files read by* `tar` *would reside within the directory specified on the command-line.* Again, we need an application-specific command-line parser to generate a synthetic event that captures the value of this directory argument. Once this is done, our model learns that files read by `tar` are within this directory, or are configuration files and shared libraries that get loaded during process start-up. A policy that allows reading of configuration files, libraries, and the files below the specified directory is verified against this model.
- *The only file written by* `gzip` *is obtained by adding a* `".gz"` *suffix to its argument.* Similar to the previous two examples, this property is verified without any problem[1]

[1]Note that this property holds only if gzip is used with typical command-line options. Otherwise, one would need a more complex policy that correlates command-line parameters to the files accessed by it.

Program	Training Trace length (# Syscalls) ($\times 10^6$)	Detection			
		Trace length (# Syscalls) ($\times 10^6$)	False alarm rates		
			Base ($\times 10^{-5}$)	Unary ($\times 10^{-5}$)	Binary ($\times 10^{-5}$)
httpd	1.75	3.10	16.6	0.97	64.12
sshd	4.15	14.74	0.35	0.79	0.02

Figure 6. False alarm rates.

In some instances, we could not verify properties in the manner we expected. For instance, in the case of `httpd`, we tried verifying the property that the only files executed by it were within a cgi-bin directory. The verification succeeded, but we subsequently realized that this was because the number of distinct executables seen during the training trace was small enough that no approximation had taken place. If more scripts had been executed, then an approximation using common prefix (unary **isWithinDir**) relation would have been applied. However, since (a) there were multiple cgi-bin directories, (b) our learning algorithm currently learns only a single common prefix, and (c) the common prefix for these two directories is just `/`, the model would only capture that all executed files are within `/`. As a result, verification would not succeed in this case. To handle this problem, our learning algorithm needs to be extended to handle disjunctions: that certain variables satisfy one of many binary relations, and/or one of many unary relations. This is a topic of our continuing research.

5.2. False Alarm Analysis

To determine false positive rates, we trained the system with system call traces of different lengths. After training, the system was run in detection mode against a different system call trace. To be useful, false alarm analysis should be performed with live traffic, rather than being based on training scripts. This limited our choice of applications. In our laboratory, the two main servers that are well-exercised are `httpd` and `sshd`, so we limited our false alarm analysis to these two programs.

The results tabulated in Figure 6 show that the false alarm rates are of the order of 10^{-4}. Note that this corresponds to "raw" false alarm rates, i.e., the fraction of system calls that caused violations, without any regard to the nature of violations. In a practical system, these raw alarms will be further evaluated, based on the nature of violation. Moreover, series of alarms would be aggregated into one. These factors typically result in a further significant reduction in false alarm rates. For this reason, it is hard to evaluate the false alarm rates directly.What we can do is to compare them with those reported by previous techniques such as the FSA method, which is known to produce a modest false alarm rate.

The addition of unary relations increases the false alarm rate modestly. Note that binary relations add a very low false positive rate for `sshd`, but a much higher rate is observed for `httpd`. We investigated the reason for this, and found that this is due to the fact that in the training trace, a single system call was very rarely executed. Moreover, for the few values of the parameters to these calls, it turned out that they bore strong relations with arguments of many subsequent system calls. However, during detection, this same system call was executed several more times, and this broke the relations involving this argument value. In fact, we found that 95% of the false positives were due to this. To address this problem, one could add a notion of *confidence level* with each relation, which can be based on the number of times it has been verified during training. We are currently investigating such an approach to further reduce false alarms. Such measures may reduce the level of binary relation false positives to a fraction of the false alarm rate of the base method.

5.3. Model Precision

Average branching factor metric, originally developed by Wagner and Dean [31], has been used in the context of intrusion detection [14, 13] to measure precision of models. Basically, the idea is to determine the degree of freedom that an attacker has at each state of the model. This is roughly measured by the average number of branches that can be taken by the program at each state of the automaton during the program execution. A lower branching factor translates to improved model precision.

According to the definition in [31], system calls are partitioned into two sets, dangerous and harmless, and the average branching factor is defined in terms of branches that correspond to dangerous system calls. However, dangerous system calls are considered harmless if their arguments are known in advance. For example, `execv("/bin/ls")` is considered harmless, but if its argument were not known, attacker can potentially substitute the argument with `"/bin/sh"` to obtain a shell. We applied a similar metric to compute average branching factor in presence of argument information. For each dangerous system call, if argument values are learnt without approximations, then the system call is considered harmless. If approximations have been made while learning values, we further check if there are binary relations present for the corresponding argument or

Program	Program size(KB)	# States/ #Transitions	# Binary Relations
sshd	260	228/633	2309
wu-ftpd	435	207/492	2281
httpd	292	281/755	3638
find	68	29/71	107
tar	156	55/181	647
gv	292	195/729	3637
gzip	68	29/59	151

Figure 7. Sample model sizes of the test programs used in the experiments.

Program	Without argument learning	With argument learning	
		Unary only	Unary & Binary Rels
sshd	5.0615	0.0127	0.0004
wu-ftpd	2.1211	0.0352	0.0064
httpd	0.0711	0.0003	0.0002
find	1.1728	0.1615	0.0807
tar	4.7779	0.8709	0.2032

Figure 8. Average branching factor

Program	Workload	% Enforcement overhead	
		`realpath()` overhead	Detection overhead
gzip	Compress a 12MB file.	0	2
gv	Open and browse through a 500KB post script file.	0	5
tar	Archive 600 files into a 6MB tar file.	2	3
find	Search C header files in a directory tree of 12000 files.	41	11

Figure 9. Overhead for intrusion detection

not. If there exists a lossless binary relation, the call is considered to be harmless. Relation **equal** is considered to be lossless. Relations **isWithinDir** and **contains** are considered to be lossless if their constant parameters do not contain wild-card patterns such as "?" and "*". For instance, X **isWithinDir** Y is lossless iff $X = Ys$, where the constant parameter s does not contain any wild-card pattern. On the other hand, relation **hasSameDirAs** is lossy because only the common directory name is retained, but the rest of argument information is lost.

Figure 8 shows the average branching factor for our models without argument learning, learning unary relations alone, and learning unary and binary relations that do not lose significant information as mentioned above. The table shows that the use of binary relations leads to major improvements in branching factor.

We point out that the average branching factor of our technique cannot be directly compared to that of other techniques unless they too use an FSA-based control model. For this reason, we do not directly compare our results with those reported by [14].

5.4. Performance Overheads

Model Sizes. Figure 7 shows the programs used in our experiments, along with their model sizes in terms of number of states/transitions and relations in the models. The models are relatively small as compared to the size of programs involved.

Time for Learning Models. We studied the performance of the learning algorithm. Three programs were considered, `httpd`, `ftpd` and `sshd`. The training traces were between 100MB and 300MB, consisting of 1.5 to 4 million system calls. The learning algorithm took between 5 to 25 minutes to process these traces.

Overhead for Intrusion Detection. Our current implementation uses `ptrace` for system call interception, which itself introduces high runtime overheads for interception that can exceed 100% for some programs. To obtain better performance, an in-kernel interceptor can be used. Overheads for system call interception, including the costs of retrieving the PC, have been reported to be around 6% for a kernel implementation [9]. In addition to this, intrusion detection requires verification of binary relations, and the cost of making `realpath` calls for file names. We have measured these overheads individually, using our user level implementation, and shown it in Figure 9. We remark that `find` represents perhaps the worst-case scenario in terms of overheads because it performs a very large number of system calls involving file names, each of which incurs the overhead of a `realpath` call, and the overhead of verifying relationships on file names. An in-kernel implementation of `realpath` would likely have lower overheads, but the overhead numbers for relationship verification are unlikely to change.

6. Related Work

6.1. Static Analysis Techniques

A number of static analysis techniques have been developed for building intrusion detection models. A source-code analysis is used in [31, 19], while binary analysis is used in [14]. [14, 19] can also extract system call arguments that appear as immediate constants in the program. Binary analysis based approach of [12] additionally asso-

ciates calling context of the extracted static data using data flow analysis. They also incorporate environment dependency in the program models. However, this dependency needs to be specified manually in their approach. In contrast, our approach learns these dependencies automatically.

The primary benefit of static analysis based techniques is that they eliminate false alarms. This is because these models are *conservative*, capturing a superset of all possible behaviors that can be exhibited by a program. But the conservative nature also limits attack detection ability: only attacks that cause a program's runtime behavior to deviate from its code are detected. This means that a variety of attacks, such as input validation errors, race conditions, and so on, cannot be detected, as the erroneous behaviors do represent possible behaviors of the victim program. As a result, most attacks discussed in this paper can't be detected by these methods. Moreover, capturing accurate information about data values is quite challenging, given the complexities of a language such as C, which allows arbitrary type casts and pointer arithmetic.

6.2. Learning-Based Approaches

Intrusion Detection. A number of techniques to learn control-flow behaviors for intrusion detection have already been discussed before, so we focus our attention to techniques for learning argument information. In this regard, [18] describes techniques for learning statistical information about system call arguments for anomaly detection. The statistical information includes properties of string arguments such as its length and distribution of its characters. Furthermore, structural inferences are made over string arguments to learn a regular grammar that describes all of its normal values.

[28] proposes another host-based anomaly detection system that uses a rule-learning algorithm to model system call behaviors incorporating argument information. It uses a rule-based learning algorithm that captures a fixed number of distinct values of frequently occurring arguments.

In our terminology, both of the above two approaches are focused on unary relations, whereas the primary strength of our approach is that of learning the more complex binary relations. Moreover, our approach is able to utilize control-flow context to improve the precision of dataflow relationships, whereas the above approaches don't do that.

Hypothesizing Program Properties. [1] describes a technique for automatically extracting likely program properties from execution traces. However, it relies on humans to specify regions within a trace where such property extraction will be attempted. This contrasts with our technique, which is fully automated. [23] has similar goals as [1], but is fully automated. The primary difference with our technique is that they focus on invariant properties, whereas our algorithm is focused on temporal properties on traces. Technically, the two problems are quite different, requiring different techniques to be employed. For instance, algorithms for learning invariants can be speeded up by exploiting transitivity, i.e., if p holds and $p \rightarrow q$, then we need not explicitly verify q. Unfortunately, this is not true for trace properties.

Mobile Code Security. In [27], we described models similar to those of this paper. The goal of [27] was to provide a overall view of the model-carrying code approach for mobile code security, and hence that paper provides only a superficial treatment of models. The results presented in this paper improves over [27] in many important ways. First, we develop a formal treatment of dataflow properties in this paper. Second, we show how control flow contexts from different control-flow models can be utilized to improve precision of dataflow relationships, whereas [27] is limited to FSA method. Third, we develop an efficient algorithm for learning relationships and analyzing its complexity. Fourth, we show how to parameterize the model by incorporating dependence on program's environment, including command-line argument, environment variables, open file descriptors, and so on. Finally, and most importantly, we provide a detailed evaluation of our technique for intrusion detection in this paper, whereas [27] was concerned with mobile code security.

7. Conclusion

In this paper, we presented an approach for enhancing the accuracy of host-based intrusion detection models by capturing dataflow information. This approach can be layered over existing techniques for learning control-flows. We presented a formal treatment of data flow properties of traces, and presented an efficient learning algorithm that is parameterized with respect to relations of interest. Through experimental evaluation, we showed that the approach was effective in detecting sophisticated attacks on which most previous techniques fail. We also established that the models are compact and produce low false alarm rates. An important benefit of our approach is that it enables formal reasoning about the security guarantees that can be provided when these models are used for intrusion detection.

Acknowledgments

We thank Diptikalyan Saha for his invaluable help with verification, and Zhenkai Liang for his support with system call interposition environment for model extraction. We would also like to thank Shabbir Dahodwala, Daniel C. DuVarney, Yow-Jian Lin and C.R. Ramakrishnan for several discussions on model extraction and verification. Finally, we thank the anonymous reviewers for their insightful comments and suggestions.

References

[1] G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2002.

[2] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, 1996.

[3] CERT CC. CERT Advisory CA-2001-33 Multiple Vulnerabilities in WU-FTPD, 2001.

[4] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Network and Distributed System Security Symposium*, San Diego, CA, February 2004.

[5] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, Baltimore, MD, August 2005.

[6] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, 2004.

[7] H. Feng, O. Kolesnikov, P. Folga, W. Lee, and W. Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*, May 2003.

[8] Common security exploit and vulnerability matrix v2.0. Published on World-Wide Web at URL http://www.tripwire.com/files/literature/poster /Tripwire_exploit_poster.pdf, 2002.

[9] D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *ACM conference on Computer and Communications Security (CCS)*, pages 318–329, Washington, DC, October 2004.

[10] D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *USENIX Security Symposium*, pages 103–118, San Diego, CA, USA, August 2004.

[11] A. Ghosh and A. Schwartzbard. A study in using neural networks for anomaly and misuse detection. In *USENIX Security Symposium*, Washington, DC, August 1999.

[12] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller. Environment-sensitive intrusion detection. In *Recent Advances in Intrusion Detection (RAID)*, September 2005.

[13] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *USENIX Security Symposium*, San Francisco, CA, August 2002.

[14] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Network and Distributed System Security Symposium*, San Diego, CA, February 2004.

[15] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security (JCS)*, 6(3):151–180, 1998.

[16] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Annual Computer Security Applications Conference (ACSAC)*, December 1994.

[17] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *USENIX Security Symposium*, Baltimore, MD, August 2005.

[18] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *European Symposium on Research in Computer Security*, Gjøvik, Norway, October 2003.

[19] L. C. Lam and T. Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *Recent Advances in Intrusion Detection (RAID)*, Sophia Antipolis, French Riviera, France, September 2004.

[20] K. Lhee and S. J. Chapin. Detection of file-based race conditions. *International Journal of Information Security (IJIS)*, 4(1-2):105–119, 2005.

[21] C. C. Michael and A. Ghosh. Simple, state based approaches to program-based anomaly detection. In *ACM Transactions on Information and System Security (TISSEC)*, 2002.

[22] NIST, 2001. http://www.nist.gov/dads/HTML/trie.html.

[23] J. Perkins and M. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *ACM International Symposium on Foundations of Software Engineering (FSE)*, Newport Beach, CA, USA, November 2004.

[24] N. Provos. Improving host security with system call policies. In *USENIX Security Symposium*, Washington, DC, USA, August 2003.

[25] W. Purczynski. GNU fileutils - recursive directory removal race condition, March 2002. Bugtraq-fileutils mailing list.

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

[27] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, New York, October 2003.

[28] G. Tandon and P. Chan. Learning rules from system call arguments and sequences for anomaly detection. In *ICDM Workshop on Data Mining for Computer Security (DMSEC)*, pages 20–29, 2003.

[29] E. Tsyrklevich and B. Yee. Dynamic detection and prevention of race conditions in file accesses. In *USENIX Security Symposium*, Washington, DC, USA, August 2003.

[30] P. Uppuluri, A. Ray, and U. Joshi. Preventing race condition attacks on file systems. In *(ACM) Symposium on Applied Computing (SAC)*, 2005.

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

[32] D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems. In *ACM conference on Computer and Communications Security (CCS)*, 2002.

[33] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.

[34] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable-length audit trail patterns. In *Recent Advances in Intrusion Detection (RAID)*, Toulouse, France, October 2000.

[35] XSB. The XSB logic programming system v2.3, 2001. Available from http://www.cs.sunysb.edu/~sbprolog.