Forensic Analysis of File System Intrusions using Improved Backtracking

Sriranjani Sitaraman and S. Venkatesan
Department of Computer Science
The University of Texas at Dallas
Richardson, Texas 75083-0688.
ginss@student.utdallas.edu, venky@utdallas.edu

Abstract

Intrusion detection systems alert the system administrators of intrusions but, in most cases, do not provide details about which system events are relevant to the intrusion and how the system events are related. We consider intrusions of file systems. Existing tools, like BackTracker, help the system administrator backtrack from the detection point, which is a file with suspicious contents, to possible entry points of the intrusion by providing a graph containing dependency information between the various files and processes that could be related to the detection point. We improve such backtracking techniques by logging certain additional parameters of the file system during normal operations (real-time) and examining the logged information during the analysis phase. In addition, we use data flow analysis within the processes related to the intrusion to prune unwanted paths from the dependency graph. This results in significant reduction in search space, search time, and false positives. We also analyze the effort required in terms of storage space and search time.

Keywords: Intrusion Detection, Data Flow Analysis, Dynamic Slicing, Backtracking, File System

1. Introduction

Computer intrusions and attacks are increasing every year [3] and many of these are done with automated and sophisticated attack techniques [2]. Consequently, discovery or detection of intrusion is becoming more difficult. To help in determining how an attack happened and who is behind the attack, many events are logged continuously during a machine's normal operation. After an intrusion, the system administrators have to examine huge log files for connections from unusual network locations or unusual activity in the system. Tools such as Tripwire [18] [19] help system administrators by automating some parts of intrusion detection. A *detection point* refers to the state on the local computer that alerts the user of the intrusion [20]. Deleted or modified files, processes with unusual activity, *setuid* or *setgid* files, unauthorized entries in system configuration file or network configuration file, etc., are called detection points. A number of security tools are available to help detect intrusions, secure the system, and deter break-ins [1].

The logs provide a list of events that occurred prior to the intrusion, and often it is the system administrator's task to determine how these events relate to each other and how these events affect the detection point. For example, the attacker may have installed a back-door in one login session, and used that back-door to gain unauthorized access in a subsequent session. An installed *rootkit* is an example of such Unix intrusions [12]. The log will not indicate that these two sessions are related and extensive examination of the log is needed. An automated process for detecting dependencies between the various system events would greatly benefit intrusion analysis. BackTracker is such a tool, and was developed by King and Chen [20] to automatically identify the entry point used to gain access to a system and the sequence of steps leading to the detection point by displaying chains of events in a dependency graph.

We next present an overview of the BackTracker system and describe how it determines the sequence of events that affect a detection point.

1.1. BackTracker

BackTracker uses a modified Linux kernel to log events that describe dependencies between operating system objects [20]. There are two components: the online component, which runs and logs information when the computer is in operation and the off-line component, which analyzes the log after an intrusion. King et al [21] show how BackTracker can generate forward causal graphs that can be used to track other hosts that are involved in multi-hop intrusions.

While the various applications in the system are execut-

ing, BackTracker's online logger component logs external information associated with events such as invocations of system calls and the related OS-level objects like files and processes. Some events create dependencies. For example, writing to a file by a process causes the file (the sink object) to depend on the process (the source object). The dependency associated with an event is represented as *source* $\rightarrow$ *sink*. If process *P* writes to file *f*, the dependency created by this event is represented as $P \rightarrow f$. The *time threshold* for an object refers to the maximum time that an event can occur and still be considered relevant for that object. The time threshold of an object can be considered as a time instant t_{max} and all time instants *t* such that $t < t_{max}$. With the log of the objects and dependency-causing events, BackTracker's off-line graph generator component, namely *GraphGen*, constructs the dependency graph as follows. First, *GraphGen* is initialized with the object associated with the detection point and its time threshold is set to the earliest time that unusual contents/behavior was noticed in the detection point. *GraphGen* then reads the log of events in the reverse order of their occurrence starting from the time the detection point was identified. For each event *E*, *GraphGen* evaluates whether *E* can affect any object that is already in the dependency graph and whether the event has occurred within the time threshold of such an object in the graph. If the event *E* is determined to affect an object O_2 in the graph and the source object for this event, O_1, is not already in the graph, then O_1 is added to the graph. The time threshold of O_1 is set to the time *t* when event *E* occurs because *E* does not depend on events that occur after *t*. More details about the various events and dependencies tracked by BackTracker can be found in [20].

As an example, consider the sequence of events (from the log) shown in Table 1. The intrusion detection system determined that *File 7* has suspicious contents. Let the actual path of the attacker be: *File 2* $\rightarrow$ *Process B* $\rightarrow$ *Process D* $\rightarrow$ *File 6* $\rightarrow$ *Process F* $\rightarrow$ *File 7*. With *File 7* as the detection point, BackTracker constructs the dependency graph corresponding to the sequence of events of Table 1 as shown in Figure 1. In the graph, the label on each event shows the time of occurrence of the event.

We consider the problem of presenting a compact dependency graph to the system administrator to analyze. The dependency graph generated by BackTracker may be too large even after applying some of the suggested filtering rules. We propose two additional steps that can reduce the size of the dependency graph. First, we add additional fields to the log with details about file offsets where a read or write operation is performed, and use this information while generating the graph. Second, we propose the use of data flow analysis within processes to determine the events relevant to the detection point. Each of these two steps can reduce the graph size significantly. Data flow analysis has been used exten-

Time	Event
1.	*Process A* creates *Process B*
2.	*Process B* reads from *File 1*
3.	*Process B* reads from *File 2*
4.	*Process B* reads from *File 3*
5.	*Process B* creates *Process D*
6.	*Process D* writes to *File 4*
7.	*Process D* writes to *File 5*
8.	*Process D* writes to *File 6*
9.	*Process B* creates *Process C*
10.	*Process C* reads from *File 4*
11.	*Process C* writes to *File 7*
12.	*Process B* creates *Process E*
13.	*Process E* reads from *File 5*
14.	*Process E* writes to *File 7*
15.	*Process B* creates *Process F*
16.	*Process F* reads from *File 6*
17.	*Process F* writes to *File 7*
18.	*File 7* is identified as the detection point

Table 1. Sample Sequence of System Events

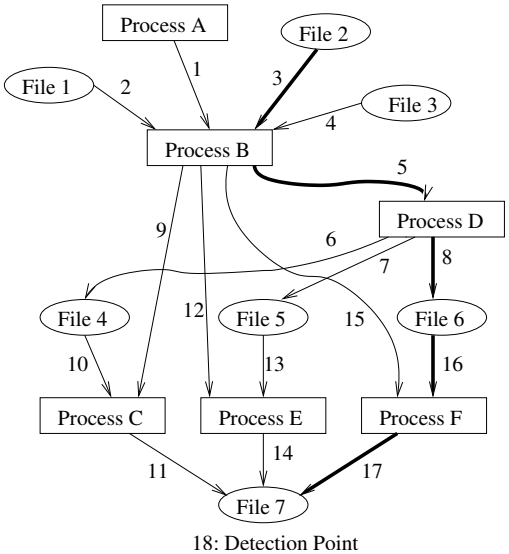


Figure 1. Dependency Graph generated by BackTracker

sively to optimize code during the compilation process [8]. Data flow information such as *reaching definitions*, that indicate which definitions of a variable may be used at a point of concern in a program, can be collected using static or dynamic methods, and this information can be used to reduce the number of possible paths in the dependency graph.

Our initial analysis shows that there is a moderate space overhead in the logging phase and time overhead in the analysis phase, but the resulting graph size is quite small when compared to the size of the graphs constructed without our optimization.

The rest of the paper is organized as follows: In Section 2, we describe the system model used, and Section 3 presents an overview of the main components of our tool. Our solution to the problem of large dependency graphs is presented in Section 4. Section 5 discusses ideas for implementation of the tool, and Section 6 analyzes the algorithm's performance, and limitations of our approach. Section 7 compares our contribution to related work, and Section 8 concludes the paper.

2. System Model

Consider a typical Unix system where the operating system, commonly called as the *kernel*, interacts directly with the hardware. The user applications use the system call interface to access the operating system's functions. Data is stored in *files* and files are organized on the hard disk in a Unix-based file system. The file system is accessed via system calls. The attributes of files such as the file's owner, file size, last modification time, etc., are stored in *inodes*. A *program* is an executable file, and a *process* is an instance of the program in execution [10]. Processes execute simultaneously in the Unix system, and system calls allow creation, termination, synchronization, etc., of processes.

There are two main components: an online *logger* and an off-line *graph generator*. The *logger* component can be built into the kernel or implemented as a loadable kernel module and executed along with other user applications. In the event of an intrusion, with the log of events and a detection point, the off-line *graph generator* constructs a graph of dependencies that relate to the detection point. The *graph generator*'s output is given to the system administrator for analysis of the intrusion.

We now describe the various objects, events and dependencies that are relevant to the backtracking tool. The backtracking tool is used for tracking file system based intrusions only. It tracks the flow of information between operating system objects and events. It does not track application level objects or events.

1. **Objects:**
   The OS-level objects that are tracked by the backtracking tool are files, processes and filenames. A *file* object is identified by its inode and contents. A *process* object is described by its unique PID and a version number. Every process except the swapper process is tracked from its creation to termination. Swapper is tracked from the time it makes the first system call. *Filename* objects indicate the absolute pathnames of files. These objects are affected by system calls such as *creat*(), *open*(), *unlink*(), etc.
2. **Events:**
   The system calls invoked by processes are the *events* that are tracked by *logger*. The system call events affect OS objects. For example, when a process writes to a file f using the *write*() system call, file f is the object affected by the event. When *logger* logs an event, it stores all the identifying information about the calling process and the affected object. In addition to that, *logger* stores the values of the parameters that were provided as input to the system call as well as the return value of the system call. As described in Section 4, this information will be used when the *graph generator* analyzes the trace and performs data flow analysis to reduce the size of the graph. Note that this logging operation can be performed by modifying the system call or by modifying the application/system program.
3. **Dependencies:**
   An event E results in a dependency between the OS objects that are associated with E. A dependency is represented as *source object* $\rightarrow$ *sink object*.
   Dependencies can be of the following types:
   (i) *Between two processes:*
   Such dependencies are created when one process creates another process (using the *fork*() system call) or sends a signal to another process. A dependency of the form *Process A* $\rightarrow$ *Process B* is created when *Process A* forks *Process B*. The same dependency is created when *Process A* sends a signal to *Process B* with the *kill*() system call.
   (ii) *Between a process and a file:*
   A *Process* $\rightarrow$ *File* dependency is created when the file's contents or attributes are affected by some action by a process. The *write*() and *chown*() system calls are examples that produce a *Process* $\rightarrow$ *File* dependency. A *read*() system call introduces a dependency of the type: *File* $\rightarrow$ *Process*. This dependency arises due to the fact that the process' actions may depend on the contents read from the file. As an example, a process A depends on a file F when process A reads data from file F, and this is represented as *File F* $\rightarrow$ *Process A*.
   (iii) *Between a process and a filename:*
   A *Process* $\rightarrow$ *Filename* dependency is created when the process invokes a system call that modifies a file-

name object. Examples of such system calls are *rename*(), *unlink*(), *creat*(), *link*(), *mount*(), etc. A *Filename* $\rightarrow$ *Process* dependency is created when a system call that takes a filename object as input succeeds. For example, system calls such as *open*(), *stat*(), etc., will not succeed if the file identified by the filename argument does not exist. In such cases, the *Filename* $\rightarrow$ *Process* dependency is not created.

2.1. Definitions

Following are some terms used in the subsequent sections.

Offset Interval in a File We define *offset interval*, $[a, b]$, $b \geq a$, in a file f as the set of byte offsets from a to b in f.

Suspicious Offset Interval A *suspicious offset interval* of a file f is an offset interval in f that is known to contain suspicious contents (for example, if file f is a detection point). Note that a file can have more than one suspicious offset interval. Let the set of all suspicious offset intervals of a file f be denoted by S_f. S_f, for a file f, is initialized with either the offset interval where suspicious contents are found (if f is a detection point) or is initialized to null (for all other files).

Offset Interval of Read/Write Operation The offset interval of a read or write operation is $[a, b]$ if the read or write operation was performed between offsets a and b in the file. For example, if a process reads n bytes from a file from an offset of 100 from the start of the file, then the offset interval of this *read*() system call is $[100, 100+n]$.

Overlapping Offset Interval Offset interval, w, of a write operation is said to be *overlapping* an offset interval s of file f if w and s have some common offsets, i.e., if $w \cap s \neq \phi$. For example, if a process writes to offsets [50, 100] of file f, it *overlaps* offset intervals [75, 125], [25, 60], [75, 80], etc., of file f.

Latest Overlapping Offset Interval For each offset b in a suspicious interval s of file f, we define the *latest overlapping offset interval* as the offset interval corresponding to the latest write that was performed at offset b in file f. If no write operation was performed at offset b in suspicious interval s of file f, then the latest overlapping offset interval for b is null. Note that a suspicious offset interval consists of one or more offsets, and hence may have multiple latest overlapping offset intervals.

3. Design of Backtracking tool

Figure 2 illustrates the various steps involved in backtracking a file system based intrusion. When the programs are executing, the *logger* component logs the system events. An intrusion detection system detects an intrusion either immediately after it occurs or at a later time. A detection point (a suspicious OS object) is identified. With the detection point and the trace of the system events (found in the log), the off-line *graph generator* component constructs the dependency graph. This dependency graph can be further pruned using the data flow analysis tool explained in Section 4.2.

We now explain the two main components of the backtracking tool in more detail.

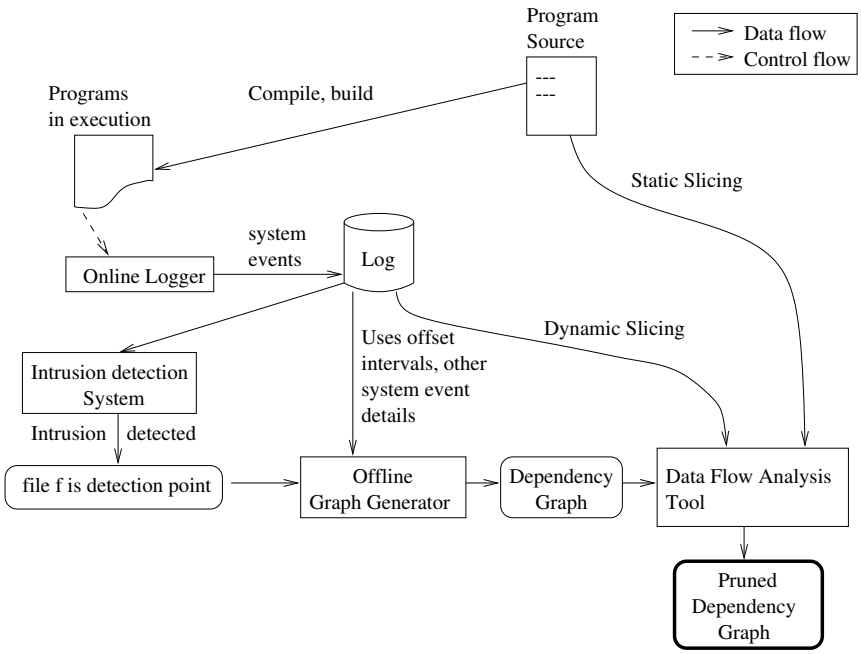
Logger:

Logging of system events and affected operating system objects while the system is executing constitutes an important phase of the tool, thereby enabling backtracking from the detection point to the entry point of intrusion. The responsibility of the *logger* is to monitor all the applications executing on the target system and log the various events occurring on the system at the OS level. The *logger* logs all the information needed to enable a replay of all the events, if possible, during the subsequent analysis and graph generation phase. This involves tracing the system calls executed by the various applications, and logging them along with the values of the input parameters and return values.

Graph Generator:

The *graph generator* is an off-line component of the tool that analyzes the log created by the online *logger* component. The detection point, indicated by an intrusion detection system or other tools, is used as the starting point for the analysis, and the log is examined 'backwards' i.e., from the last event to the first, in a manner similar to BackTracker. An outline of the graph generation algorithm as used in BackTracker is given below. Note that the dependency graph is initialized to the object corresponding to the detection point.

Initialize dependency graph to object corresponding to the detection point.
/ read events from latest to earliest, starting from the time when the detection point was identified by the intrusion detection system*/*
for each event E in log {
  let $E = O_1 \rightarrow O_2$
  if (O_2 exists in the graph and E affects O_2 by the time



threshold of O_2)
  if (object O_1 not already in graph) {
    add O_1 to graph
    connect O_1 to O_2 with directed edge $O_1 \rightarrow O_2$
    set time threshold of O_1 to time of E
  }
}

4. Improvements to BackTracker

The motivation for our work has been the reduction in the size of the graph generated by BackTracker. We now explain how to find unwanted paths in the graph. The two main improvements to the existing tool, BackTracker, are presented below.

4.1. Offset Intervals

We log the file offsets when read or write operations are performed. The offset intervals associated with the *read*() and *write*() system calls help in reducing the size of the resulting dependency graph. For instance, consider that *Process A* writes to *File 1* to offset interval [10, 20], and *Process B* reads from *File 1* subsequently from offset interval [50, 100] with no intermediate operation (by any process) that modifies the contents of *File 1*. In this case, there does not exist a dependency between processes *A* and *B* with the path: *Process A* $\rightarrow$ *File 1* $\rightarrow$ *Process B*. BackTracker would include such a path in the dependency graph because the write operation of *Process A* to *File 1* had happened within the time threshold of the read operation of *File 1* by *Process B*. We remove such dependencies from the graph using the offset interval information.

We propose changes to the graph generation algorithm when a write event or a read event is encountered in the log. The changes are described in the following sections.

4.1.1. Write Event The following checks are performed in the graph generation algorithm while examining a write event *E* from the log in order to use the available offset interval information. Let S_f be the set of all suspicious offset intervals of a file *f*. If *f* is the detection point, then S_f is initialized to the suspicious offset interval detected in *f*, otherwise, S_f is initialized to null.

if (*E* corresponds to a *write*() system call by process *P* on file *f*) {
  let $E = O_P \rightarrow O_f$
  if (O_f exists in graph and *E* affects O_f by time
  threshold of O_f)
    if (offset interval of *E* is a latest overlapping offset
    interval for some suspicious offset interval *s* of file *f*,
    i.e., $s \in S_f$)
      if (O_P not already in graph) {
        add O_P to graph
        connect O_P to O_f with directed edge, $O_P \rightarrow O_f$
        set time threshold of O_P to time of *E*
      }
}

4.1.2. Read Event When a read event *E* of process *P* from file *f* is encountered, the event is added to the graph if the object associated with *P* is already in the graph. We need to know what the suspicious offset intervals of a file are as it is not necessary to track write operations that write to a part

of the file that is never read from subsequently. So, in our backwards analysis of the event log, when we encounter a read operation, we add the offset interval of this read operation to S_f, the set of all suspicious offset intervals of file *f*.

The following actions are taken by the graph generation algorithm while examining a read event *E*.

```
if (E corresponds to a read() system call by process P from file f)
{
  let E = O_f → O_P
  if (O_P exists in graph and E affects O_P by time
  threshold of O_P)
    if (O_f not already in graph) {
      add O_f to graph
      connect O_f to O_P with directed edge, O_f → O_P
      set time threshold of O_f to time of E
      add offset interval of E to the set S_f of suspicious
      offset intervals of file f
    }
}
```

We present an example to better illustrate the benefits of using offset intervals. Table 2 shows the same sequence of events in the log of Table 1 with the corresponding offset interval information. Figure 3 shows the dependency graph of Figure 1 with offset interval information for the *read*() and *write*() system calls. Suppose the intrusion detection system indicated that *File 7* is the detection point with suspicious offset interval [50, 75]. Given the offset interval information for all the *read*() and *write*() system calls, our algorithm will produce a dependency graph as shown in Figure 4.

The following steps were taken while generating the dependency graph shown in Figure 4 for the sequence of events in Table 2.

1. Time 18: First, the graph is initialized with *File 7*, the detection point. The suspicious offset interval of *File 7* is [50, 75].
2. Time 17: *Process F* writes to *File 7* in offset interval [50, 100]. This write event's offset interval is determined to be a latest overlapping interval of the suspicious offset interval [50, 75] of *File 7*. So we add the source of this event, *Process F*, and an edge for the event to the graph. The time threshold of *Process F* is set to 17, the time of this event.
3. Time 16: *Process F* reads from *File 6*. The sink object *Process F* of the event exists in the graph constructed so far and the event's time is within the time threshold of the object of *Process F*. So, the source object of this event, *File 6*, is added to the graph. Also, the offset interval of this read event, [125, 175] is added to the list of suspicious offset intervals of *File 6*.
4. Time 15: The dependency *Process B* → *Process F* is added to the graph as *F* is already in the graph and the event occurs within the time threshold of *Process F*.
5. Time 14: *Process E* writes to *File 7* in offset interval [100, 125]. This offset interval does not overlap the suspicious offset interval of *File 7*, [50, 75]. So the dependency corresponding to this event is not added to the graph.
6. Time 13, 12: Since *Process E* is not in the graph, the events occurring at times 13 and 12 are skipped.
7. Time 11, 10, 9: The write event at time 11 by *Process C* to *File 7* does not have an offset interval that overlaps the suspicious offset interval of *File 7*. So, *Process C* is not added to the graph, and the next two events at times 10 and 9 are also skipped.
8. Time 8: *Process D* writes to *File 6* in offset interval [100, 150]. *File 6* is in the graph, and its only suspicious offset interval [125, 175] overlaps with this event's offset interval. So, *Process D* is added to the graph, with the time threshold set to 8.
9. Time 7, 6: Both *File 4* and *File 5* are not found in the dependency graph, so these events (at time instants 7 and 6) are skipped.

Time	Event
1.	*Process A* creates *Process B*
2.	*Process B* reads from *File 1*, [50, 55]
3.	*Process B* reads from *File 2*, [65, 200]
4.	*Process B* reads from *File 3*, [75, 125]
5.	*Process B* creates *Process D*
6.	*Process D* writes to *File 4*, [300, 350]
7.	*Process D* writes to *File 5*, [25, 30]
8.	*Process D* writes to *File 6*, [100, 150]
9.	*Process B* creates *Process C*
10.	*Process C* reads from *File 4*, [5, 10]
11.	*Process C* writes to *File 7*, [150, 175]
12.	*Process B* creates *Process E*
13.	*Process E* reads from *File 5*, [15, 20]
14.	*Process E* writes to *File 7*, [100, 125]
15.	*Process B* creates *Process F*
16.	*Process F* reads from *File 6*, [125, 175]
17.	*Process F* writes to *File 7*, [50, 100]
18.	*File 7* is identified to have suspicious contents in [50, 100]

Table 2. Sequence of System Events with Offset Intervals

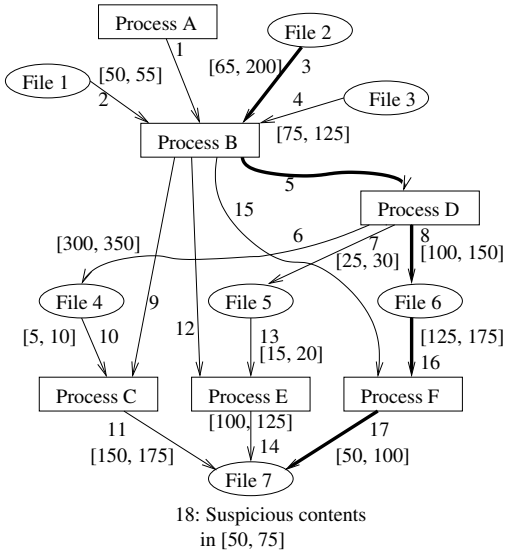


Figure 3. Graph of Figure 1 with offset interval information

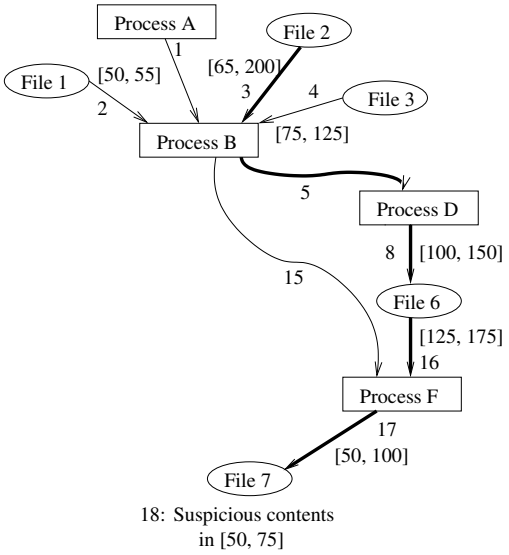


Figure 4. Smaller dependency graph constructed using offset intervals

10. Time 5: *Process D* exists in the graph, so *Process B* is added to the graph for this event.
11. Time 4, 3, 2: Objects for *File 1*, *File 2* and *File 3* are added to the graph. The offset interval of each of these read events is added to the respective file's list of suspicious offset intervals.
12. Time 1: *Process A* is added to the graph as *Process B* is found in the graph.

The dependency graph of Figure 4 has fewer objects and dependency edges than the graph of Figure 1.

4.2. Data Flow Analysis

In the reduced dependency graph of Figure 4 that was obtained after applying offset intervals information, note that *Process B* had read data from *File 1*, *File 2*, and *File 3*. The suspicious content that *Process B* passed on to *Process D* in a buffer variable was written to *File 6* by *Process D*. While analyzing the events in the log, it is not clear from where *Process B* obtained the suspicious data (and then passed to *Process D*). The possible data sources are the three files, *File 1*, *File 2* or *File 3*, some internal definition of the data buffer or user data read by *Process B*. We use program slicing techniques on the programs of *Processes D* and *B* to determine the reaching definitions for the value of the buffer variable used in that *write*() system call. This approach reduces the search space further. Note that this may require the source code of the programs involved.

The process *global data flow analysis* is used in order to determine how data flows through the various application program [8]. In data flow analysis, the definition d is said to *reach* a point p if there is a path in the program from the point immediately following d to p, such that d is not "killed" along that path. The definition of a variable a is said to be *killed* if between two points along the path there is a read of a or an assignment to a [8].

A *program slice* refers to the set of statements of the program that influences the value of a variable at a point in the program. The initial notion of a program slice was a *static slice* obtained irrespective of the values of the input variables. Given a variable, a program location and a set of values for all input variables, the task of determining which statements in the program affected the value of that variable at that location is referred to as *dynamic program slicing* [4] [6].

Static program slicing techniques can be used to reduce the size of the dependency graph but they may be very conservative and inefficient. A static analysis of the source code of a program cannot indicate what execution paths the program may have. Hence, static program slicing may present more reaching definitions for a variable's value than the actual set of definitions that could possibly be executed. Dynamic slicing takes a particular test case, represented by a known set of values for all the input variables, and evaluates the program thereby yielding more probable program slices.

In the example shown in Figure 4, after applying dynamic slicing on the programs of *Process D* and *Process B* with the trace generated by *logger*, suppose that the only reaching definition (of the buffer variable used in the write

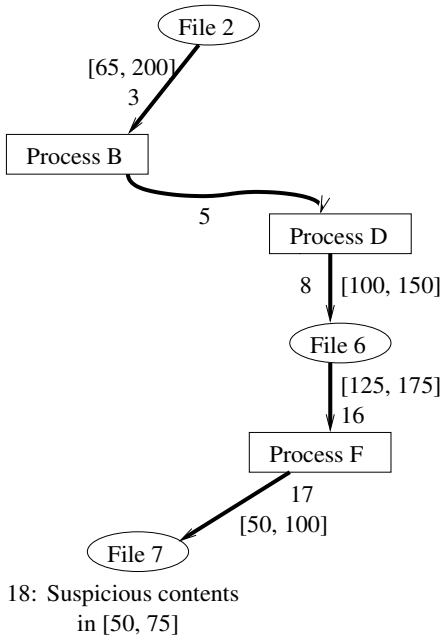


Figure 5. Dependency Graph after applying Dynamic Slicing

event at time 8) is determined to be the *read*() system call invoked by *Process B* to read contents from *File 2*. This would yield the smaller dependency graph shown in Figure 5. Note that to perform program slicing for the buffer variable of a particular write event, the location in the program of that *write*() system call, in addition to the values of all the input variables, is needed. The location (line number) of the *write*() system call in the program, corresponding to an event, can be determined by techniques described in Section 5.

5. Implementation Ideas

The component *logger* of our tool can be built as part of the Linux kernel or can be implemented as a Linux loadable kernel module. OS Virtual Machines that were introduced to enable time-sharing of expensive hardware are now used in a variety of applications [14] [15]. The secure and fine granular logging of system execution that virtual machines provide can be used in intrusion detection [13]. User applications can be executed on the host system or inside a target system which itself is running as a user application on the host system. In the latter case, the target system is a host process that transfers control to *logger*, a Virtual Machine Monitor (VMM), whenever a system call is invoked by a user application in the target system. Whenever *logger* is invoked, it examines the host system for the state of the target system process and extracts information about the event and the affected objects.

Whenever a *write*() system call is invoked, *logger* needs to print the line number of the *write*() system call in the source program. The line number information is needed to perform data flow analysis during the graph generation phase of the backtracking tool. In order to make the line numbers available to *logger*, this information is added to a section of the object file during compilation of the user application. The compiler has to be enhanced to perform this operation and all the user applications have to be compiled using this enhanced compiler. The *-g* option of *gcc* can be used to produce debugging information in the operating system's native format [26]. The ability to read the required line number details from the debugging information section of the object file can be built into *logger*. Similar capability can be found in the Gnu Debugger (GDB) application [27].

6. Performance and Limitations

We analyze the overheads of time and storage space in the two phases of our backtracking tool, namely logging and graph generation, and compare the time and space requirements of our improved backtracking tool with existing tools such as BackTracker.

Logging Phase:

Our *logger* component needs to log more information that enables the *graph generator* to replay the events and perform data flow analysis. The additional information consists of the actual values passed to the system calls, and the return values of the system calls. The logger in our backtracking tool also needs to obtain the line number information from the relevant sections of the object file whenever a *write*() system call is encountered.

Graph Generation Phase:

With the improved graph generation algorithm, the size of the dependency graph will be reduced. Hence, with fewer paths to track, our backtracking tool generates the final dependency graph faster. Dynamic slicing will produce good results by pruning more unwanted paths in the graph, but at the expense of additional time in the graph generation phase.

A limitation of our approach is the storage overhead during the logging phase. The detailed traces are needed in order to replay the various programs and determine program slices dynamically. We can reduce the space required by the log by choosing static program slicing instead of dynamic

program slicing. Note that in order to perform static data flow analysis on a program, the source code for that program is required. Also, execution backtracking has its limitations since it may not be feasible to undo certain actions of the applications and replay them. In such cases, dynamic slicing techniques may produce conservative results.

7. Related Work

The idea of using causality of system events has been exploited by other research projects such as King and Chen's BackTracker [20] and the Repairable File System [33]. In Repairable File System, a forward analysis from the detection point is performed to determine the operating system objects that were affected by the system intrusion. Ammann, Jajodia and Liu [9] have proposed techniques to detect the flow of contaminated transactions through a database and roll back those transactions that are affected directly or indirectly by contaminated transactions. In contrast, our intrusion detection tool backtracks from a detection point to determine the entry point of an intrusion. Only OS-based objects and events are tracked.

BackTracker performs forward as well as backward analysis of operating system events to detect the flow of information from and to the detection point. BackTracker filters the resulting graph to prioritize likely paths of an intrusion, but in doing so, may hide important sequences of events. Our work, in comparison, performs backward analysis from a detection point. With moderate storage overhead, the improvements suggested in this paper aim to reduce the size of the dependency graph while retaining the important sequences of events that led to the detection point.

Weiser [29] [30] introduced the concept of program slicing to make debugging of programs easier. Many techniques to obtain static program slices were proposed in [11] [17] [25]. In addition to debugging, program slices are useful in testing, maintenance, and understanding of programs [4] [6]. Techniques such as PSE (Postmortem Static Analysis) by Manevich et al [24] use information about a program failure (such as the kind of failure, and its location in the program's source code) to produce a set of execution traces along which the program can be driven to the given failure. Notions of dynamic program slicing were proposed by Korel and Laski [22] and Agrawal and Horgan [7]. Agrawal et al present a debugging model, based on dynamic slicing and execution backtracking techniques, that easily lends itself to automation in [5]. Dynamic slicing techniques are still topics of active research [31] [32]. Surveys of the various program slicing methods are presented in [16] [23] [28]. In this work, we have applied dynamic slicing techniques for detection of file system based intrusions.

8. Conclusion

Analyzing an intrusion with huge log files is an arduous task for system administrators. The system administrator must determine how the intruder gained access to the system and conducted the attack. Existing tools, like BackTracker, provide a dependency graph of the events in the log that relate to the intrusion. In this paper, we have addressed the problem of reducing the size of the dependency graph. We have proposed two improvements to the graph generation algorithm, which when applied properly can result in significant reduction in search space and search time.

References

[1] CERT/CC List of security tools. 2001, http://www.cert.org/tech_tips/security_tools.html.

[2] CERT/CC Overview of attack trends. 2002, http://www.cert.org/archive/pdf/attack_trends.pdf.

[3] CERT/CC Overview: Incident and vulnerability trends. 2003, http://www.cert.org/present/cert-overview-trends/.

[4] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the symposium on Testing, analysis, and verification*, pages 60–73. ACM Press, 1991.

[5] H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software - Practice and Experience*, 23(6):589–616, 1993.

[6] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256. ACM Press, 1990.

[7] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256. ACM Press, 1990.

[8] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[9] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1167–1185, September/October 2002.

[10] M. J. Bach. *The Design of the Unix Operating System*. Prentice Hall PTR, 1986.

[11] J.-F. Bergeretti and B. A. Carr. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):37–61, 1985.

[12] M. Bishop. Unix security: Security in programming. *SANS'96*, 1996.

[13] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[14] J. Grizzard, E. Dodson, G. Conti, J. Levine, and H. Owen. Towards a trusted immutable kernel extension (tike) for self-healing systems: a virtual machine approach. In *Proceedings of the Fifth IEEE Information Assurance Workshop (IAW)*, pages 444–446. IEEE, June 2004.

[15] J. Grizzard, S. Krasser, G. C. Owen, and E. Dodson. Towards an approach for automatically repairing compromised network systems. In *Proceedings of the IEEE Symposium on Network Computing and Application's Workshop on Trustworthy Network Computing (IEEE-NCA)*, August 2004.

[16] T. Hoffner, M. Kamkar, and P. Fritzson. Evaluation of program slicing tools. In *Proceedings of Automated and Algorithmic Debugging*, pages 51–69, 1995.

[17] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[18] G. Kim. Advanced applications of tripwire for servers: Detecting intrusions, rootkits and more.. *Whitepaper, Tripwire*.

[19] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: A file system integrity checker. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[20] S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 223–236. ACM Press, 2003.

[21] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching intrusion alerts through multi-host causality. In *Proceedings of 2005 Network and Distributed System Security Symposium (NDSS) (to appear)*, 2005.

[22] B. Korel and J. Laski. Dynamic slicing of computer programs. *The Journal of Systems and Software*, 13:187–195, 1990.

[23] A. D. Lucia. Program slicing: Methods and applications. In *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149. IEEE Computer Society Press, Los Alamitos, California, USA, Nov 2001.

[24] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. Pse: Explaining program failures via postmortem static analysis. In *Proceedings of SIGSOFT'04/FSE-12*. ACM Press, 2004.

[25] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184. ACM Press, 1984.

[26] R. M. Stallman. *Using and Porting GNU CC, Version 2.8.1*. 1995.

[27] R. M. Stallman. *GDB Manual, Version 6.2.1*. 2004.

[28] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[29] M. Weiser. Programmers use slices when debugging. *Communications, ACM*, 25(7):446–452, 1982.

[30] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[31] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.

[32] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*, pages 319–329, May 2003.

[33] N. Zhu and T. Chiueh. Design, implementation, and evaluation of repairable file service. *Proceedings of The International Conference on Dependable Systems and Networks*, June 2003.