A Declarative Framework for Intrusion Analysis

Matt Fredrikson

Computer Sciences Department
University of Wisconsin – Madison

Abstract

In this paper we consider the problems of computer intrusion analysis, understanding, and recovery. More specifically, we identify the various difficulties associated with these problems, and propose a solution based on system events and the dependencies between them for simplifying our motivating problems. We then present a tool we have designed called SLog that presents a system administrator with all of the facilities required to analyze the event information present in system logs, and present only the event information pertinent to an intrusion in a vastly simplified manner. We discuss the ways in which the major design goals of SLog – simplicity, extensibility, and scalability – are important when dealing with intrusions in realistic scenarios. Finally, we demonstrate the ability of SLog to accurately return a simplified view of the relevant events in a realistic intrusion case study.

1 Introduction

Intrusions on computer systems remain a formidable menace to the substantial infrastructure present in today's public and private networks. The CERT coordination center has remained active in the task of following events related to computer intrusions for the past several decades, and has compiled a list of observed trends in this domain. Among the more alarming trends are an increase in automated attacks, increasing sophistication in attack tools, faster discovery of important system vulnerabilities, and an increasing focus on infrastructure attacks [5]. These findings indicate that we can expect intrusions to become more frequent and more sophisticated.

This poses several challenges for network administrators charged with the task of preventing attacks from occuring in the first place, as well as detecting and recovering from the attacks that will inevitably occur. There is a sizeable body of literature, as well as an active research community, that focuses on the first two problems of prevention and detection. Prevention measures include timely patch updates to vulnerable software, security-conscious operating system design [1, 6, 10], and network perimiter security devices such as firewalls. Detectoin measures are diverse and numerous, ranging from manual analysis [4] to fully automatic intrusion detectors [13, 7].

The problem of intrusion recovery, on the other hand, has not been studied as extensively by the academic community. This is certainly not for lack of interesting or substantive problems in the area. An administrator is faced with two important problems after an intrusion. First, he must determine the cause of the intrusion, which in most cases means discovering which process on the system allowed the intruder an undesired level of access. This step of the recovery is essential, as without it, the administrator must expect a similar intrusion at some point in the future. Next, a prudent administrator must discover all entities on the system that may have been compromised or affected in some way by the intrusion. Without this step, no guarantees can be made regarding the integrity of the system's contents.

Having underscored the importance of proper intrusion recovery, we now focus our attention on some of the difficulties involved with recovery on a real system. We observe that in a typical network setting, an administrator may have numerous sources of relevant information available, including application logs, intrusion detector logs at both the host and network level, as well as the entire state of the persistent store. Furthermore, each of these sources can potentially contain an overwhelming amount of information. In an enterprise setting, it is not unreasonable to expect log files with gigabytes of data. Clearly, culling the relevant information from these sources is a difficult, time consuming, and error-prone process.

The fundamental problem here is that while these sources may contain all of the information that is required to successfully recover from an intrusion, this information is effectively drowned out by the high degree of operational noise generated by everyday, legitimate use of the system. The task assigned to our hypothetical system administrator would be made much simpler if there were a way for him to automatically filter the irrelevant information that corresponds to legitimate activity, and focus only on those details that pertain to the intrusion at hand. This is the problem that we will address in this work.

This paper describes a tool called SLog that attempts to filter out the irrelevant portions of an event history, leaving only those portions that provide information that may be valuable when attempting to understand and recover from a specific intrusion. SLog does this by reasoning about the system in terms of the events that are reported to have occurred on it, as well as the dependencies between these events. The primary goal of SLog is to provide a simplified view of the objects and events that have been affected in some way by the intrusion. However, it is crucial that this task is performed without over-simplifying the analysis in such a way that relevant events are discarded from the final report, leaving the administrator without a full picture of the effects of the intrusion. In an attempt to realize these conflicting goals, SLog provides facilities for viewing and reasoning about events at different conceptual granularities.

The rest of the paper will proceed as follows. Section 2 gives an overview of the foundational principles, goals, and design decisions that were made while constructing SLog. Section 3 describes in some detail the various language constructs used in SLog to implement our design decisions, as well as their syntax and semantics. Section 4 presents a case study involving a realistic intrusion scenario to which we applied SLog, as well as a discussion of the results we collected from this case study. Section 5 discusses some important related work, and Section 6 concludes the paper and hints at some directions for future work.

2 Design

Having motivated the need for a system such as SLog, we describe the overall design of our tool in this section. We begin by giving a more thorough description of the goals we hope to achieve with such a system, continue by describing the high-level operational workflow of the system, and conclude by enumerating the specific components that we implemented to realize this workflow.

2.1 Design Goals

The basic functional goal of SLog is to provide a user who is faced with the difficult task of understanding a system intrusion with a simplified view of the events and objects on a system and the ways in which they may be related to the intrusion. To construct such a tool, we rely on several key observations and principles.

Our first observation is that understanding, and in turn recovery, will come about much more quickly if the user is able to easily distinguish the events and objects that are related to an intrusion from those that are not. As such, we would like our tool to allow the user to create a compact summary of all the intrusion-relevant events that have occurred, while excluding all events corresponding to everyday legitimate behavior.

The second such observation is that the system that has been intruded upon may contain several different sources of information, each relevant to the events that transpired in some way. Furthermore, we do not wish to assume that the user was aware of our tool before the intrusion occurred. This anti-assumption rules out the possibility of reliance on event information provided by a system monitoring facility of our own construction. Therefore, we would like our tool to be able to use any and all event information that the user may be able to provide. Finally, we would like our system to incorporate these disparate information sources with very little effort to the user – without recompilation or significant format transformation cost.

The third observation that guided our efforts when designing SLog is that events from system and application logs are oftentimes complex and numerous. For example, logging system call invocations made by Firefox over a one-minute time period resulted in over 340,000 events. To make matters worse, each such event is associated with a fair amount of data in the form of argument information that requires a non-trivial amount of knowledge about the workings of the operating system. Even if our tool is capable reducing the logs to only those events related to the intrusion, the user is left with a great deal of analysis work if he is forced to think in terms of these events. We would like our system to offer facilities for further reducing the amount of information about intrusion-relevant events that is ultimately presented to the user.

Our fourth guiding principle is rooted in the adversarial nature of the analysis we wish to perform. As we design the facilities which allow the user to selectively control the information that is presented, we must proceed with caution, as poor choices in this area could create opportunities for an attacker. Specifically, we observe that certain simplifications, or reductions in the amount of presented information, could remove from the user's view a sequence of events that are necessary for an adequate understanding of the intrusion at hand. Assuming the attacker is aware of our analysis, this gives him a specific portion of the system to focus on as a vector for his intrusion, as he is congnizant of the fact that we will simply ignore it. With this in mind, we seek simplifications that summarize the given events, rather than those that purposefully ignore certain events.

2.2 Simplification Workflow

To imagine how we might go about fulfilling our desired goals as stated in Section 2.1, we begin by laying out a general workflow for the simplification process. We present our workflow in Figure 1. The first step corresponds to the collection and subsequent selection of system event information from a number of sources. This step must be executed with care, as we would like to have available all of the possibly relevant event information, but if too much is available, it may bog the analysis down. We envision this step as being performed by the user.

The second step in the workflow is the normalization of the information sources. This is necessary, as we do not want to limit our tool to any fixed set of information sources and formats for compatability and extensibility reasons. Therefore, it is necessary for our tool to extract the necessary information from the sources, and store it in memory in an easily accessible format that is conducive to our further analysis. We envision this step as a joint effort between our tool and the user; the user specifies information about the format of the information sources, and our tool parses these specifications and performs the extraction.

The third step in our workflow is perhaps the most crucial. After the information sources have been parsed and loaded, we perform the task of analyzing the structure and semantics of the events. Once this analysis is complete, we use it to partition the events and objects on the system into those that are necessary to gain an understanding of the intrusion, and those that are not. This step is performed largely by our tool, although it will likely require some basic information from the user.

The final step in our workflow corresponds to the final simplification on the event data that our tool performs. Namely, once the relevant events have been extracted from the information sources, the user may require that the events be presented in a summarized or further simplified form than they appear in the logs. As in the first analysis step, this task is performed by our tool for the most part, according to a user-defined specification.

2.3 System Components

Having stated our design goals and laid out a general plan of attack for performing our desired analysis, we now give a brief overview of each operational component of SLog.

General Overview Observing the workflow presented in Section 2.2, we see that one factor is present in every step – input from the user. This is necessary to achieve the degree of flexibility mandated by our design goals. However, its presence in every step of our analysis suggests that our tool must be designed around it. This leaves us two options. We can design a tool that is highly interactive, and prompts the user for information as the analysis is underway. Alternatively, we can design a tool that is based on a lightweight programming language. We opted for the second alternative, as we envision the use of our tool in batch environments, where unnecessary interaction that is required of the user is nothing more than an annoyance. As SLog is essentially a lightweight programming language, each of the components we describe will correspond to a construct in the programming language. This fact will be made more explicit in later sections.

Information Extraction Essential to fulfilling our goal of independence from any particular data source is the ability to extract meaningful information from disparate sources of varied formats. We begin our discussion of this problem by stating a few of our requirements for an information extraction solution.

Most importanly, the extraction technique must scale well to very large data sources. As we would like to allow the user to reason about arbitrary events, it may often be the case that the events of interest occur frequently and are information-rich. Our second requirement is for the extraction technique to be relatively straightforward for the user to work with, even intuitive if at all possible. Our third requirement may seem obvious, but we would also like the extraction to return perfect information as specified by the user. We explicitly state this requirement due to the recent use of machine learning for the task of information extraction [15].

We selected the declarative information extraction techniques developed by Shen *et al.* [14]. Our reasons for selecting this technique fall in one-to-one correspondence with our stated requirements of an extraction technique. First, in their exposition of the technique, the authors demonstrate the ability of this technique to scale to sizeable datasets. This is accomplished through aggressive workflow optimizations of an extraction program based on the underlying statistics of the dataset. Beyond the optimizations presented by Shen *et al.,* the ability to optimize declarative programs in data intensive settings is a commonly scited reason for their use [3]. Second, the declarative extraction specifications are simple to construct and understand. Although this claim is based on subjective opinion, in our personal experience it is a frequently held sentiment. Finally, the ability of this technique to return precise, complete results follows from the well-defined semantics of the Datalog language, as we will see.

Event Analysis and Extraction As stated in our goals, we would like to present the user with only those events and objects that are relevent to the intrusion at hand. Doing so requires some notion of the relationships between various events. In other words, we can reduce this problem to the task of inferring which events are related to certain events that are *obviously* and *undeniably* part of an intrusion.

This leads us to the most important design decision of SLog, namely the selection of a particular relation be-

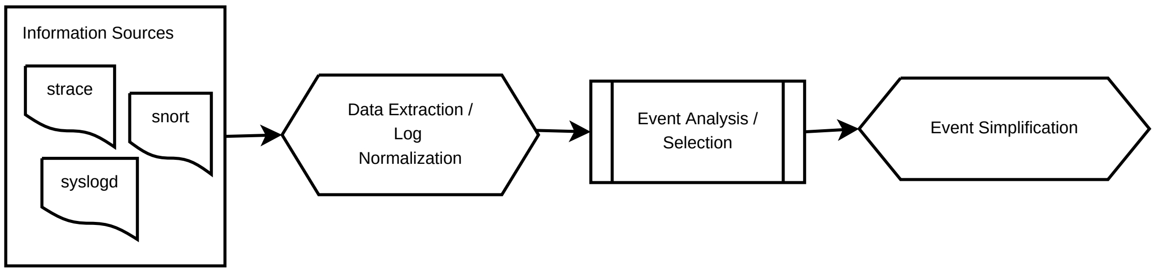
Information Sources
strace
snort
syslogd
Data Extraction / Log Normalization
Event Analysis / Selection
Event Simplification

Figure 1: Workflow diagram for the simplification process.

tween events to use in our analysis. We use the notion of *data dependence* between events for two reasons. First, a data dependence of event A on event B implies the ability of B to exert some kind of control on A. This is an important notion when reasoning about intrusions, as an attacker must be able to leverage vulnerabilities in applications to exert control over sensitive portions of the system if he is to execute any sort of intrusion that concerns us. Our second reason for using data dependencies is the fact that we expect our information sources to provide us with plenty of event *data*, if nothing else! There is always virtue in using that which is already available.

To reliably and accurately compute dependencies between the events we are given, we require some additional semantic information from the user. Rather than relying entirely on value equivalence when inferring dependencies, we require three additional pieces of information:

1. *Type Information*: We only reason about the equality between two pieces of data if those data are of the same type. On a typical system, a type may correspond to file descriptors, file names, or process identifiers, among others.

2. *Kill Events*: We say that certain events kill certain pieces of data when they reference them. This corresponds to the notion of invalidating the future use of a particular data value. For example, the `close` system call kills the file descriptor that is passed to it, so events that use that file descriptor at a later time are not performing valid actions.

3. *Definition and Use Information*: We further refine our notion of dependence by incorporating definition and use information into the calculations. In order for A to be dependent on B, A must *use* a piece of data that B *defines*.

The use of this additional information greatly increases the precision of our dependence calculations, as well as the user's control over which data elements are used in the dependence calculations. The type and kill information reduce the number of spurious dependencies calculated by SLog, while the definition and use information allow the user to precisely specify that only a subset of the available event data is actually relevant in the dependence calculations.

We now state the rules for dependence calculation. Let $Data(A)$ be the set of all data elements attached to event A, $use(A)$ correspond to the set of all data elements used by A, $def(A)$ correspond to the set of all data elements defined by A, and $kill(A)$ correspond to the set of all data elements killed by A. Furthermore, as we are reasoning about events that presumably occur on a continuous timeline, we attach a time value to each event, $Time(A)$. Then we say that event A is data-dependent on event B ($depends(A,B) = True$) if

- $\exists\ d_A \in Data(A),\ d_B \in Data(B)$ such that $d_A \equiv d_B$
- $type(d_A) = type(d_B)$
- $d_A \in use(A)$ and $d_B \in def(B)$
- $Time(A) > Time(B)$
- $\nexists$ event C such that $d_C \in Data(C)$, $d_C \equiv d_A$, $d_C \in kill(C)$, and $Time(B) < Time(C) < Time(A)$

Intuitively, these rules require that for one event to be dependent on another, the two events must contain a piece of data of the same type and value, and that the dependence-inducing piece of data isn't killed between the occurence of the two events.

After inferring dependencies between the events available to our analysis, the task of selecting only the events that are relevant to the intrusion at hand is a fairly straightforward task. We borrow the concept of a *backwards slice* from the program analysis literature to do so. For our purposes, a backwards slice simply corresponds to the transitive closure of the *depends* relation specified above, starting at a specified event. More precisely, we say that event S is in the backwards slice of event A if

- $depends(A,S)$ is true, *or*
- $\exists\ S_1,\ldots,S_n$ such that $depends(A,S_1) = True \wedge depends(S_1,S_2) = True \wedge \ldots \wedge depends(S_n,S) = True$

Given some event E that is clearly and unquestionably a symptom of an intrusion, we can constrict our view of the events to only those events that may have caused this suspicious event by doing a backwards slice on E.

2.4 Event Simplification

As stated in our design goals, we would like to provide the user with facilities that allow him to simplify his view of the relevant events that occured due to the intrusion. Furthermore, we require that these facilities do not correspond to *lossy* simplifications, or simplifications that might result in the loss of pertinent event information. These requirements significantly narrowed our range of choices, primarily to the set of *summary filters*. In other words, rather than pruning certain events from our view, we opted to summarize sequences of events that are available to us.

We refer to this simplification technique as "semantic abstraction", as our goal is to summarize related sequences of events in such a way as to create an abstracted view of the events that preserves the semantics of the underlying events. The most useful abstractions for the purpose of simplification are those that correspond to everyday system entities that people are familiar with. For instance, we might consider abstracting sequences of file-related system call events into file objects and dependencies between the processes that invoke the system calls.

We give the user the ability to define event abstractions by allowing them simple syntactical constructs in the SLog scripting language for specifying new events and objects in terms of those that have already been inferred. Furthermore, to ensure that dependence information is not lost through this abstraction, we provide them with a similar construct for specifying customized dependence relationships between abstracted events, beyond the default data dependence relations discussed in the previous subsection.

3 The SLog Language

We now describe how each of the components described in Section 2.3 is represented in the SLog language, as well as the manner in which each of these constructs is interpreted and used in the analysis performed by SLog.

3.1 Language Constructs and Syntax

SLog is a declarative language in the spirit of other popular declarative languages such as Prolog and Datalog. The syntax of SLog is derived from that of Datalog. It is useful to think of there being three first-class entities in an SLog program.

- *Events/Objects* Events can correspond either to concrete event instances extracted from the data sources, or to abstracted events defined in terms of previously inferred events and objects.
- *Data* All events are attached to a finite amount of data. In SLog, data instances are instantiated with event and type specifications.
- *Dependence Relations* Besides the default dependence relation induced by data instances, custion dependence relations may be instantiated at runtime in terms of previously inferred events, data, or dependence relations.

There are seven distinct types of statements recognized by SLog.

- *Event Abstraction Specifiers* These statements are used to specify the fact that *event* belongs to a set of events corresponding to a particular *level* of abstraction. By convention, concrete log entries correspond to $level = 1$.
  $level(event, level)$.
- *Type Specifiers* Given an event named *event*, we specify the type information of its associated data with a type specifier statement in the following manner:
  $eventT(type_1, type_2, \dots, type_n)$.
- *Kill Statements* We specify that *event* kills it's *n*th parameter, which is of type *type*, in the following way:
  $kill(event, type, n)$.
- *Def and Use Statements* We specify that *event* uses its *n*th argument, which is of type *type*, in the following way:
  $use(event, type, n)$.
  And similarly, for data defined by *event*:
  $def(event, type, n)$.
- *Custom Dependence Relations* Custom dependence relations can be specified with a fair amount of freedom. Basically, they correspond to a typical Datalog rule statement, where the head must be of the form $depends(event_1, id_1, event_2, id_2)$. Here, $event_1$, $event_2$ correspond to valid event instances as specified by an event specification statement. id_1, id_2 correspond to unique identifiers for the events. In the case of concrete log events, this identifier is usually a time.
- *Event Descriptors* Event descriptors are the core of SLog. They are used to extract event information, including associated data, from logs, as well as to define abstracted events in terms of previously inferred

```
open(time_t, file_t, string_t, handle_t).
read(time_t, handle_t, string_t, int_t, int_t).
close(time_t, handle_t, int_t).

def(open,'handleType','4').
use(read,'handleType','2').
kill(close, handle_t).
level(open,'1').

open(?time,?file,?perms,?ret) :- docs(?d), lines(?d, ?line), time(?line, ?time),syscall(?line, 'open'),
    arg(?line, '1', ?file),arg(?line, '2', ?perms), retval(?line, ?ret).
read(?time,?file,?data,?len,?ret) :- docs(?d), lines(?d, ?line), time(?line, ?time),syscall(?line, 'read'),
    arg(?line, '1', ?file), arg(?line, '2', ?data), arg(?line, '3', ?len), retval(?line, ?ret).
close(?time,?file,?ret) :- docs(?d), lines(?d, ?line), time(?line, ?time), syscall(?line, 'close'),
    arg(?line, '1', ?file), retval(?line, ?ret).

?-slice('close,'11:05:35.466110').
```

Figure 2: Example SLog program for file-related activities.

events and data. They are of the form:
$event(data_1,\ldots,data_n)$:-
$ext_1(data_1),\ldots,ext_m(data_{n-1},data_n)$.
Here, ext_1, ... ,ext_m correspond to external procedural predicates that are invoked by SLog as the statement is evaluated. These procedural predicates extract a portion of data from the information sources, and are the primary innovation presented in the work of Shen *et al.* [14]. It should be noted that the body of event descriptors is not restricted to any particular form, as long as extraction predicates are used.

- *Query Statements* Currently, we only allow query statements corresponding to backwards slices from a particular event. We will add more in the future. A query statement is of the form:
?- $slice(event,id,?eventN,?idN)$.

We present an example SLog program in Figure 2. This program extracts strace log events corresponding to common file-related activities, and queries SLog for a backwards slice on a specific instance of the `close` system call.

3.2 Semantics

The semantics of an SLog program can be understood in terms of transformations that are performed on each of the inferred events, and applications of the dependence relations to these transformed event instances. Evaluation proceeds as follows:

1. Event instances are instantiated by executing all of the event description statements. This is done using standard Datalog program evaluation algorithms [2], except external programs must be invoked for each extractor used in an event description statement.

2. Event instance tuples are transformed into vertical tuples, one for each event instance and associated data instance.

3. Query statements are executed over the transformed vertical tuples. The results of the query are output in a graph description language such as Graphviz DOT.

Event Instance Transformation The transformations applied to inferred event instance tuples essentially correspond to a vertical partitioning of the tuples. Specifically, a new set of tuples is created, with attributes corresponding only to events and event identifiers (as previously mentioned, event identifiers typically correspond to the time at which an event occurs). Furthermore, a new set of tuples is created for each data instance attached to an inferred event. A template for the transformation semantics is presented in Figure 3. Structuring the data in this way allows dependencies between events to be inferred using a small number of very general rules (inference rules are specified in the Appendix).

Event Inference Semantics It is reasonable to question the validity of applying Datalog evaluation rules to a program that can invoke an external, Turing-complete procedure as though it were nothing more than a ground clause. Shen *et al.* show that the semantics of such an evaluation can be understood in terms of typical least-model semantics as long as each procedural predicate behaves as though it represents a finite, deterministic relation [14].

Query Semantics Executing a query implies inferring all of the relevant event dependencies to compute the requested backward slice. This is performed using Datalog fact inference over the transformed, vertically-partitioned tuples computed in Step 2. The rules used in this inference are presented in the Appendix.

Example We present a short example of how the first two evaluation steps would operate over the following line from an `strace` log when given the program from Figure 2:

```
11:05:36.464761 open("/etc/ld.so.cache", O_RDONLY) = 3
```

First, the extraction statement on line 10 would produce the following tuple:

```
open('11:05:36', '/etc/ld.so', 'O_RDONLY', '3').
```

Next, the transformation step would produce the following tuples:

```
eventinstance('open', '11:05:36')
data('open', '11:05:36', '1', 'timeType', '11:05:36')
data('open', '11:05:36', '2', 'fileType', '"/etc/ld.so"')
data('open', '11:05:36', '3', 'stringType', 'O_RDONLY')
data('open', '11:05:36', '4', 'handleType', '3')
```

4 Case Study and Results

In this section, we briefly present a case study corresponding to an application of SLog to a realistic intrusion scenario, and discuss the performance of SLog in this case study.

4.1 Intrusion Scenario

To evaluate the performance of SLog in a realistic setting, we performed an intrusion on a Linux system and collected event information corresponding to the intrusion. The intrusion we performed is based on that used for the Honeynet Project Forensic Challenge [12]. Namely, a system running RedHat 6.2 configured as a typical internet server is running a vulnerable version of the `rpc.statd` service. We send an exploit string containing shellcode to launch a remote shell that waits on a predetermined TCP port to this process, connect to the remote shell, and place a “rootkit” in the file `/hacked`. At this point, we kill the remote shell, and restart the system.

4.2 SLog Case Study

The event log sources that we use in our case study were collected using the `strace` utility for Linux systems. `strace` records each instance of a specified set of system calls, as well as their corresponding argument information. We wrote SLog extraction specifications for a number of important system call events, as well as a semantic abstraction that allows us to view these events in terms of files and processes. The semantic abstraction that we selected corresponds to the desired output of the Backtracker tool designed by King *et al.* [8]. Dependencies are calculated between file and process events in an intuitive manner using observed system call invocations. For instance, we say that a process is dependent on another process if one forks the other,

```
depends('process',?pid1,'process',?pid2) :-
    fork(?time3,?pid2,?pid1).
```

Collected Data The data collected by `strace` for this case study corresponds to the operation of our victim machine from the moment the first user logs in, until the system is shut down. As we are only concerned with a subset of the system calls executed in this time period, this corresponds to 2885 system call events executed by 50 processes that may be relevant, as well as their associated argument information. As our goal with this case study is to demonstrate the utility of SLog in a realistic scenario, rather than to demonstrate the full range of its functionality, we wish only to use SLog to replicate the output of a tool of *well known* utility, such as Backtracker. For this reason, we only used information from `strace` logs, rather than from multiple sources, as has been discussed throughout the paper.

Results The final output of SLog as applied to this case study is presented in Figure 4. Five nodes corresponding to configuration files have been removed to allow the image to fit on the page. This analysis was produced by telling SLog to perform a backwards slice on the file named `/hacked`. This is realistic given the scenario, as this file is an object on the system that is clearly and indisputably a symptom of the intrusion, and a likely point of discovery by a system administrator. The graph in Figure 4 shows us that SLog is capable of returning a small subset of the events and objects relevant to an intrusion; the node representing the vulnerable process that allowed the intrusion, `rpc.statd`, is only three hops away from the node representing `/hacked`. Furthermore, we see that the output does not contain a large number of irrelevant information that the user must sort through to obtain an understanding of the intrusion.

Performance The one drawback to the current incarnation of SLog is its performance. Subjectively, we observe that SLog takes far too long to analyze a relatively small amount of data. In this case study, on the 2885 events analyzed by SLog, the extraction and normalization phase of the workflow took 46 minutes running on a standard workstation with an Intel Core 2 Duo running at 1.86 GHz and three gigabytes of memory. Furthermore, constructing the slice presented in Figure 4 took approximately two minutes, and constructing a dependence graph for all of the events in the logs took a surprising 61 minutes. These results represent a failure to meet one of our stated goals, namely to scale to logs in the gigabyte range, and is of primary concern for our future work.

5 Related Work

We begin our discussion of related work with the observation that the dependence tracking facilities presented in

$$\begin{aligned} \mathit{eventT}(\tau_1, \tau_2, \ldots, \tau_n), \mathit{event}(\alpha_1, \alpha_2, \ldots, \alpha_n) \quad \rightarrow \quad & \mathit{eventinstance}(\mathit{event}, \alpha_1), \\ & \mathit{data}(\mathit{event}, \alpha_1, \tau_1, \alpha_1), \\ & \mathit{data}(\mathit{event}, \alpha_1, \tau_2, \alpha_2), \\ & \ldots, \\ & \mathit{data}(\mathit{event}, \alpha_1, \tau_n, \alpha_n) \end{aligned}$$

Figure 3: SLog transformation semantics.

this paper are nothing more than an instance of the more general problem of information flow tracking. Information flows have been used extensively to reason about various system security related topics. Recently, there have been a number of systems that utilize taint tracking to reason about information flows dynamically for the purpose of exploit detection and repair [11]. These tools use information flows proactively in real-time online analyses of specific problems, whereas we are concerned primarily with their utility in comprehensive analysis and overall understanding of events after they occur.

Closer to the type of information flows considered in our work are those used by Kruger *et al.* for the purpose of detecting privacy violations in application-level software [9]. In this work, the authors used dynamic data equivalence with type information to infer dependencies between events, but did not use *def* and *use* information explicitly as in our work to allow fine-grained control over the inference process. They do not consider the problem of integrating multiple sources of event information explicitly as we do. Finally, Kruger *et al.* used dependence information as much for real-time policy enforcement as for post-mortem analysis, which we do not consider in our work.

Perhaps the most relevant previous work to the problem and solution we consider is the work of King *et al* with Backtracker [8]. Here the authors introduce the idea of using dependencies between events for the problem of intrusion analysis, and present a tool for collecting the relevant information off of one type of system, as well as a tool for analyzing the information and producing slice graphs as in our work. However, our work diverges from theirs on several important ways. First, the dependence analysis performed by Backtracker was designed entirely in terms of the event information provided by the authors' custom event logging utility. The tool is not capable of reasoning about dependencies in general, and there is no way for a user to specify additional notions of dependence as in our work. We see this as a major design limitation of Backtracker, and a major part of the inspiration for this work. Additionally, the facilities used by backtracker for event simplification involve lossy filters, as discussed in Section 2.4. We see this not only as a serious flaw from a security perspective, but also as a factor that limits the extensibility and scalability of Backtracker. To summarize, while King *et al.* introduced some of the fundamental concepts used in our own work, we see the work presented in this paper as a more principled approach to a similar problem, built with the issues of extensibility and scalability in mind from the beginning.

6 Conclusions

In this paper, we attempted to address some of the problems related to the process of analyzing, understanding, and recovering from computer system intrusions. We took an *event-oriented* approach, where the system and information relevant to the intrusion are thought of in terms of events on and between objects on the system that are recorded on one or more log files. Furthermore, we defined a notion of *dependence* between events and objects on the system as a way to refine our view of the events and focus on only the information that is explicitly relevant to an intrusion.

As the major contribution of our work, we designed and implemented SLog, a tool based on these principles and observations that allows a user to specify syntactic and semantic information about the events found in an arbitrary set of log sources, and perform *detailed* and *precise* analysis of the events. Furthermore, SLog provides facilities for performing *semantic abstraction* over sequences of events inferred from log sources, allowing the user to simplify his view of the events.

As our evaluation of SLog demonstrates, we were able to fulfill our initial goals of *simplicity* and *extensibility*, but not that of *scalability*. In our experience, the current incarnation of SLog exhibits unacceptable performance costs when run on moderate data sets. Therefore, the main thrust of our future work on SLog will relate to the issue of scalability. When complete, it is our hope that SLog will be able to perform the analysis demonstrated in this paper over log sources that are tens or hundreds of gigabytes in size.

References

[1] National Security Agency. Security-enhanced linux. www.nsa.gov/selinux.

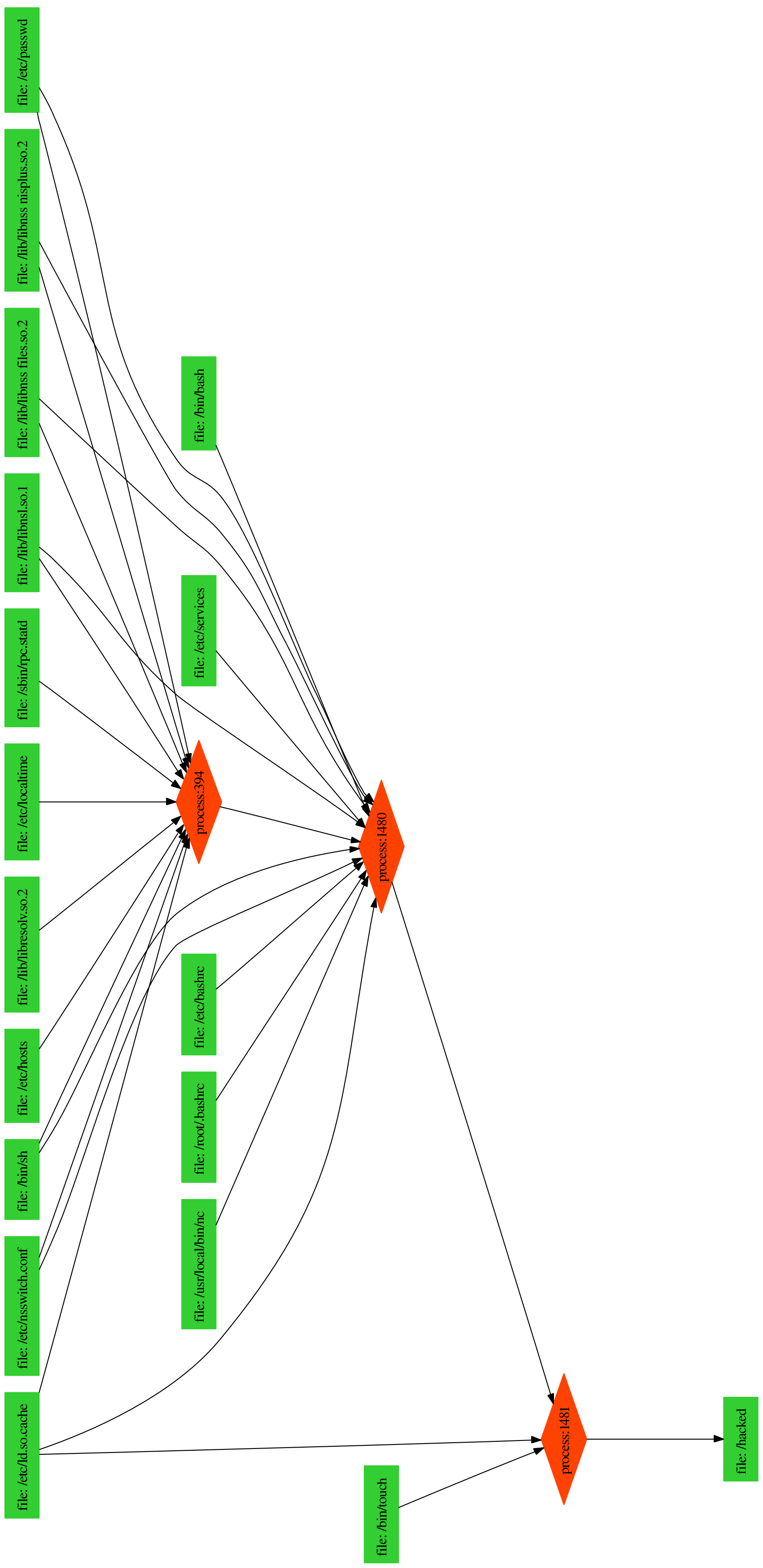

Figure 4: Final output of SLog

[2] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *SIGMOD '86: Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, pages 16–52, New York, NY, USA, 1986. ACM.

[3] Eric A. Brewer. *Combining Systems and Databases: A Search Engine Retrospective*, pages 711–724. MIT Press, 2005.

[4] CERT Coordination Center. Intruder detection checklist. www.cert.org/tech_tips/intruder_detection_checklist.html.

[5] CERT Coordination Center. Overview of attack trends. www.cert.org/archive/pdf/attack_trends.pdf.

[6] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 17–30, New York, NY, USA, 2005. ACM.

[7] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 18–29, New York, NY, USA, 1994. ACM.

[8] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*.

[9] Louis Kruger, Hao Wang, Somesh Jha, Patrick McDaniel, and Wenke Lee. Towards discovering and containing privacy violations in software. Technical report, University of Wisconsin – Madison, 2005.

[10] Andrew C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM.

[11] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[12] The Honeynet Project. Forensic challenge. www.honeynet.org/challenge/index.html.

[13] The Sourcefire Project. The snort lightweight network intrusion detection system. www.snort.org.

[14] Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1033–1044. VLDB Endowment, 2007.

[15] Eduardo F. A. Silva, Flavia A. Barros, and Ricardo B. C. Prudencio. A hybrid machine learning approach for information extraction. In *HIS '06: Proceedings of the Sixth International Conference on Hybrid Intelligent Systems*, page 44, Washington, DC, USA, 2006. IEEE Computer Society.

Appendix

SLog Dependence Inference Rules

```
depends(?event1,?time1,?event2,?time2) :-
        data(?event1,?time1,?a1,?type,?d),
        data(?event2,?time2,?a2,?type,?d),
        def(?event2,?type,?a2),
        use(?event1,?type,?a1),?time1>?time2.
killed(?event1,?time1,?event2,?time2) :-
        data(?event1,?time1,?a1,?type,?d),
        data(?event2,?time2,?a2,?type,?d),
        ?time1>?time,?time>=?time2,
        eventInstance(?event,?time),
        data(?event,?time,?argno,?type,?d),
        kill(?event,?type,?argno).
clear(?event1,?time1,?event2,?time2) :-
        depends(?event1,?time1,?event2,?time2),
        not killed(?event1,?time1,?event2,?time2).
clear(?event1,?time1,?event2,?time2) :-
         clear(?event1,?time1,?event3,?time3),
         clear(?event3,?time3,?event2,?time2).
```