

UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

CROWDSTRIKE, INC.,
Petitioner
v.

GOSECURE, INC.,
Patent Owner

Case No. IPR2025-00067
Patent No. 9,106,697

DECLARATION OF SAMUEL T. KING

I, Samuel T. King, declare as follows:

1. I make this declaration based on my own personal knowledge and, if called upon to testify, would testify competently to the matters contained herein.

2. I am the author of a Ph.D. dissertation titled, “Analyzing Intrusions Using Operating System Level Information Flow” (the “King Dissertation”). I submitted the King Dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Computer Science and Engineering) in the University of Michigan in 2006. I have reviewed Exhibit 1006 in this proceeding, which is a true and accurate copy of the King Dissertation. I have been asked to provide details concerning my research (as described in the King Dissertation), efforts to disseminate this research, and how a person interested in my research would have located the King Dissertation.

3. At a high level, the King Dissertation included a relatively comprehensive description of my research that had been presented and published separately in the years leading to my dissertation defense. Accordingly, individuals who (1) attended my presentations or read my earlier publications and (2) were interested in obtaining more details would have specifically searched for my other publications (e.g., by searching my name directly) and would have obtained a more detailed description of my work in the King Dissertation. In paragraphs, 4-6 below, I

provide some detail about my prior presentations and publications that are related to (and summarized by) the King Dissertation.

4. In the years preceding my dissertation defense in 2006, I presented and published key concepts of my related research. These details ultimately formed the foundation of the King Dissertation. For example, I presented a publication, “*Backtracking Intrusions*,” at the *Proceedings of the 2003 Symposium on Operating Systems Principles (SOSP)* in October 2003. Appendix A is a true and correct copy of this publication. Additionally, I presented a publication, Samuel T. King et al., “*Enriching intrusion alerts through multi-host causality*,” at the *Proceedings of the 2005 Network and Distributed System Security Symposium (NDSS)*, in February 2005. Appendix B is a true and correct copy of this publication. Both publications and my presentations related to these two publications describe aspects of my research that was comprehensively described in the King Dissertation.

5. As I was in the process of finalizing my dissertation, I presented the work described in the King Dissertation in at least a dozen academic symposia across the United States between January-December 2006. One such presentation took place on April 10, 2006 at the Princeton University Computer Science (CS) Colloquium Series. See Appendix C, which includes a true and correct copy of the Wayback Machine archive of the website for the Department of Computer Science of Princeton showing my scheduled April 10, 2006 presentation there. Another

presentation took place on March 9, 2006, at the Cornell University Computer Science (CS) Colloquium Series, Fall 2006. *See id.*, which includes a true and correct copy of the Wayback Machine archive of the website for the Department of Computer Science of Cornell showing my scheduled March 9, 2006 presentation there. Demonstrating the overlap between these presentations and the King Dissertation, both presentations used the King Dissertation's identical title: "Analyzing Intrusions Using Operating System Level Information Flow." *Id.* Further, I discussed the same topics covered in the King Dissertation, including the use of BackTracker, ForwardTracker, and Bi-directional Distributed BackTracker (BDB) systems to create information flow graphs, as evidenced by the description of the presentations. *Id.*

6. My thesis advisor, Dr. Peter Chen, also maintained a publicly accessible University of Michigan website documenting research he supervised. Appendix D is a true and correct copy of a Wayback Machine archive that shows many of my papers related to the research summarized in the King Dissertation were available on Dr. Chen's website as of February 2007. Among those relevant papers, both papers referenced above in ¶4, and the ACM Journal version of my SOSP 2003 paper were linked on Dr. Chen's website as of February 2007. Additionally, the ReVirt publication in 2002, the USENIX Technical Conference publication from 2003, the two other publications listed from 2005 that I co-authored, and the SubVirt

publication in 2006 are related to the work in the King Dissertation. As indicated in Appendix D, Dr. Chen's website included a link to the King Dissertation, describing the document as "Sam King's Ph.D. thesis." *Id.*

7. Further demonstrating that the King Dissertation was indeed available to individuals interested in this research area, it was cited by at least one academic article prior to 2010. Specifically, the King Dissertation was cited by the article, "*BorderPatrol: Isolating Events for Black-box Tracing*," published on April 1, 2008, in The Association for Computing Machinery (ACM) SIGOPS Operating Systems Review (<https://doi.org/10.1145/1357010.1352613>). Appendix E is a true and correct copy of this publication.

8. I have also been asked (1) to assume that interested individuals could have searched library records by thesis title and (2) to consider whether, even without knowledge of my prior work, the King Dissertation title is adequately descriptive that an individual interested in this research area would have located it through keyword searching. It is my opinion that a person interested in this subject matter could indeed have searched for and discovered the King Dissertation as of 2006. An individual interested in the field of computer intrusion detection and analysis would have been able to locate the King Dissertation through a reasonable search given the keywords included in my dissertation's title, such as "computer intrusions" and/or "operating system level information flow."

9. I have been retained on behalf of the Petitioner to provide assistance in the above-captioned proceedings related to the public availability of the King Dissertation. I am being compensated for my services in this matter at my normal consulting rate plus reasonable expenses. My statements are objective, and my compensation does not depend on the outcome of this matter.

10. I declare under penalty of perjury that the foregoing is true and correct. I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code.

Dated: 03/18/2025

A handwritten signature in black ink, appearing to be 'SK' with a stylized flourish.

Samuel T. King

APPENDIX A

Backtracking Intrusions

Samuel T. King
kingst@umich.edu

Peter M. Chen
pmchen@umich.edu

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122

ABSTRACT

Analyzing intrusions today is an arduous, largely manual task because system administrators lack the information and tools needed to understand easily the sequence of steps that occurred in an attack. The goal of BackTracker is to identify automatically potential sequences of steps that occurred in an intrusion. Starting with a single detection point (e.g., a suspicious file), BackTracker identifies files and processes that could have affected that detection point and displays chains of events in a dependency graph. We use BackTracker to analyze several real attacks against computers that we set up as honeypots. In each case, BackTracker is able to highlight effectively the entry point used to gain access to the system and the sequence of steps from that entry point to the point at which we noticed the intrusion. The logging required to support BackTracker added 9% overhead in running time and generated 1.2 GB per day of log data for an operating-system intensive workload.

Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection—information flow controls, invasive software (e.g., viruses, worms, Trojan horses); K.6.4 [Management of Computing and Information Systems]: System Management—management audit; K.6.5 [Management of Computing and Information Systems]: Security and Protection—invasive software, unauthorized access (e.g., hacking, phreaking).

General Terms

Management, Security.

Keywords

Computer forensics, intrusion analysis, information flow.

1. INTRODUCTION

The frequency of computer intrusions has been increasing rapidly for several years [4]. It seems likely that, for the foreseeable future, even the most diligent system administrators will continue to cope routinely with computer break-ins. After discovering an intrusion,

a diligent system administrator should do several things to recover from the intrusion. First, the administrator should understand how the intruder gained access to the system. Second, the administrator should identify the damage inflicted on the system (e.g., modified files, leaked secrets, installed backdoors). Third, the administrator should fix the vulnerability that allowed the intrusion and try to undo the damage wrought by the intruder. This paper addresses the methods and tools an administrator uses to understand how an intruder gained access to the system.

Before an administrator can start to understand an intrusion, she must first detect that an intrusion has occurred [2]. There are numerous ways to detect a compromise. A tool such as TripWire [20] can detect a modified system file; a network or host firewall can notice a process conducting a port scan or launching a denial-of-service attack; a sandboxing tool can notice a program making disallowed or unusual patterns of system calls [18, 16] or executing foreign code [22]. We use the term *detection point* to refer to the state on the local computer system that alerts the administrator to the intrusion. For example, a detection point could be a deleted, modified, or additional file, or it could be a process that is behaving in an unusual or suspicious manner.

Once an administrator is aware that a computer is compromised, the next step is to investigate how the compromise took place [1]. Administrators typically use two main sources of information to find clues about an intrusion: system/network logs and disk state [15]. An administrator might find log entries that show unexpected output from vulnerable applications, deleted or forgotten attack toolkits on disk, or file modification dates which hint at the sequence of events during the intrusion. Many tools exist that make this job easier. For example, Snort can log network traffic; Ethereal can present application-level views of that network traffic; and The Coroner's Toolkit can recover deleted files [14] or summarize the times at which files were last modified, accessed, or created [13] (similar tools are Guidance Software's EnCase, Access Data's Forensic Toolkit, Internal Revenue Services' ILook, and ASR Data's SMART).

Unfortunately, current sources of information suffer from one or more limitations. Host logs typically show only partial, application-specific information about what happened, such as HTTP connections or login attempts, and they often show little about what occurred on the system after the initial compromise. Network logs may contain encrypted data, and the administrator may not be able to recover the decryption key. The attacker may also use an obfuscated custom command set to communicate with a backdoor, and the administrator may not be able to recover the backdoor program to help understand the commands. Disk images may contain useful information about the final state, but they do not provide a complete history of what transpired during the attack. A general limitation of most tools and sources of information is that they

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOSP'03, October 19-22, 2003, Bolton Landing, New York, USA.
Copyright 2003 ACM 1-58113-757-5/03/0010...\$5.00.

intermingle the actions of the intruder (or the state caused by those actions) with the actions/state of legitimate users. Even in cases where the logs and disk state contain enough information to understand an attack, identifying the sequence of events from the initial compromise to the point of detection point is still largely a manual process.

This paper describes a tool called BackTracker that attempts to address the shortcomings in current tools and sources of information and thereby help an administrator more easily understand what took place during an attack. Working backward from a detection point, BackTracker identifies chains of events that could have led to the modification that was detected. An administrator can then focus her detective work on those chains of events, leading to a quicker and easier identification of the vulnerability. In order to identify these chains of events, BackTracker logs the system calls that induce most directly dependencies between operating system objects (e.g., creating a process, reading and writing files). BackTracker's goal is to provide helpful information for most attacks; it does not provide complete information for every possible attack.

We have implemented BackTracker for Linux in two components: an on-line component that logs events and an off-line component that graphs events related to the attack. BackTracker currently tracks many (but not all) relevant OS events. We found that these events can be logged and analyzed with moderate time and space overhead and that the output generated by BackTracker was helpful in understanding several real attacks against computers we set up as honeypots.

2. DESIGN OF BACKTRACKER

BackTracker's goal is to reconstruct a time-line of events that occur in an attack. Figure 1 illustrates this with BackTracker's results for an intrusion on our honeypot machine that occurred on March 12, 2003. The graph shows that the attacker caused the Apache web server (httpd) to create a command shell (bash), downloaded and unpacked an executable (/tmp/xploit/ptrace), then ran the executable using a different group identity (we believe the executable was seeking to exploit a race condition in the Linux ptrace code to gain root access). We detected the intrusion by seeing the ptrace process in the process listing.

There are many levels at which events and objects can be observed. Application-level logs such as Apache's log of HTTP requests are semantically rich. However, they provide no information about the attacker's own programs, and they can be disabled by an attacker who gains privileged access. Network-level logs provide more information for remote attacks, but they can be rendered useless by encryption or obfuscation. Logging low-level events such as machine instructions can provide complete information about the computer's execution [12], but these can be difficult for administrators to understand quickly.

BackTracker works by observing OS-level objects (e.g., files, filenames, processes) and events (e.g., system calls). This level is a compromise between the application level (semantically rich but easily disabled) and the machine level (difficult to disable but semantically poor). Unlike application-level logging, OS-level logging cannot separate objects within an application (e.g., user-level threads), but rather considers the application as a whole. While OS-level semantics can be disrupted by attacking the kernel, gaining kernel-mode control can be made considerably more difficult than gaining privileged user-mode control [19]. Unlike network-level logging, OS-level events can be interpreted even if the attacker encrypts or obfuscates his network communication.

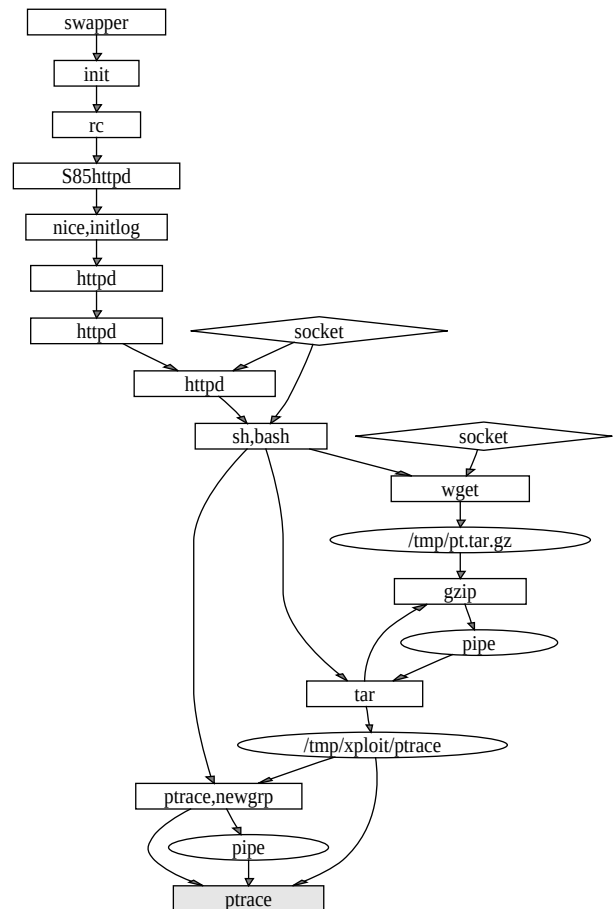


Figure 1: Filtered dependency graph for ptrace attack. Processes are shown as boxes (labeled by program names called by execve during that process's lifetime); files are shown as ovals; sockets are shown as diamonds. BackTracker can also show process IDs, file inode numbers, and socket ports. The detection point is shaded.

This section's description of BackTracker is divided into three parts (increasing in degree of aggregation): objects, events that cause dependencies between objects, and dependency graphs. The description and implementation of BackTracker is given for Unix-like operating systems.

2.1 Objects

Three types of OS-level objects are relevant to BackTracker's analysis: processes, files, and filenames.

A process is identified uniquely by a process ID and a version number. BackTracker keeps track of a process from the time it is created by a fork or clone system call to the point where it exits. The one process that is not created by fork or clone is the first process (swapper); BackTracker starts keeping track of swapper when it makes its first system call.

A file object includes any data or metadata that is specific to that file, such as its contents, owner, or modification time. A file is identified uniquely by a device, an inode number, and a version number. Because files are identified by inode number rather than by name, BackTracker tracks a file across rename operations. BackTracker treats pipes and named pipes as normal files. Objects associated with System V IPC (messages, shared memory, sema-

phores) can also be treated as files, though the current BackTracker implementation does not yet handle these.

A filename object refers to the directory data that maps a name to a file object. A filename object is identified uniquely by a canonical name, which is an absolute pathname with all `./` and `../` links resolved. Note the difference between file and filename objects. In Unix, a single file can appear in multiple places in the filesystem directory structure, so writing a file via one name will affect the data returned when reading the file via the different name. File objects are affected by system calls such as `write`, whereas filename objects are affected by system calls such as `rename`, `create`, and `unlink`.

It is possible to keep track of objects at a different granularity than processes, files, and filenames. One could keep track of finer-grained objects, such as file blocks, or coarser-grained objects, such as all files within a directory. Keeping track of objects on a finer granularity reduces false dependencies (similar to false sharing in distributed shared memory systems), but is harder and may induce higher overhead.

2.2 Potential Dependency-Causing Events

BackTracker logs events at runtime that induce dependency relationships between objects, i.e. events in which one object affects the state of another object. These events are the links that allow BackTracker to deduce timelines of events leading to a detection point. A dependency relationship is specified by three parts: a source object, a sink object, and a time interval. For example, the reading of a file by a process causes that process (the sink object) to depend on that file (the source object). We denote a dependency from a source object to a sink object as *source->sink*.

We use time intervals to reduce false dependencies. For example, a process that reads a file at time 10 does not depend on writes to the file that occur after time 10. Time is measured in terms of an increasing event counter. Unless otherwise stated, the interval for an event starts when the system call is invoked and ends when the system call returns. A few types of events (such as shared memory accesses) are aggregated into a single event over a longer interval because it is difficult to identify the times of individual events.

There are numerous events which cause objects to affect each other. This section describes potential events that BackTracker could track. Section 2.3 describes how BackTracker uses dependency-causing events. Section 2.4 then describes why some events are more important to track than others and identifies the subset of these dependencies logged by the current BackTracker prototype. We classify dependency-causing events based on the source and sink objects for the dependency they induce: process/process, process/file, and process/filename.

2.2.1 Process/Process Dependencies

The first category of events are those for which one process directly affects the execution of another process. One process can affect another directly by creating it, sharing memory with it, or signaling it. For example, an intruder may login to the system through `sshd`, then fork a shell process, then fork a process that performs a denial-of-service attack. Processes can also affect each other indirectly (e.g., by writing and reading files), and we describe these types of dependencies in the next two sections.

If a process creates another process, there is a parent->child dependency because the parent initiated the existence of the child and because the child's address space is initialized with data from the parent's address space.

Besides the traditional `fork` system call, Linux supports the `clone` system call, which creates a child process that shares the parent's address space (these are essentially kernel threads). Children that are created via `clone` have an additional bi-directional parent<->child dependency with their parent due to their shared address space. In addition, `clone` creates a bi-directional dependency between the child and other processes that are currently sharing the parent's address space. Because it is difficult to track individual loads and stores to shared memory locations, we group all loads and stores to shared memory into a single event that causes the two processes to depend on each other over a longer time interval. We do this grouping by assuming conservatively that the time interval of the shared-memory dependency lasts from the time the child is created to the time either process exits or replaces its address space through the `execve` system call.

2.2.2 Process/File Dependencies

The second category of events are those for which a process affects or is affected by data or attributes associated with a file. For example, an intruder can edit the password file (process->file dependency), then log in using the new password file (file->process dependency). Receiving data from a network socket can also be treated as reading a file, although the sending and receiving computers would need to cooperate to link the receive event with the corresponding send event.

System calls like `write` and `writv` cause a process->file dependency. System calls like `read`, `readv`, and `execve` cause a file->process dependency.

Files can also be mapped into a process's address space through `mmap`, then accessed via load/store instructions. As with shared memory between processes, we aggregate mapped-file accesses into a single event, lasting from the time the file is `mmap`'ed to the time the process exits. This conservative time interval allows BackTracker to not track individual memory operations or the un-mapping or re-mapping of files. The direction of the dependency for mapped files depends on the access permissions used when opening the file: mapping a file read-only causes a file->process dependency; mapping a file write-only causes a process->file dependency; mapping a file read/write causes a bi-directional process<->file dependency. When a process is created, it inherits a dependency with each file mapped into its parent's address space.

A process can also affect or be affected by a file's attributes, such as the file's owner, permissions, and modification time. System calls that modify a file's attributes (e.g., `chown`, `chmod`, `utime`) cause a process->file dependency. System calls that read file attributes (e.g., `fstat`) cause a file->process dependency. In fact, any system call that specifies a file (e.g., `open`, `chdir`, `unlink`, `execve`) causes a file->process dependency if the filename specified in the call exists, because the return value of that system call depends on the file's owner and permissions.

2.2.3 Process/Filename Dependencies

The third category of events are those that cause a process to affect or be affected by a filename object. For example, an intruder can delete a configuration file and cause an application to use an insecure default configuration. Or an intruder can swap the names of current and backup password files to cause the system to use out-of-date passwords.

Any system call that includes a filename argument (e.g., `open`, `creat`, `link`, `unlink`, `mkdir`, `rename`, `rmdir`, `stat`, `chmod`) causes a filename->process dependency, because the return value of the system call depends on the existence of that filename in the file system directory tree. In addition, the process is affected by all parent

```

foreach event E in log { /* read events from latest to earliest */
  foreach object O in graph {
    if (E affects O by the time threshold for object O) {
      if (E's source object not already in graph) {
        add E's source object to graph
        set time threshold for E's source object to time of E
      }
      add edge from E's source object to E's sink object
    }
  }
}

```

Figure 2: Constructing a dependency graph. This code shows the basic algorithm used to construct a dependency graph from a log of dependency-causing events with discrete times.

directories of the filename (e.g., opening the file /a/b/c depends on the existence of /a and /a/b). A system call that reads a directory causes a filename->process dependency for all filenames in that directory.

System calls that modify a filename argument cause a process->filename dependency if they succeed. Examples are creat, link, unlink, rename, mkdir, rmdir, and mount.

2.3 Dependency Graphs

By logging objects and dependency-causing events during run-time, BackTracker saves enough information to build a graph that depicts the dependency relationships between all objects seen over that execution. Rather than presenting the complete dependency graph, however, we would like to make understanding an attack as easy as possible by presenting only the relevant portion of the graph. This section describes how to select the objects and events in the graph that relate to the attack.

We assume that the administrator has noticed the compromised system and can identify at least one *detection point*, such as a modified, extra, or deleted file, or a suspicious or missing process. Starting from that detection point, our goal is to build a dependency graph of all objects and events that causally affect the state of the detection point [23]. The part of the BackTracker system that builds this dependency graph is called GraphGen. GraphGen is run off-line, i.e. after the attack.

To construct the dependency graph, GraphGen reads the log of events, starting from the last event and reading toward the beginning of the log (Figure 2). For each event, GraphGen evaluates whether that event can affect any object that is currently in the dependency graph. Each object in the evolving graph has a time threshold associated with it, which is the maximum time that an event can occur and be considered relevant for that object. GraphGen is initialized with the object associated with the detection point, and the time threshold associated with this object is the earliest time at which the administrator knows the object's state is compromised. Because the log is processed in reverse time order, all events encountered in the log after the detection point will occur before the time threshold of all objects currently in the graph.

Consider how this algorithm works for the set of events shown in Figure 3a (Figure 3b pictures the log of events as a complete dependency graph):

- GraphGen is initialized with the detection point, which is file X at time 10. That is, the administrator knows that file X has the wrong contents by time 10.

- GraphGen considers the event at time 8. This event does not affect any object in the current graph (i.e. file X), so we ignore it.
- GraphGen considers the event at time 7. This event also does not affect any object in the current graph.
- GraphGen considers the event at time 6. This event affects file X in time to affect its contents at the detection point, so GraphGen adds process C to the dependency graph with an edge from process C to file X. GraphGen associates time 6 to process C, because only events that occur before time 6 can affect C in time to affect the detection point.
- GraphGen considers the event at time 5. This event affects an object in the dependency graph (process C) in time, so GraphGen adds file 1 to the graph with an edge to process C (at time 5).
- GraphGen considers the event at time 4. This event affects an object in the dependency graph (process C) in time, so GraphGen adds process A to the dependency graph with an edge to process C (at time 4).
- GraphGen considers the event at time 3. This event affects process A in time, so we add file 0 to the graph with an edge to process A (at time 3).
- GraphGen considers the event at time 2. This event does not affect any object in the current graph.
- GraphGen considers the event at time 1. This event affects file 1 in time, so we add process B to the graph with an edge to file 1 (at time 1).
- GraphGen considers the event at time 0. This event affects process B in time, so we add an edge from process A to process B (process A is already in the graph).

The resulting dependency graph (Figure 3c) is a subset of the graph in Figure 3b. We believe this type of graph to be a useful picture of the events that lead to the detection point, especially if it can reduce dramatically the number of objects and events an administrator must examine to understand an attack.

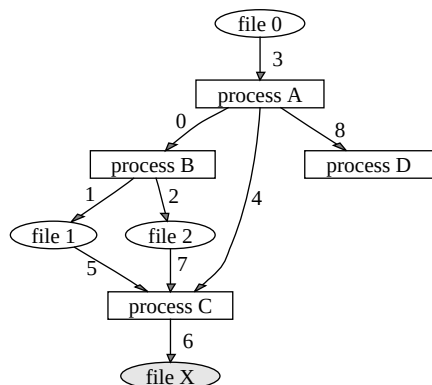
The full algorithm is a bit more complicated because it must handle events that span an interval of time, rather than events with discrete times. Consider a scenario where the dependency graph currently has an object O with time threshold t. If an event $P \rightarrow O$ occurs during time interval $[x-y]$, then we should add P to the dependency graph iff $x < t$, i.e. this event started to affect O by O's time threshold. If P is added to the dependency graph, the time threshold associated with P would be $\text{minimum}(t, y)$, because the event would have no relevant effect on O after time t, and the event itself stopped after time y.

Events with intervals are added to the log in order of the *later* time in their interval. This order guarantees that GraphGen sees the event and can add the source object for that event as soon as possible (so that the added source object can in turn be affected by events processed subsequently by GraphGen).

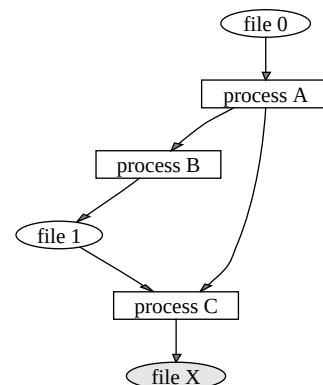
For example, consider how GraphGen would handle an event process B->file 1 in Figure 3b with a time interval of 1-7. GraphGen would encounter this event at a log time 7 because events are ordered by the later time in their interval. At this time, file 1 is not yet in the dependency graph. GraphGen remembers this event and continually re-evaluates whether it affects new objects as they are added to the dependency graph. When file 1 is added to the graph (log time 5), GraphGen sees that the event process B->file 1 affects file 1 and adds process B to the graph. The time threshold for process B would be time 5 (the lesser of time 5 and time 7).

time 0: process A creates process B
time 1: process B writes file 1
time 2: process B writes file 2
time 3: process A reads file 0
time 4: process A creates process C
time 5: process C reads file 1
time 6: process C writes file X
time 7: process C reads file 2
time 8: process A creates process D

(a) event log



(b) dependency graph
for complete event log



(c) dependency graph
generated by GraphGen

Figure 3: Dependency graph for an example set of events with discrete times. The label on each edge shows the time of the event. The detection point is file X at time 10. By processing the event log, GraphGen prunes away events and objects that do not affect file X by time 10.

GraphGen maintains several data structures to accelerate its processing of events. Its main data structure is a hash table of all objects currently in the dependency graph, called GraphObjects. GraphGen uses GraphObjects to determine quickly if the event under consideration affects an object that is already in the graph. GraphGen also remembers those events with time intervals that include the current time being processed in the log. GraphGen stores these events in an ObjectsIntervals hash table, hashed on the sink object for that event. When GraphGen adds an object to GraphObjects, it checks if any events in the ObjectsIntervals hash table affect the new object before the time threshold for the new object. Finally, GraphGen maintains a priority queue of events with intervals that include the current time (prioritized by the starting time of the event). The priority queue allows GraphGen to find and discard events quickly whose intervals no longer include the current time.

2.4 Dependencies Tracked By Current Prototype

Section 2.2 lists numerous ways in which one object can potentially affect another. It is important to note, however, that *affecting* an object is not the same as *controlling* an object. Dependency-causing events vary widely in terms of how much the source object can control the sink object. Our current implementation of BackTracker focuses on tracking the events we consider easiest for an attacker to use to accomplish a task; we call these events *high-control events*.

Some examples of high-control events are changing the contents of a file or creating a child process. It is relatively easy for an intruder to perform a task by using high-control events. For example, an intruder can install a backdoor easily by modifying an executable file, then creating a process that executes it.

Some examples of low-control events are changing a file's access time or creating a filename in a directory. Although these events can affect the execution of other processes, they tend to generate a high degree of noise in the dependency graph. For example, if BackTracker tracks the dependency caused by reading a directory, then a process that lists the files in /tmp would depend on all processes that have ever created, renamed, or deleted filenames in

/tmp. Timing channels [24] are an example of an extremely low-control event; e.g., an attacker may be able to trigger a race condition by executing a CPU-intensive program.

Fortunately, BackTracker is able to provide useful analysis without tracking low-control events, even if low-control events are used in the attack. This is because it is difficult for an intruder to perform a task solely by using low-control events. Consider an intruder who wants to use low-control events to accomplish an arbitrary task; for example, he may try to cause a program to install a backdoor when it sees a new filename appear in /tmp.

Using an existing program to carry out this task is difficult because existing programs do not generally perform arbitrary tasks when they see incidental changes such as a new filename in /tmp. If an attacker can cause an existing program to perform an arbitrary task by making such an incidental change, it generally means that the program has a bug (e.g., buffer overflow or race condition). Even if BackTracker does not track this event, it will still be able to highlight the buggy existing program by tracking the chain of events from the detection point back to that program.

Using a new, custom program to carry out an arbitrary task is easy. However, it will not evade BackTracker's analysis because the events of writing and executing such a custom program are high-control events and BackTracker will link the backdoor to the intruder's earlier actions through those high-control events. To illustrate this, consider in Figure 3b if the event "file 1->process C" was a low-control event, and process C was created by process B (rather than by process A as shown). Even if BackTracker did not track the event "file 1->process C", it would still link process B to the detection point via the event "process B->process C".

BackTracker currently logs and analyzes the following high-control events: process creation through fork or clone; load and store to shared memory; read and write of files and pipes; receiving data from a socket; execve of files; load and store to mmap'ed files; and opening a file. We have implemented partially the logging and tracking of file attributes and filename create, delete, and rename (these events are not reflected in Section 5's results). We plan to implement logging and tracking for System V IPC (messages, shared memory, semaphores) and signals.

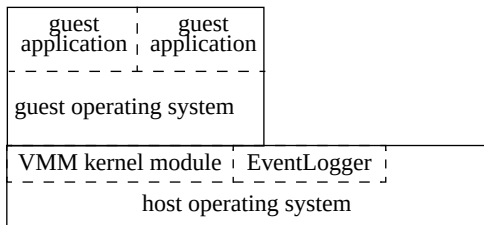


Figure 4: Implementation structure. Our current prototype runs the target operating system (Linux 2.4.18) and applications in a virtual machine contained within a host process. The virtual-machine monitor (VMM) kernel module calls a kernel procedure (EventLogger) when a guest application invokes or returns from a system call or when a guest application process exits. EventLogger then reads information about the event from the virtual machine’s physical memory.

3. IMPLEMENTATION STRUCTURE FOR LOGGING EVENTS AND OBJECTS

While the computer is executing, BackTracker must log information about objects and dependency-causing events to enable the dependency-graph analysis described in Section 2. The part of BackTracker that logs this information is called EventLogger. After the intrusion, an administrator can run GraphGen off-line on a log (or concatenation of logs spanning several reboots) generated by EventLogger. GraphGen produces a graph in a format suitable for input to the dot program (part of AT&T’s Graph Visualization Project), which generates the human-readable graphs used in this paper.

There are several ways to implement EventLogger. The strategy for our current BackTracker prototype is to run the target operating system (Linux 2.4.18) and applications inside a virtual machine and to have the virtual-machine monitor call a kernel procedure (EventLogger) at appropriate times (Figure 4). The operating system running inside the virtual machine is called the *guest operating system* to distinguish it from the operating system that the virtual machine is running on, which is called the *host operating system*. Guest processes run on the guest operating system inside the virtual machines; host processes run on the host operating system. The entire virtual machine is encapsulated in a host process. The log written by EventLogger is stored as a host file (compressed with gzip). The virtual-machine monitor prevents intruders in the guest from interfering with EventLogger or its log file.

EventLogger gleans information about events and objects inside the target system by examining the state of the virtual machine. The virtual-machine monitor notifies EventLogger whenever a guest application invokes or returns from a system call or when a guest application process exits. EventLogger learns about the event from data passed by the virtual-machine monitor and from the virtual machine’s physical memory (which is a host file). EventLogger is compiled with headers from the guest kernel and reads guest kernel data structures from the guest’s physical memory to determine event information (e.g., system call parameters), object identities (e.g., file inode numbers, filenames, process identifiers) and dependency information (e.g., it reads the address map of a guest process to learn what mmap’ed files it inherited from its parent). The code for EventLogger is approximately 1300 lines, and we added 40 lines of code to the virtual-machine monitor to support EventLogger. We made no changes to the guest operating system.

Another possible strategy is to add EventLogger to the target operating system and not use a virtual machine. To protect EventLog-

ger’s log from the intruder, one could store the log on a remote computer or in a protected file on the local computer.

The results of BackTracker’s analysis are independent of where EventLogger is implemented. We have ported EventLogger to a standalone operating system (Linux 2.4.18) to give our local system administrators the option of using BackTracker without using a virtual machine. To port EventLogger to the target operating system, we modified the code that gleans information about events and objects; this porting took one day.

The main reason we use a virtual-machine-based structure is to leverage ReVirt, which enables one to replay the complete, instruction-by-instruction execution of a virtual machine [12]. This ability to replay executions at arbitrarily fine detail allows us to capture complete information about workloads (e.g., real intrusions) while still making changes to EventLogger. Without the ability to replay a workload repeatedly, we would only be able to analyze information captured by the version of EventLogger that was running at the time of that workload. This ability is especially important for analyzing real attacks, since real attackers do not re-issue their workload upon request. EventLogger can log events and objects during the original run or during a replaying run.

One of the standard reasons for using a virtual machine—not trusting the target operating system—does not hold for BackTracker. If an attacker gains control of the guest operating system, she can carry out arbitrary tasks inside the guest without being tracked by BackTracker (in contrast, ReVirt works even if the attacker gains control of the guest operating system).

We use a version of the UMLinux virtual machine [8] that uses a host kernel (based on Linux 2.4.18) that is optimized to support virtual machines [21]. The virtualization overhead of the optimized UMLinux is comparable to that of VMWare Workstation 3.1. CPU-intensive applications experience almost no overhead, and kernel-intensive applications such as SPECweb99 and compiling the Linux kernel experience 14-35% overhead [21].

4. PRIORITIZING PARTS OF A DEPENDENCY GRAPH

Dependency graphs for a busy system may be too large to scrutinize each object and event. Fortunately, not all objects and events warrant the same amount of scrutiny when a system administrator analyzes an intrusion. This section describes several ways to prioritize or filter a dependency graph in order to highlight those parts that are mostly likely to be helpful in understanding an intrusion. Of course, there is a tradeoff inherent to any filtering. Even objects or events that are unlikely to be important in understanding an intrusion may nevertheless be relevant, and filtering these out may accidentally hide important sequences of events.

One way to prioritize important parts of a graph is to ignore certain objects. For example, the login program reads and writes the file `/var/run/utmp`. These events cause a new login session to depend on all prior login sessions. Another example is the file `/etc/mtab`. This file is written by mount and umount and is read by bash at startup, causing all events to depend on mount and umount. A final example is that the bash shell commonly writes to a file named `.bash_history` when it exits. Shell invocations start by reading `.bash_history`, so all actions by all shells depend on all prior executions of bash. While these are true dependencies, it is easier to start analyzing the intrusion without these objects cluttering the graph, then to add these objects if needed.

A second way to prioritize important parts of a graph is to filter out certain types of events. For example, one could filter out some low-control events.

	<i>bind</i> (Fig 5-6)	<i>ptrace</i> (Fig 1)	<i>openssl-too</i> (Fig 7)	<i>self</i> (Fig 8)
time period being analyzed	24 hours		61 hours	24 hours
# of objects and events in log	155,344 objects 1,204,166 events		77,334 objects 382,955 events	2,187,963 objects 55,894,869 events
# of objects and events in unfiltered dependency graph	5,281 objects 9,825 events	552 objects 2,635 events	495 objects 2,414 events	717 objects 3,387 events
# of objects and events in filtered dependency graph	24 objects 28 events	20 objects 25 events	28 objects 41 events	56 (36) objects 81 (49) events
growth rate of EventLogger's log	0.017 GB/day		0.002 GB/day	1.2 GB/day
time overhead of EventLogger	0%		0%	9%

Table 1: Statistics for BackTracker's analysis of attacks. This table shows results for three real attacks and one simulated attack. Event counts include only the first event from a source object to a sink object. GraphGen and the filtering rules drastically reduce the amount of information that an administrator must peruse to understand an attack. Results related to EventLogger's log are combined for the *bind* and *ptrace* attacks because these attacks are intermingled in one log. Object and events counts for the *self* attack are given for two different levels of filtering.

These first two types of filtering (objects and events) may filter out a vital link in the intrusion and thereby disconnect the detection point from the source of the intrusion. Hence they should be used only for cases where they reduce noise drastically with only a small risk of filtering out vital links. The remainder of the filtering rules do not run the risk of breaking a vital link in the middle of an attack sequence.

A third way to simplify the graph is to hide files that have been read but not written in the time period being analyzed (read-only files). For example, in Figure 3c, file 0 is read by process A but is not written during the period being analyzed. These files are often default configuration or header files. Not showing these files in the graph does not generally hinder one's ability to understand an attack because the attacker did not modify these files in the time period being considered and because the processes that read the files are still included in the dependency graph. If the initial analysis does not reveal enough about the attack, an administrator may need to extend the analysis further back in the log to include events that modified files which were previously considered read-only. Filtering out read-only files cannot break a link in any attack sequence contained in the log being analyzed, because there are no events in that log that affect these files.

A fourth way to prioritize important parts of a graph is to filter out helper processes that take input from one process, perform a simple function on that input, then return data to the main process. For example, the system-wide bash startup script (/etc/bashrc) causes bash to invoke the id program to learn the name and group of the user, and the system startup scripts on Linux invoke the program consoletype to learn the type of the console that is being used. These usage patterns are recognized easily in a graph: they form a cycle in the graph (usually connected by a pipe) and take input only from the parent process and from read-only files. As with the prior filtering rule, this rule cannot disconnect a detection point from an intrusion source that precedes the cycle, because these cycles take input only from the main process, and the main process is left in the dependency graph.

A fifth way to prioritize important parts of a graph is to choose several detection points, then take the intersection of the dependency graphs formed from those dependency points. The intersection of the graphs is likely to highlight the earlier portion of an attack (which affect all detection points), and these portions are important to understanding how the attacker initially gained control in the system.

We implement these filtering rules as options in GraphGen. GraphGen includes a set of default rules which work well for all attacks we have experienced. A user can add to a configuration file regular expressions that specify additional objects and events to filter. We considered filtering the graph after GraphGen produced it, but this would leave in objects that should have been pruned (such as an object that was connected only via an object that was filtered out).

Other graph visualization techniques can help an administrator understand large dependency graphs. For example, a post-processing tool can aggregate related objects in the graph, such as all files in a directory, or show how the graph grows as the run progresses.

We expect an administrator to run GraphGen several times with different filtering rules and log periods. She might first analyze a short log that she hopes includes the entire attack. She might also filter out many objects and events to try to highlight the most important parts of an intrusion without much noise from irrelevant events. If this initial analysis does not reveal enough about the attack, she can extend the analysis period further back in the log and use fewer filtering rules.

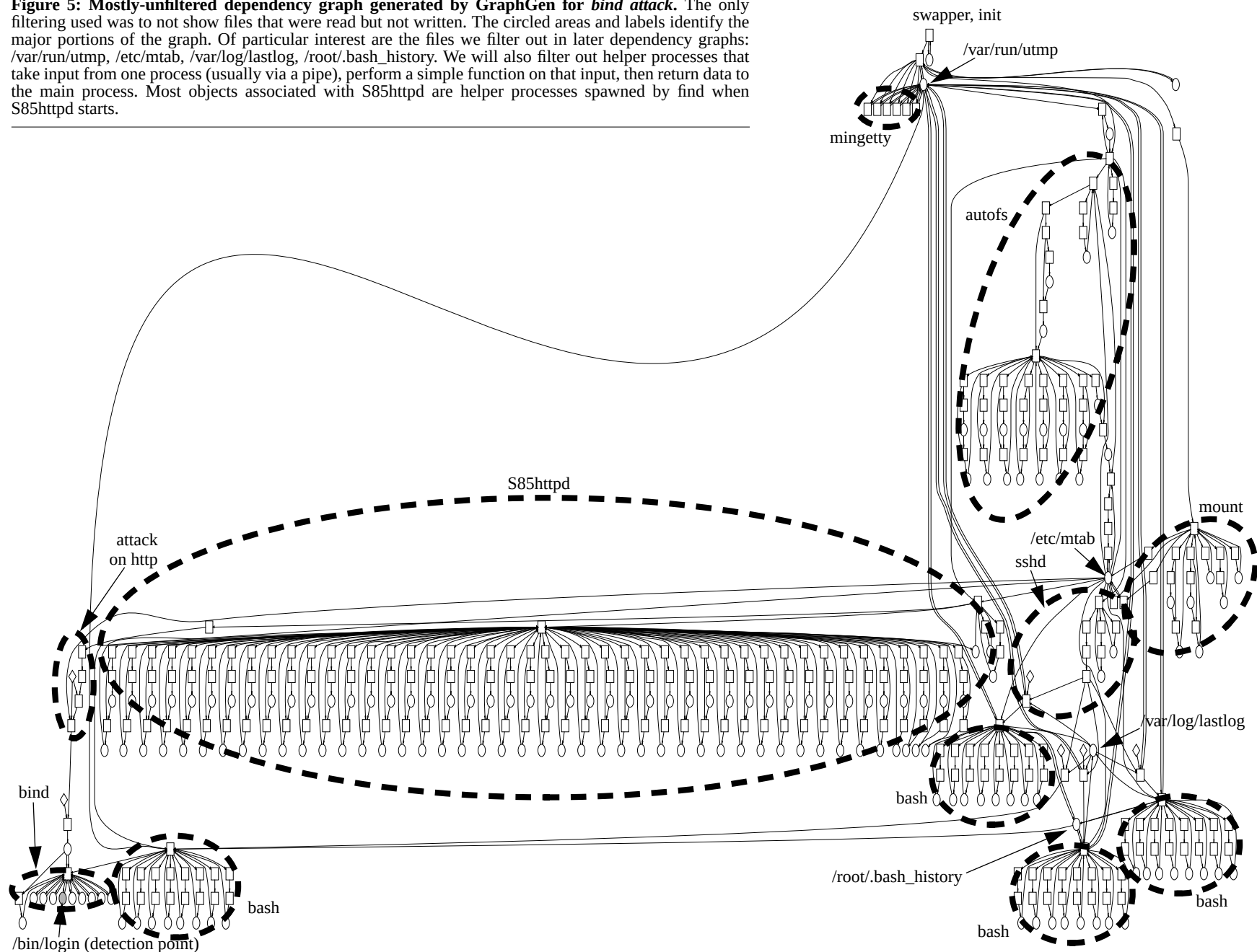
5. EVALUATION

This section evaluates how well BackTracker works on three real attacks and one simulated attack (Table 1).

To experience and analyze real attacks, we set up a honeypot machine [9, 25] and installed the default configuration of RedHat 7.0. This configuration is vulnerable to several remote and local attacks, although the virtual machine disrupts some attacks by shrinking the virtual address space of guest applications. Our honeypot configuration is vulnerable to (at least) two attacks. A remote user can exploit the OpenSSL library used in the Apache web server (httpd) to attain a non-root shell [5], and a local user can exploit sendmail to attain a root shell [3]. After an attacker compromises the system, they have more-or-less free reign on the honeypot—the y can read files, download, compile, and execute programs, scan other machines, etc.

We ran a variety of tools to detect intruders. We used a home-grown imitation of TripWire [20] to detect changes to important system files. We used Ethereal and Snort to detect suspicious amounts of incoming or outgoing network traffic. We also perused the system manually to look for any unexpected files or processes.

Figure 5: Mostly-unfiltered dependency graph generated by GraphGen for *bind* attack. The only filtering used was to not show files that were read but not written. The circled areas and labels identify the major portions of the graph. Of particular interest are the files we filter out in later dependency graphs: */var/run/utmp*, */etc/mtab*, */var/log/lastlog*, */root/.bash_history*. We will also filter out helper processes that take input from one process (usually via a pipe), perform a simple function on that input, then return data to the main process. Most objects associated with S85httpd are helper processes spawned by find when S85httpd starts.



We first evaluate how necessary it is to use the filtering rules described in Section 4. Consider an attack we experienced on March 12, 2003 that we named the *bind attack*. The machine on this day was quite busy: we were the target of two separate attacks (the *bind attack* and the *ptrace attack*), and one of the authors logged in several times to use the machine (mostly to look for signs of intruders, e.g., by running `netstat`, `ps`, `ls`, `pstree`). We detected the attack by noticing a modified system binary (`/bin/login`). EventLogger's log for this analysis period covered 24 hours and contained 155,344 objects and 1,204,166 events (all event counts in this paper count only the first event from a specific source object to a specific sink object).

Without any filtering, the dependency graph generated by GraphGen for this attack contains 5,281 objects and 9,825 events. While this is two orders of magnitude smaller than the complete log, it is still far too many events and objects for an administrator to analyze easily. We therefore consider what filtering rules we can use to reduce the amount of information presented to the administrator, while minimizing the risk of hiding important steps in the attack.

Figure 5 shows the dependency graph generated by GraphGen for this attack after filtering out files that were read but not written. The resulting graph contains 575 objects and 1,014 events. Important parts of the graph are circled or labeled to point out the filtering rules we discuss next.

Significant noise comes from several root login sessions by one of the authors during the attack. The author's actions are linked to the attacker's actions through `/root/.bash_history`, `/var/log/lastlog`, and `/var/run/utmp`. `/etc/mtab` also generates a lot of noise, as it is written after most system startup scripts and read by each bash shell. Finally, a lot of noise is generated by helper processes that take input only from their parent process, perform a simple function on that input, then return data to the parent (usually through a pipe). Most processes associated with `S85httpd` on the graph are helper processes spawned by `find` when `S85httpd` starts.

Figure 6 shows the dependency graph for the *bind attack* after GraphGen applies the following filtering rules: ignore files that were read but not written; ignore files `/root/.bash_history`, `/var/run/lastlog`, `/var/run/utmp`, `/etc/mtab`; ignore helper processes that take input only from their parent process and return a result through a pipe. We use these same filtering rules to generate dependency graphs for all attacks.

These filtering rules reduce the size of the graph to 24 objects and 28 events, and make the *bind attack* fairly easy to analyze. The attacker gained access through `httpd`, downloaded a rootkit using `wget`, then wrote the rootkit to the file `"/tmp/.bind"`. Sometime later, one of the authors logged in to the machine, noticed the suspicious file, and decided to execute it out of curiosity (don't try this at home!). The resulting process installed a number of modified system binaries, including `/bin/login`. This graph shows that BackTracker can track across several login sessions. If the attacker had installed `/bin/login` without being noticed, then logged in later, we would be able to backtrack from a detection point in her second session to the first session by her use of the modified `/bin/login`.

Figure 1 shows the filtered dependency graph for a second attack that occurred in the same March 12, 2003 log, which we named the *ptrace attack*. The intruder gained access through `httpd`, downloaded a tar archive using `wget`, then unpacked the archive via `tar` and `gzip`. The intruder then executed the `ptrace` program using a different group identity. We later detected the intrusion by seeing the `ptrace` process in the process listing. We believe the `ptrace` process was seeking to exploit a race condition in the Linux `ptrace` code to gain root access. Figures 1 and 6 demonstrate Back-

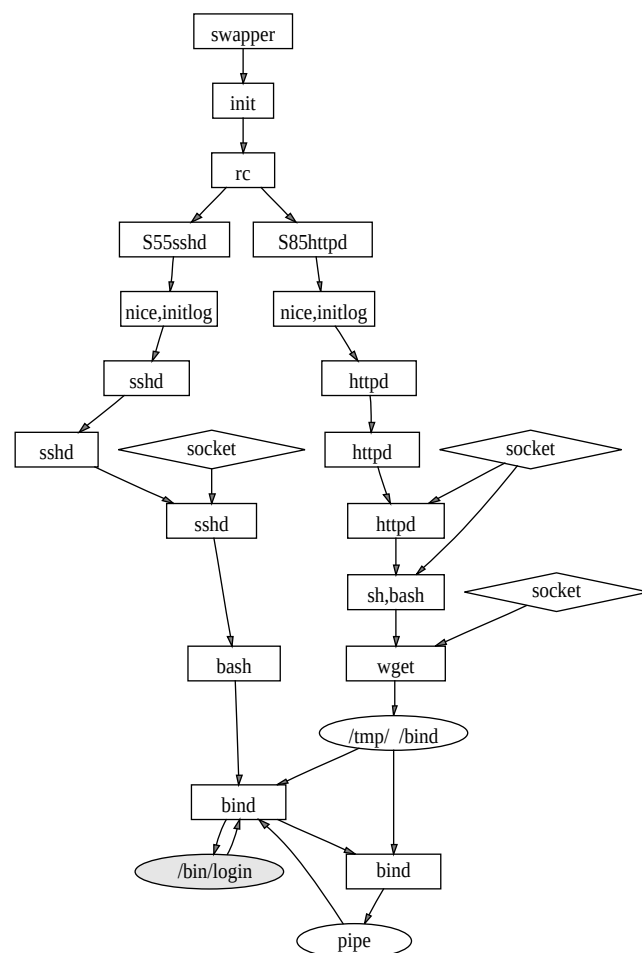


Figure 6: Filtered dependency graph for *bind attack*.

Tracker's ability to separate two intermingled attacks from a single log. Changing detection points from `/bin/login` to `ptrace` is sufficient to generate distinct dependency graphs for each attack.

Figure 7 shows the filtered dependency graph for an attack on March 2, 2003 which we named the *openssl-too attack*. The machine was used lightly by one of the authors (to check for intrusions) during the March 1-3 period covered by this log. The attacker gained access through `httpd`, downloaded a tar archive using `wget`, then installed a set of files using `tar` and `gzip`. The attacker then ran the program `openssl-too`, which read the configuration files that were unpacked. We detected the intrusion when the `openssl-too` process began scanning other machines on our network for vulnerable ports.

Another intrusion occurred on our machine on March 13, 2003. The filtered dependency graph for this attack is almost identical to the *ptrace attack*.

Figure 8a shows the default filtered dependency graph for an attack we conducted against our own system (*self attack*). *self attack* is more complicated than the real attacks we have been subjected to. We gain unprivileged access via `httpd`, then download and compile a program (`sxp`) that takes advantage of a local exploit against `sendmail`. When `sxp` runs, it uses `objdump` to find important addresses in the `sendmail` binary, then executes `sendmail` through `execve` to overflow an argument buffer and provide a root shell. We use this root shell to add a privileged user to the password files.

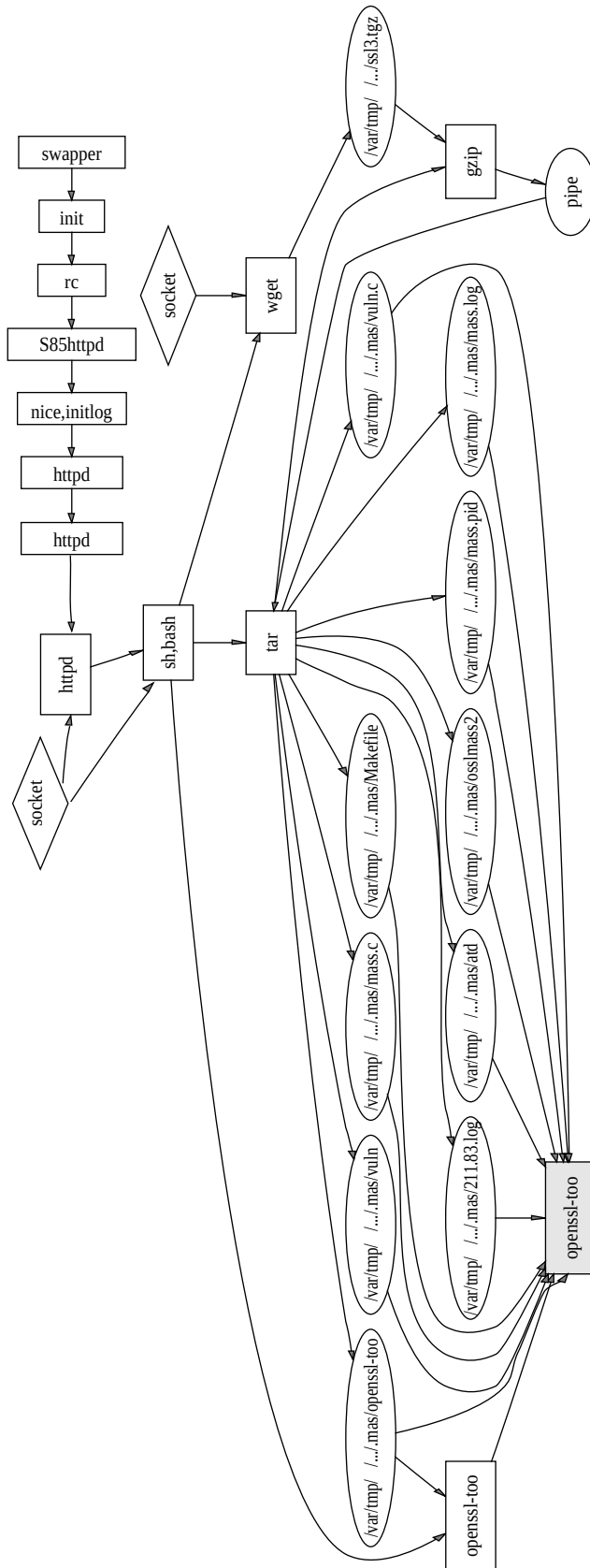


Figure 7: Filtered dependency graph for *openssl-too* attack.

Later, we log into the machine using this new user and modify `/etc/xinetd.conf`. The detection point for this attack is the modified `/etc/xinetd.conf`.

One goal for this attack is to load the machine heavily to see if BackTracker can separate the attack events from normal events. Over the duration of the workload, we continually ran the SPECweb99 benchmark to model the workload of a web server. To further stress the machine, we downloaded, unpacked, and continually compiled the Linux kernel. We also logged in several times as root and read `/etc/xinetd.conf`. The dependency graph shows that BackTracker separates this legitimate activity from the attack.

We anticipate that administrators will run GraphGen multiple times with different filtering rules to analyze an attack. An administrator can filter out new objects and events easily by editing the configuration file from which GraphGen reads its filter rules. Figure 8b shows the dependency graph generated with an additional rule that filters out all pipes. While this rule may filter out some portions of the attack, it will not usually disconnect the detection point from the from an intrusion source, because pipes are inherited from a process's ancestor, and BackTracker will track back to the ancestor through process creation events. In Figure 8, filtering out pipes eliminates `objdump`, which is related to the attack but not critical to understanding the attack.

Next we measure the space and time overhead of EventLogger (Table 1). It is non-trivial to compare running times with and without EventLogger, because real attackers do not re-issue their workload upon request. Instead we use ReVirt to replay the run with and without EventLogger and measure the difference in time. The replay system executes busy parts of the run at the same speed as the original run (within a few percent). The replay system eliminates idle periods, however, so the percentage overhead is given as a fraction of the wall-clock time of the original run (which was run without EventLogger).

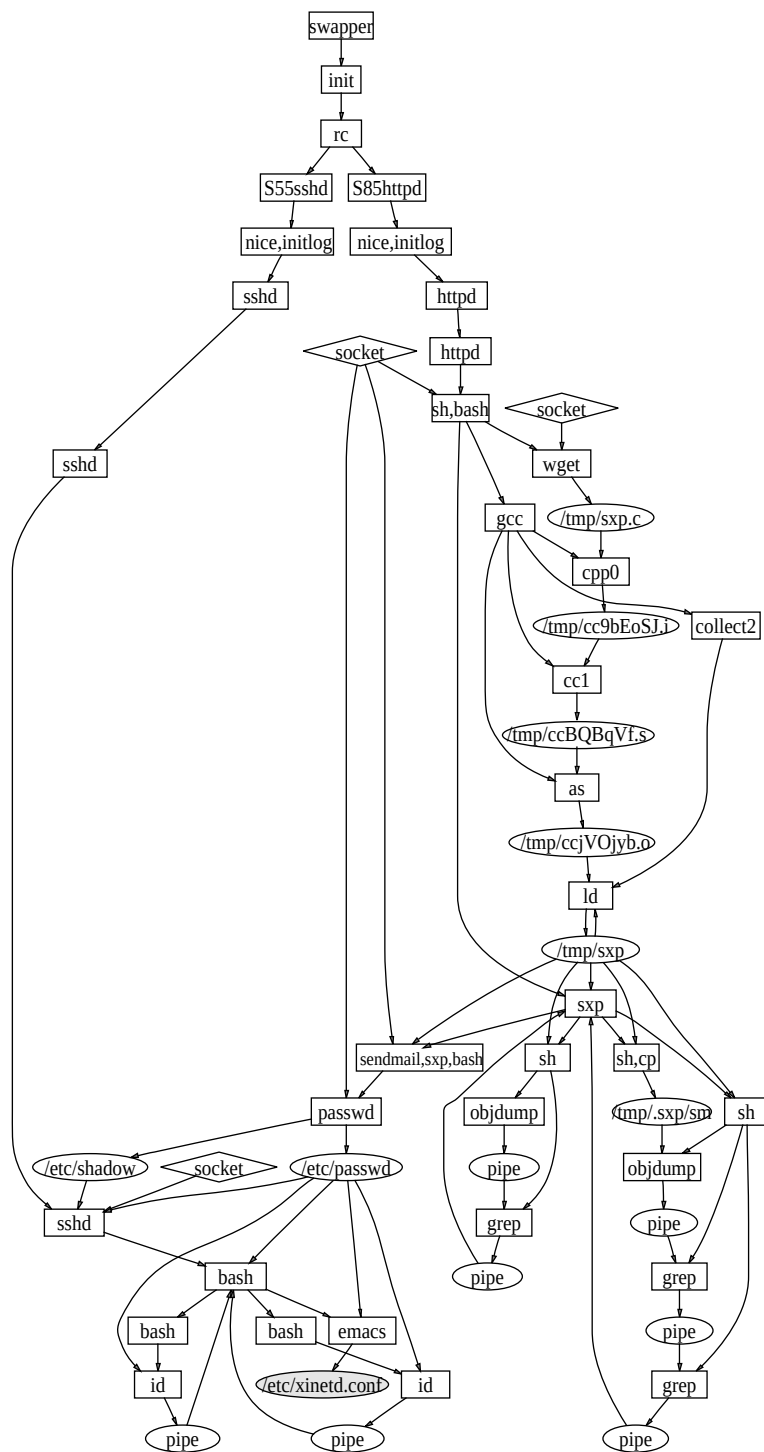
For the real attacks, the system is idle for long periods of time. The average time and space overhead for EventLogger is very low for these runs because EventLogger only incurs overhead when applications are actively using the system.

The results for *self attack* represent what the time and space overheads would be like for a system that is extremely busy. In particular, serving web pages and compiling the Linux kernel each invoke a huge number of relevant system calls. For this run, EventLogger slows the system by 9%, and its compressed log grows at a rate of 1.2 GB/day. While this is a substantial amount of data, a modern hard disk is large enough to store this volume of log traffic for several months.

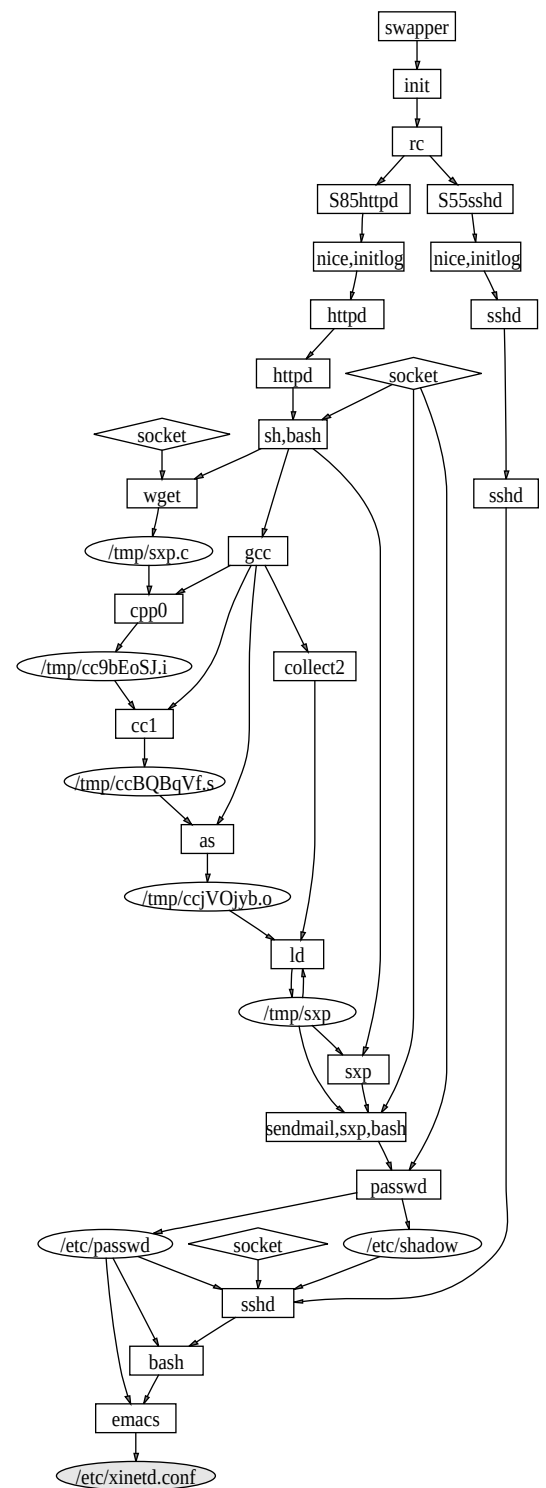
GraphGen is run after the attack (off-line), so its performance is not as critical as that of EventLogger. On a 2.8 GHz Pentium 4 with 1 GB of memory, GraphGen took less than 20 seconds to process the logs for each of the real attacks. GraphGen took just under 3 hours to process the log for the intensive *self attack*. We believe GraphGen's running time could be reduced severalfold (without affecting the performance of EventLogger) by making small changes to the format of the log written by EventLogger and by combining several of the processing stages that make up GraphGen.

6. ATTACKS AGAINST BACKTRACKER

In the prior section, we showed that BackTracker helped analyze several real attacks. In this section, we consider what an intruder can do to hide his actions from BackTracker. An intruder may attack the layers upon which BackTracker is built, use events that



(a) default filtering rules



(b) default filtering rules + filter out pipes

Figure 8: Filtered dependency graph for *self* attack. Figure 8a shows the dependency produced by GraphGen with the same filtering rules used to generate Figures 1, 6, and 7. Figure 8b shows the dependency graph produced by GraphGen after adding a rule that filters out pipes. Figure 8b is a subgraph of Figure 8a.

BackTracker does not monitor, or hide his actions within large dependency graphs.

An intruder can try to foil BackTracker by attacking the layers upon which BackTracker's analysis or logging depend. One such layer is the guest operating system. BackTracker's analysis is accurate only if the events and data it sees have their conventional meaning. If an intruder can change the guest kernel (e.g., to cause a random system call to create processes or change files), then he can accomplish arbitrary tasks inside the guest machine without being tracked by BackTracker. Many operating systems provide interfaces that make it easy to compromise the kernel or to work around its abstractions. Loadable kernel modules and direct access to kernel memory (`/dev/kmem`) make it trivial to change the kernel. Direct access to physical memory (`/dev/mem`) and I/O devices make it easy to control applications and files without using the higher-level abstractions that BackTracker tracks. Our guest operating system disables these interfaces [19]. The guest operating system may also contain bugs that allow an intruder to compromise it without using standard interfaces [7]. Researchers are investigating ways to use virtual machines to make it more difficult for intruders to compromise the guest operating system, e.g., by protecting the guest kernel's code and sensitive data structures [17].

Another layer upon which the current implementation of BackTracker depends is the virtual-machine monitor and host operating system. Attacking these layers is considerably more difficult than attacking the guest kernel, since the virtual-machine monitor makes the trusted computing base for the host operating system much smaller than the guest kernel.

If an intruder cannot compromise a layer below BackTracker, he can still seek to stop BackTracker from analyzing the complete chain of events from the detection point to the source of the attack. The intruder can break the chain of events tracked if he can carry out one step in his sequence using *only* low-control events that BackTracker does not yet track. Section 2.4 explains why this is relatively difficult.

An intruder can also use a hidden channel to break the chain of events that BackTracker tracks. For example, an intruder can use the initial part of his attack to steal a password, send it to himself over the network, then log in later via that password. BackTracker can track from a detection point during the second login session up to the point where the intruder logged in, but it cannot link the use of the password automatically to the initial theft of the password. BackTracker depends on knowing and tracking the sequence of state changes on the system, and the intruder's memory of the stolen password is not subject to this tracking. However, BackTracker will track the attack back to the beginning of the second login session, and this will alert the administrator to a stolen password. If the administrator can identify a detection point in the first part of the attack, he can track from there to the source of the intrusion.

An intruder can also try to hide his actions by hiding them in a huge dependency graph. This is futile if the events in the dependency graph are the intruder's actions because the initial break-in phase of the attack is not obfuscated by a huge graph after the initial phase. In addition, an intruder who executes a large number of events is more likely to be caught.

An intruder can also hide his actions by intermingling them with innocent events. GraphGen includes only those events that potentially affect the detection point, so an intruder would have to make it look as though innocent events have affected the detection point. For example, an intruder can implicate an innocent process by reading a file the innocent process has written. In the worst case,

the attacker would read all recently written files before changing the detection point and thereby implicate all processes that wrote those files. As usual, security is a race between attackers and defenders. GraphGen could address this attack by filtering out file reads if they are too numerous and following the chain of events up from the process that read the files. The attacker could then implicate innocent processes in more subtle ways, etc.

Finally, an attacker can make the analysis of an intrusion more difficult by carrying out the desired sequence of steps over a long period of time. The longer the period of attack, the more log records that EventLogger and GraphGen have to store and analyze.

In conclusion, there are several ways that an intruder can seek to hide his actions from BackTracker. Our goal is to analyze a substantial fraction of current attacks and to make it more difficult to launch attacks that cannot be tracked.

7. RELATED WORK

BackTracker tracks the flow of information [11] across operating system objects and events. The most closely related work is the *Repairable File Service* [29], which also tracks the flow of information through processes and files by logging similar events. The Repairable File Service assumes an administrator has already identified the process that *started* the intrusion; it then uses the log to identify files that potentially have been contaminated by that process. In contrast, BackTracker begins with a process, file, or filename that has been affected by the intrusion, then uses the log to track back to the source of the intrusion. The two techniques are complementary: one could use backtracking to identify the source of the intrusion, then use the Repairable File Service's forward tracking to identify the files that potentially have been contaminated by the intrusion. However, we believe that an intruder can hide her actions much more easily from the forward tracking phase, e.g., by simply touching all files in the system. Even without deliberately trying to hide, we believe an intruder's changes to system files will quickly cause all files and processes to be labeled as potentially contaminated. For example, if an intruder changes the password file, all users who subsequently log into the system will read this file, and all files they modify will be labeled as potentially contaminated.

In addition to the direction of tracking, BackTracker differs from the Repairable File Service in the following ways: (1) BackTracker tracks additional dependency-causing events (e.g., shared memory, mmap'ed files, pipes and named pipes); (2) BackTracker labels and analyzes time intervals for events, which are needed to handle aggregated events such as loads/store to mmap'ed files; and (3) BackTracker uses filter rules to highlight the most important dependencies. Perhaps most importantly, we use BackTracker to analyze real intrusions and evaluate the quality of the dependency graphs it produces for those attacks. The evaluation for the Repairable File Service has so far focused on time and space overhead—to our knowledge, the spread of contamination has been evaluated only in terms of number of processes, files, and blocks contaminated and has been performed only on a single benchmark (SPEC SDNET) with a randomly chosen initial process.

Work by Ammann, Jajodia, and Liu tracks the flow of contaminated transactions through a database and rolls data back if it has been affected directly or indirectly by contaminated transactions [6]. The Perl programming language also tracks the flow of tainted information across perl program statements [28]. Like the Repairable File Service, both these tools track the forward flow of contaminated information rather than backtracking from a detection point to the source of the intrusion.

Program slicing is a programming language technique that identifies the statements in a program that potentially affect the values at a point of interest [26]. Dynamic slicers compute the slice based on a specific set of inputs. BackTracker could be viewed as a dynamic program slicer on a self-modifying program, where variables are operating system objects, and program statements are dependency-causing operating system events.

Several other projects assist administrators in understanding intrusions. CERT's Incident Detection, Analysis, and Response Project (IDAR) seeks to develop a structured knowledge base of expert knowledge about attacks and to look through the post-intrusion system for signs that match an entry in the existing knowledge base [10]. Similarly, SRI's DERBI project looks through system logs and file system state after the intrusion for clues about the intrusion [27]. These tools automate common investigations after an attack, such as looking for suspicious filenames, comparing file access times with login session times, and looking for suspicious entries in the password files. However, like investigations that are carried out manually, these tools are limited by the information logged by current systems. Without detailed event logs, they are unable to describe the sequence of an attack from the initial compromise to the detection point.

8. CONCLUSIONS AND FUTURE WORK

We have described a tool called BackTracker that helps system administrators analyze intrusions on their system. Starting from a detection point, such as a suspicious file or process, BackTracker identifies the events and objects that could have affected that detection point. The dependency graphs generated by BackTracker help an administrator find and focus on a few important objects and events to understand the intrusion. BackTracker can use several types of rules to filter out parts of the dependency graph that are unlikely to be related to the intrusion.

We used BackTracker to analyze several real attacks against computers we set up as honeypots. In each case, BackTracker was able to highlight effectively the entry point used to gain access to the system and the sequence of steps from the entry point to the point at which we noticed the intrusion.

In the future, we plan to track more dependency-causing events, such as System V IPC, signals, and dependencies caused by file attributes. We have also implemented a tool to track dependencies forward. The combination of this tool and BackTracker will allow us to start from a single detection point, backtrack to allow an administrator to identify the source of the intrusion, then forward track to identify other objects that have been affected by the intrusion. Significant research will be needed to filter out false dependencies when tracking forward because, unlike for backward tracking, an intruder can easily cause an explosion of the dependency graph to include all files and processes.

9. ACKNOWLEDGMENTS

The ideas in this paper were refined during discussions with George Dunlap, Murtaza Basrai, and Brian Noble. Our shepherd Frans Kaashoek and the anonymous reviewers provided valuable feedback that helped improve the quality of this paper. This research was supported in part by National Science Foundation grants CCR-0098229 and CCR-0219085 and by Intel Corporation. Samuel King was supported by a National Defense Science and Engineering Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

10. REFERENCES

- [1] Steps for Recovering from a UNIX or NT System Compromise. Technical report, CERT Coordination Center, April 2000. http://www.cert.org/tech_tips/win-UNIX-system_compromise.html.
- [2] Detecting Signs of Intrusion. Technical Report CMU/SEI-SIM-009, CERT Coordination Center, April 2001. <http://www.cert.org/security-improvement/modules/m09.html>.
- [3] L-133: Sendmail Debugger Arbitrary Code Execution Vulnerability. Technical report, Computer Incident Advisory Capability, August 2001. <http://www.ciac.org/ciac/bulletins/l-133.shtml>.
- [4] CERT/CC Overview Incident and Vulnerability Trends. Technical report, CERT Coordination Center, April 2002. <http://www.cert.org/present/cert-overview-trends/>.
- [5] Multiple Vulnerabilities In OpenSSL. Technical Report CERT Advisory CA-2002-23, CERT Coordination Center, July 2002. <http://www.cert.org/advisories/CA-2002-23.html>.
- [6] Paul Ammann, Sushil Jajodia, and Peng Liu. Recovery from Malicious Transactions. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1167–1185, September 2002.
- [7] Ken Ashcraft and Dawson Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, May 2002.
- [8] Kerstin Buchacker and Volkmar Sieh. Framework for testing the fault-tolerance of systems including OS and network aspects. In *Proceedings of the 2001 IEEE Symposium on High Assurance System Engineering (HASE)*, pages 95–105, October 2001.
- [9] Bill Cheswick. An Evening with Berferd in Which a Cracker is Lured, Endured, and Studied. In *Proceedings of the Winter 1992 USENIX Technical Conference*, pages 163–174, January 1992.
- [10] Alan M. Christie. The Incident Detection, Analysis, and Response (IDAR) Project. Technical report, CERT Coordination Center, July 2002. <http://www.cert.org/idar>.
- [11] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.
- [12] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, December 2002.
- [13] Dan Farmer. What are MACtimes? *Dr. Dobbs's Journal*, October 2000.
- [14] Dan Farmer. Bring out your dead. *Dr. Dobbs's Journal*, January 2001.
- [15] Dan Farmer and Wietse Venema. Forensic computer analysis: an introduction. *Dr. Dobbs's Journal*, September 2000.
- [16] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedings of 1996 IEEE Symposium on Computer Security and Privacy*, 1996.
- [17] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In

Proceedings of the 2003 Network and Distributed System Security Symposium (NDSS), February 2003.

- [18] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 1996 USENIX Technical Conference*, July 1996.
- [19] Xie Huagang. Build a secure system with LIDS, 2000. http://www.lids.org/document/build_lids-0.2.html.
- [20] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: a file system integrity checker. In *Proceedings of 1994 ACM Conference on Computer and Communications Security (CCS)*, November 1994.
- [21] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating System Support for Virtual Machines. In *Proceedings of the 2003 USENIX Technical Conference*, pages 71–84, June 2003.
- [22] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution Via Program Shepherd. In *Proceedings of the 2002 USENIX Security Symposium*, August 2002.
- [23] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.
- [24] Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, October 1973.
- [25] The HoneyNet Project, editor. *Know your enemy: revealing the security tools, tactics, and motives of the blackhat community*. Addison Wesley, August 2001.
- [26] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.
- [27] W. M. Tyson. DERBI: Diagnosis, Explanation and Recovery from Computer Break-ins. Technical Report DARPA Project F30602-96-C-0295 Final Report, SRI International, Artificial Intelligence Center, January 2001. <http://www.dougmo-ran.com/dmoran/publications.html>.
- [28] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl, 3rd edition*. O'Reilly & Associates, July 2000.
- [29] Ningning Zhu and Tzi-cker Chiueh. Design, Implementation, and Evaluation of Repairable File Service. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN)*, June 2003.

APPENDIX B

Enriching intrusion alerts through multi-host causality

Samuel T. King, Z. Morley Mao, Dominic G. Lucchetti, Peter M. Chen

Department of Electrical Engineering and Computer Science
Computer Science and Engineering Division
University of Michigan
<http://www.eecs.umich.edu/CoVirt>

Abstract

Current intrusion detection systems point out suspicious states or events but do not show how the suspicious state or events relate to other states or events in the system. We show how to enrich an IDS alert with information about how those alerts causally lead to or result from other events in the system. By enriching IDS alerts with this type of causal information, we can leverage existing IDS alerts to learn more about the suspected attack. Backward causal graphs can be used to find which host allowed a multi-hop attack (such as a worm) to enter a local network; forward causal graphs can be used to find the other hosts that were affected by the multi-hop attack. We demonstrate this use of causality on a local network by tracking the Slapper worm, a manual attack that spreads via several attack vectors, and an e-mail virus. Causality can also be used to correlate distinct network and host IDS alerts. We demonstrate this use of causality by correlating Snort and host IDS alerts to reduce false positives on a testbed system connected to the Internet.

1. Introduction

Intrusion detection systems (IDSs) are used to alert system administrators to possible attacks. IDSs monitor events at various levels in a system and attempt to alert administrators to events that match known attacks (signature-based IDS) or do not match normal behavior (anomaly-based IDS). Host-based IDSs monitor events on a single host, such as system calls or file accesses. Network-based IDSs monitor packets that are sent or received.

One factor that limits the accuracy and usefulness of IDS alerts is the lack of accompanying context. IDS alerts point out suspicious events without relating them to other events and state in the system. For example, Snort [1]

highlights suspicious messages but does not describe how the receiving system reacted to these messages or what caused the sending system to send the suspicious message. Tripwire [6] alerts administrators to modified system files but does not show what caused the modifications or how the modified system files were used afterward. Additional context may help an administrator see the connections between different alerts or find other hosts that have been compromised.

This paper describes how to enrich an IDS alert with causal information about the state or events detected by the IDS. Informally, causal information shows which events led to the state/event detected by the IDS and which events were affected by the state/event detected by the IDS. Formally, this causal information is defined by Lamport's *happens-before* relationship [8], which results from messages between hosts and interprocess communication within a host. For example, a causal dependency is created from process A to process B if process A forks process B, if process A sends a message to process B, or if process A writes a file that is later read by process B. By iteratively analyzing these causal dependencies, we compute the transitive closure of causal relationships in the form of a multi-host causality graph. A causality graph with an IDS alert as a root node shows which events were affected by the state/event that generated the alert. A causality graph with an IDS alert as a leaf node shows which events led to the state/event that generated the alert.

Causality graphs can improve the effectiveness of IDS alerts, as in the following scenarios:

A worm exploits a vulnerability in a company's public web server, then starts infecting machines that are behind the company firewall. An IDS on one of the internal machines detects anomalous activity, but no IDS alerts occur on the company's public web server. A causality graph can trace a worm from the machine that detected the intrusion back to the company's public web server that allowed the worm into

the network. Figures 1–3 demonstrate this capability for a testbed of machines infected by the Slapper worm.

- An intruder logs in as a privileged user and schedules a job (e.g., through the root’s crontab file) for that night that will scan slowly for other vulnerable hosts. While logging in as a privileged user and issuing a slow port scan are both suspicious, neither is suspicious enough on its own to trigger an alert to the administrator. A causality graph can show that the login session that wrote the crontab file indirectly led to the slow port scan. Figure 6 demonstrates how a causality graph linked two IDS alerts during an attack on one of our testbed computers.

The contributions of this paper are (1) to propose causality graphs as a way to enrich IDS alerts; (2) to show how to enhance the BackTracker system [7] to create bi-directional, multi-host causality graphs; (3) to demonstrate how causality graphs can help track the propagation of multi-host attacks within an administrative domain; and (4) to demonstrate how causality graphs can help correlate otherwise disconnected IDS alerts.

The following is an overview of the paper. Section 2 describes how the original BackTracker system finds and displays objects and events from a single host that causally precede a given intrusion detection point, and Section 3 describes how we expand the scope of BackTracker to find and display events from a *network of hosts* that causally precede *or causally follow* a given intrusion detection point. We call this extended system BDB, which stands for Bi-directional, Distributed BackTracker. BDB represents a mechanism that administrators can use to enrich IDS alerts; administrators can then use various policies on top of this mechanism. Sections 4 and 5 describe several scenarios that demonstrate how an administrator might use the information provided by BDB to track multi-hop attacks (worms, manual attacks, and e-mail viruses) and correlate disconnected IDS alerts. Section 7 describes some limitations of our approach, and Section 8 relates our work to prior research.

2. BackTracker

We introduced the idea of using causality to analyze intrusions with the BackTracker system [7]. BackTracker uses a modified Linux kernel to log system calls that form dependencies between operating system objects. BackTracker analyzes dependencies between three types of objects: processes, files, and filenames. For example, a $\text{process} \Rightarrow \text{file}$ dependency is formed when a process writes a file, and a $\text{file} \Rightarrow \text{process}$ dependency is formed when a process reads a file. Files are identified by inode number

to allow BackTracker to track a file across rename operations and through symbolic links. A $\text{process} \Rightarrow \text{process}$ dependency is formed when one process creates another process. Bi-directional dependencies are also possible, such as when two processes share memory with each other ($\text{process} \Leftrightarrow \text{process}$). See [7] for more details on the objects and events analyzed by BackTracker.

BackTracker’s analysis starts from a suspicious object or event detected by an IDS. BackTracker is independent of which IDS is used, as long as the IDS can identify a suspicious operating system event or object. BackTracker analyzes the causal relationships formed between objects to construct a graph of the objects and events that led to the specified intrusion detection point. We call this a *backward graph* because it looks back in time from a given IDS alert. BackTracker then filters out the dominant sources of noise in the backward graph according to default filtering rules. For example, the login program reads and writes the file `/var/run/utmp`, which makes it appear that each new login session is affected directly by all prior login sessions. BackTracker’s default rules seek to filter out events that generate a lot of noise in the backward graph but enable only a low degree of control between objects. New rules can be added easily by an administrator, and we use this to add 1-3 simple rules for each network service used in this paper (web, FTP, SMB). For example, we filter out reads and writes to the list of processes maintained by the FTP server (`/var/run/ftp.pids-all`). See [7] for a fuller discussion of the use of filtering rules.

[7] showed that combining causal analysis and a set of six simple rules was very effective. We analyzed multiple attacks on a machine we had set up as a honeypot, and in each case the resulting backward graph highlighted effectively the source of the intrusion and the path between the initial compromise and the intrusion detection point.

Figure 2 shows BackTracker’s output for a compromise caused by the Slapper worm [2]. After an IDS detected a suspicious process (`update`), BackTracker generated the backward graph that led to this detection point. This graph shows that the worm compromised the Apache web server (`httpd`) and caused it to execute a bash shell. The bash shell downloaded and unpacked a uuencoded tar file, which contains the Slapper executable and the source code for another program (`update.c`). The bash shell compiled this program, then ran the slapper process, which ran the newly compiled update program.

3. Bi-directional, multi-host causality

The original BackTracker tool helps system administrators determine how an attack occurred by showing the chain of events that led up to the point at which an intrusion was detected. In this section, we show how to generalize BackTracker’s use of causality in two ways: track-

ing causal effects forward as well as backward, and using backward and forward causality across multiple hosts on a network. We then describe new filtering rules to prioritize the causal paths most likely to describe an intrusion as it traverses across multiple hosts.

3.1. Forward tracking

BackTracker shows which events preceding an intrusion detection point could have contributed to the modified state or event that was detected. We would like to generalize this approach to analyze in the forward direction. Analyzing causality in the forward direction answers the question “What events and state were affected by the intrusion detection point?”. Consider a scenario in which an attacker replaces the `/bin/ls` executable with a program that sends a user’s private files to a collection site. Once an IDS detects that `/bin/ls` was modified, forward tracking will show which files of which users were leaked as a result; this could help limit the damage done by the intrusion (e.g., by informing the user which credit cards should be canceled).

Forward causal analysis is a straightforward extension to backward analysis. Forward tracking uses the same log of system calls and objects logged by BackTracker. Whereas BackTracker adds events to a graph if they affect an object before that object’s time threshold, forward tracking adds events to a graph if they were affected by an object after that object’s time threshold.

3.2. Multi-host causality

BackTracker’s analysis is limited to events that occurred on the same host as the state or event that was detected by the IDS. Because many attacks propagate via the network, we would like to generalize this approach to track intrusions as they infect multiple hosts. This will allow us to leverage one host’s IDS alert to find other compromised hosts upstream (using backward causality) and downstream (using forward causality). As with host-level causality, this type of tracking is limited to machines under our administrative control.

To track causality across multiple hosts, we extend BackTracker’s logging to also track network sends and receives. If process 1 on machine A sends a packet to process 2 on machine B, this forms an inter-host causal dependency from process 1 to process 2. Connecting a send event with its corresponding receive event requires identifying each packet (or connection). There are several ways to identify packets. The simplest is to use information in existing network headers, such as the source and destination IP address, port number, and sequence number (for TCP messages)¹. Another approach is to supplement mes-

sages with additional information [10], either at network routers or using our modified kernel. Finally, one could identify packets by storing a hash of their contents. We currently identify packets by their source and destination addresses and sequence number; this simple approach is sufficient for the TCP-based attacks evaluated in this paper.

3.3. Prioritizing packets

Tracking the causal relationships caused by network communication can lead to extremely large graphs. Consider the following scenario: an intruder attains a login session on an internal computer (A). He then uses that login session to compromise an internal SMB server (B). From the SMB server, he browses the departmental web server, then launches a worm. Using one of the outgoing worm messages as a starting point, an administrator can generate a multi-host, backward causality graph. This graph will include the key link in the attack, which is the message from the login session (A) to the SMB server (B). However, the graph will also include the irrelevant messages from the departmental web server. To make matters worse, the graph may also include other clients that happened to affect the execution of the departmental web server before that server sent pages to the SMB server.

Another example scenario is if a network service handles a series of requests in a single process. If this network service is compromised, it will causally depend on all prior incoming requests. Fortunately, network services are often built to create a new process to handle each (or a few) incoming requests. Creating a new process helps to limit the set of incoming packets that causally precede an intrusion.

In order to counteract the tendency of graphs to explode in size, we must prioritize which packets to include in multi-host causality graphs. This is analogous to filtering host events that are less likely to be critical steps in an intrusion. Prioritizing packets leads to the same tradeoffs inherent to any kind of filtering. Even objects or events that are unlikely to be important in understanding an intrusion may nevertheless be relevant, and filtering these out may accidentally hide important sequences of events.

There are numerous methods one could use to prioritize which packets to follow in backward or forward causal analysis. The first method is a simple heuristic that works well for today’s simple worms. A common pattern of today’s worms is to connect to a network service, compromise it (e.g., with a buffer overflow), then immediately start a root shell or a backdoor process. For this pattern, the best process to follow when performing backward causality process is the highest (i.e., earliest) pro-

use ingress filtering to check that these messages do not spoof an internal network address [5]

¹The gateway that receives messages from outside the network can

cess in the backward graph that received a network packet. The best packet to follow for this pattern is the most recent packet received by the highest process. We call this heuristic *highest process, most recent packet*.

A more general and robust method for prioritizing packets is to choose packets that are related causally to another IDS alert. For example, an administrator may use forward tracking from a host that has detected a backdoor process. The forward causal graph from this process will show all hosts that received a message from the backdoor process, as well as the subsequent actions on those hosts that were caused (directly or indirectly) by receiving that message. Some of these messages are likely innocent or ineffective, while others may be malicious and effective. Messages that are malicious and effective are more likely to lead causally to other IDS alerts and should therefore be prioritized. These other IDS alerts may be from a network anomaly detector that points out suspicious messages directly, or they may be from a host IDS that detects suspicious host activity that results indirectly from these messages.

Another method of prioritizing packets leverages the fact that most worms repeat their actions as they traverse hosts. They usually propagate from host to host using the same exploit, or perhaps using one of a few exploits. They may also perform the same activities on each host they compromise, such as installing backdoors, patching vulnerabilities, or scanning for private information. These repeated actions form a pattern that characterizes the worm. As we causally follow an intrusion alert from host to host (forward or backward), we can learn this pattern and use it to prioritize packets that cause similar patterns on other hosts.

A pattern for a worm may be characterized in many ways. It could be the set of files or processes that causally follow from a received packet. More generally, one could characterize a worm by the topology of the causal graph resulting from the received packet. In fact, one can view the topology or membership of a packet's forward causality graph as a signature of a worm and raise an IDS alert whenever one sees this signature. Similarly, one can view the topology or membership of a network service's causality graph as a profile of that network service and raise an alarm whenever one sees an anomalous causality graph.

Finally, in some cases one can take advantage of application-specific knowledge to identify more precisely which incoming packet causes a given action. For example, a mail client may be able to inform the causality tracking system which downloaded mail message contained the attachment that is being viewed. If the viewer is compromised while viewing that attachment, the causality tracker can focus on the appropriate message.

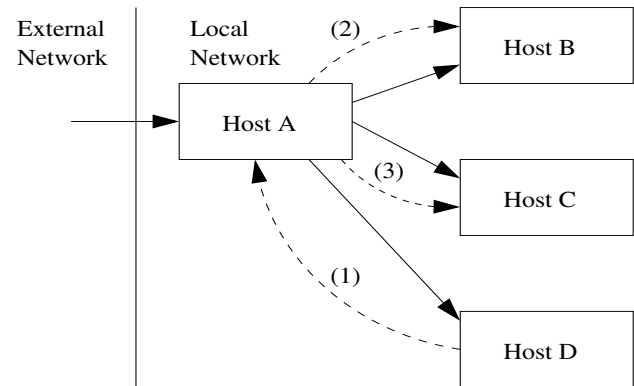


Figure 1. Inter-host propagation of the Slapper worm. The worm first infects Host A, then launches attacks against Hosts B, C, and D. After an IDS detects the attack on Host D, BDB tracks the attack backward to Host A, then tracks it forward to Hosts B and C. The solid lines depict the attacks, and the dotted lines depict the order of BDB's analysis.

4. Tracking multi-hop attacks

The prior sections introduced the mechanisms used by BDB to connect multiple alarms together. In the next two sections, we discuss and demonstrate several ways to use these mechanisms. In this section, we track attacks that traverse several hosts within an administrative domain.

4.1. Overview

Many organizations place most of their computers and services on an intranet behind a firewall, with only a few computers and services exposed to the public Internet. Relying too heavily on this type of perimeter defense has well-known flaws: if an attacker breaches the firewall (such as through an infected laptop, email virus, or bug in one of the public network services), he gains access to the computers on the local network, which are often less secured. To clean up after such a breach, the administrator must first find all the computers that have been compromised. It can be quite difficult to find which computers have been compromised during an attack, because the attacker may break into a system, steal or corrupt data, then clean up by removing telltale signs of his attack.

In this section, we show how BDB can track attacks that traverse multiple hosts, even when the intrusion is detected initially on only a single host. After an intrusion is detected on a single host, BDB tracks backward to determine what led to the intrusion on that host. Tracking

back continues across nodes until it reaches the point at which the attack entered the intranet. Finding the point the attack entered the intranet is useful in securing the network against future attacks. If the attack entered via an infected laptop, the administrator can chastise and educate the user; if the attack entered via a buggy network service, the administrator can disable that service or apply a patch [13]. After backward tracking is complete, BDB performs forward tracking to find other compromised hosts. As BDB finds compromised hosts during backward and forward tracking, it provides the opportunity to learn about the attack. The information it learns can be used to develop network and host signatures of the attack that can be used to avoid future compromises or scan other networks for compromised hosts.

As discussed in Section 3.3, prioritizing which packets to follow can be done in a variety of ways. In Section 4.2, we track the Slapper worm backward using the *highest process, most recent packet* heuristic. During the forward tracking phase, we prioritize which packets to follow by leveraging the signatures learned about the worm while tracking it backward. In Section 4.3, we prioritize the packets by following those flagged as suspicious by the Snort network IDS. In Section 4.4, we use application-specific instrumentation to identify which e-mail message led to later events on the host.

4.2. Slapper worm

We first demonstrate the ability of BDB to track attacks across multiple hosts by releasing a modified version of the Slapper worm on a local worm testbed (Figure 1). We modified the spreading pattern of Slapper to look for new victims on the local network before using its default method of searching for vulnerable machines among randomly generated IP addresses.

Our testbed contains four vulnerable web servers: one accessible from both the public Internet and the local network, and three accessible only from the local network. To make it harder to track the worm, we add background noise by running the SPECweb99 benchmark between the web servers. Two machines acted as SPECweb99 servers, and the other two machines acted as SPECweb99 clients. This workload generated 95,943 operating system objects and 1,628,937 events spread out over the four machines. The test was run for 20 minutes, with the worm being released 10 minutes after the test began.

The initial detection point occurred on Host D when an IDS detected a suspicious process named `update`; Figure 2 shows the causality graph on Host D that led to this process. The `update` process is used by Slapper to spread the worm to other machines. This process on Host D is the starting point of our backward traversal. The causality graph shows that `update` is compiled from

source code, which is downloaded by a shell over a socket. The shell is started by a compromised web server.

Using the *highest process, most recent packet* heuristic, BDB identifies a packet received by the web server process as likely to be part of the attack. That packet originated from Host A and is the starting point for backward tracking on that host. Tracking the packet on Host A results in sequence of events similar to those found on Host D. However, the *highest process, most recent packet* heuristic identifies a packet that originated from an external source, so BDB's traversal backward stops.

While tracking the worm backward from Host D to Host B, BDB collects information about the worm in the form of a backward causality graph. These graphs are signatures that describe how the worm behaves on a computer. BDB uses this information to go forward and detect other instances of the worm.

The forward traversal begins with the `httpd` process on Host A since that is the starting point of the worm (Figure 3). BDB initially considers all activity resulting from this point. In addition to communicating with Host D, the worm also contacts Hosts B and C. BDB uses forward tracking to examine the effects of each outgoing packet on the receiving hosts. To determine if the worm had infected these machines, BDB searches the forward graphs for the causality signatures found while traversing backward. In both cases, the signature is found, and the hosts are deemed exploited.

BDB tracks the Slapper worm effectively using the *highest process, most recent packet* heuristic and matching host worm signatures found during the backward traversal phase. Despite the heavy load on the network service that was exploited, BDB filtered out irrelevant events and highlighted only the actions of the worm. The backward and forward graphs of all of the infected nodes contain a total of 171 objects and 251 events, which is 2-3 orders of magnitude less than the total activity on the system.

4.3 Multi-vector manual attack

Although the *highest process, most recent packet* heuristic is effective for the Slapper worm and many other existing exploits we examined, an attacker that knows about this heuristic could evade detection. For example, an attacker could implicate an innocent request by receiving it between the initial exploit and the next phase of the attack. Another difficult scenario occurs when there are multiple network services at different roots of the backward graph; in this case, there is no unique highest process.

One way to address these shortcomings is to use a network IDS to prioritize which packets to follow. While a network IDS may not be accurate enough to track attacks

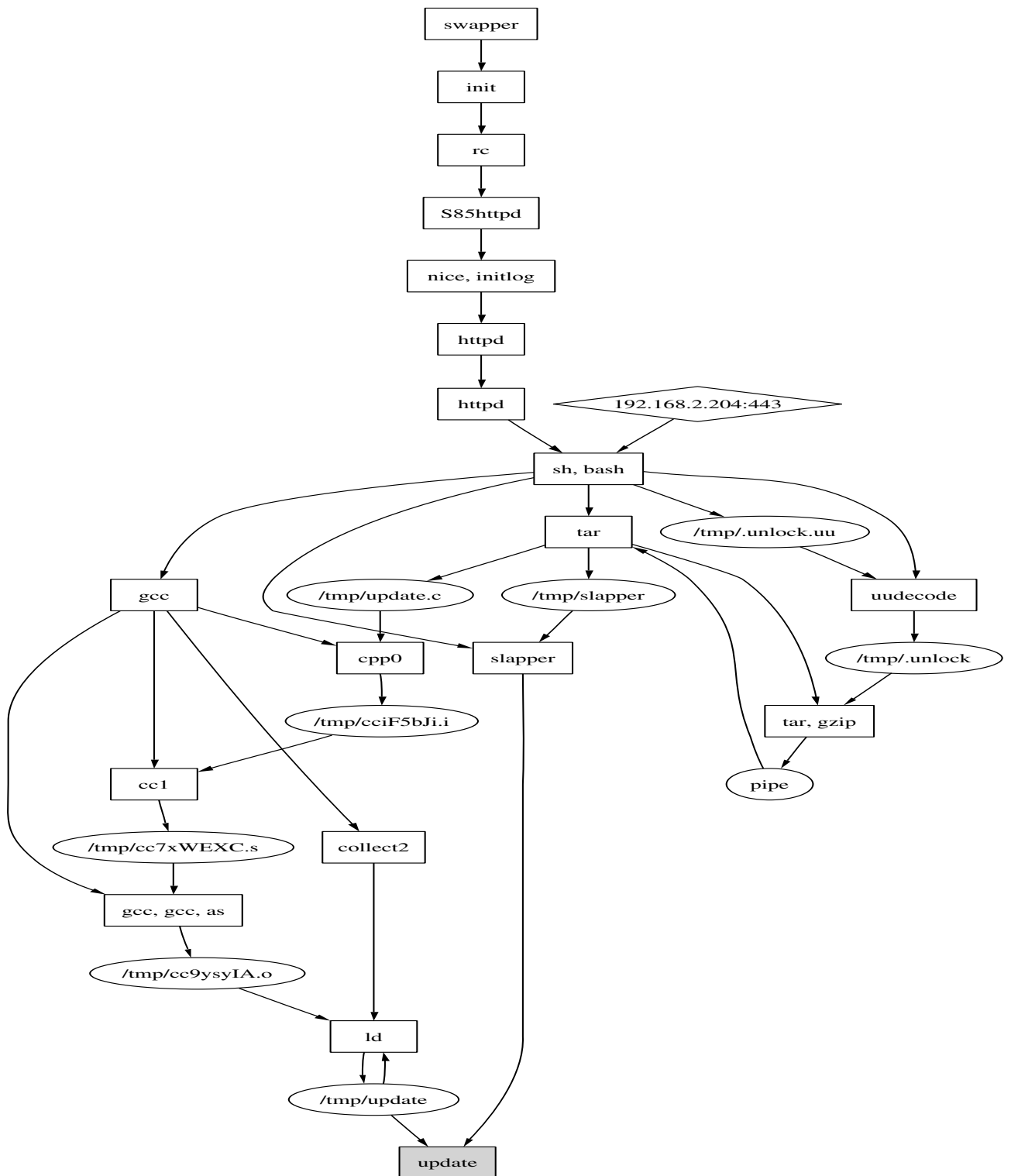


Figure 2. Backward causality graph for Slapper worm for Host D. Processes are shown as boxes (labeled by program names called by execve during that process's lifetime); files are shown as ovals; sockets are shown as diamonds. BackTracker can also show process IDs, file inode numbers, and socket ports. The intrusion detection point is shaded.

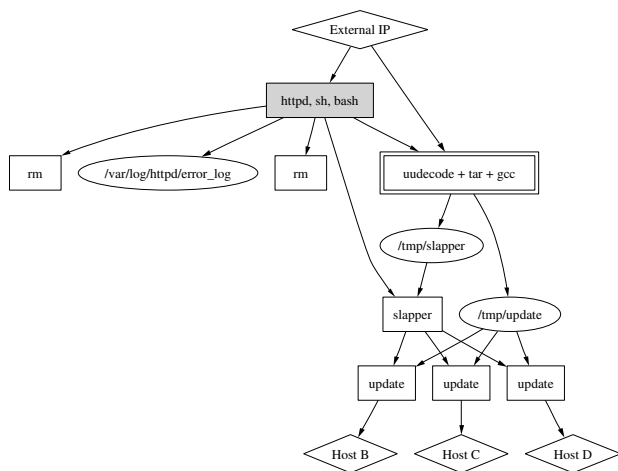


Figure 3. Forward causality graph of Host A for the Slapper attack. The subgraph formed by executing uudecode, tar, and gcc is represented by the box with double lines.

by itself, it may be accurate enough to prioritize which packets BDB should follow. In this paper, we use the network IDS Snort [1] to prioritize packets for BDB. Snort matches messages against a number of known signatures; for this test, we use rules that find commonly used shell-code instructions. For example, Snort detects consecutive no-op instructions, which are used often in buffer overflow attacks to compensate for variations in process address spaces. Although this type of network anomaly detector is prone to false positives, it is effective enough to prioritize which causally-related network packets to examine first.

To evaluate whether a network IDS can prioritize packets effectively, we carried out an attack on our local network that exploits several network services on various nodes within our system (Figure 4). We simulate a stealthy attacker by compromising one host at a time, rather than scanning the network and attacking all vulnerable hosts. We start by breaking into a publicly accessible web server. From there, we compromise succeeding hosts through vulnerabilities in the ftp and samba servers. Each attack results in an interactive shell, and we use that shell to download tools and break into the next host.

Our testbed includes 12 hosts: four web servers, four ftp servers, and four samba servers. Each of the nodes also doubles as a client. To make it harder to track the attack, we add background noise through artificial users who are logged into each computer. Each user mounts all four samba servers in his file space. The users down-

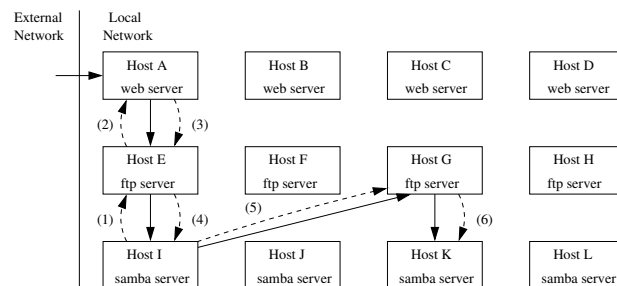


Figure 4. Inter-host propagation of the multi-vector attack. We carried out an attack on five of the 16 testbed hosts (A, E, I, G, K) using multiple attack vectors (solid lines). After an IDS detects the attack on Host I, BDB tracks the attack backward to Host E and A, then tracks it forward to Hosts E, I, G, and K.

load source code from a randomly selected web or ftp server, then unpack this source code to a randomly selected samba mounted directory and compile it. The entire process is repeated for the duration of the test. These activities result in large amounts of noise; across all twelve nodes there are over 2 GB of network traffic, 6,589,526 operating system events and 814,262 operating system objects. The test was run for 20 minutes, and the intrusions occurred 10 minutes after the test started.

The attack is detected initially on Host I, when the attacker launches a backdoor process that opens a raw socket. BDB generates a backward graph using the `backdoor` process as the detection point, then sees which of the causally related incoming packets were flagged as suspicious by Snort. In this case, one of the incoming packets in the backward graph had been flagged as suspicious by Snort. The suspicious packet came from the ftp server on Host E. Using the suspicious packet as the starting point for another BackTracker iteration, BDB again found that one of the causally related packets on Host E's backward graph had been flagged as suspicious by Snort. This packet led us back to the public web server on Host A. All causally related incoming packets on Host A are from external connections, so the backward traversal ends here.

Starting from the external web server, BDB uses Snort to prioritize among the causally related outgoing packets, with the goal of finding other compromised hosts. The forward analysis led us back to Host E and then again to Host I. From Host I, BDB found that the attacker broke into Host G and then Host K.

In the end, BDB found all of the infected hosts and highlighted 420 operating system objects and 19 network packets of the 814,262 objects and 2 GB of network data on the entire system. BDB's resulting causal graph is small enough that an administrator who wants to understand the attack in more detail can examine each object and packet by hand.

Although BDB found all compromised hosts, there were some false positives in our analysis. In particular, Snort flagged as high priority seven packets that unsuccessfully attempted to use the samba exploit. These did not affect our analysis because none of the unsuccessful break-in attempts generated any extraneous network activity, and all of the attacked hosts were eventually broken into. To reduce these false positives, BDB could prioritize network packets further by using additional IDSs; for instance, it could see which suspicious network packets led to other host IDS alerts on the receiving host.

4.4. E-mail virus

Attacks often propagate through a network via e-mail viruses. E-mail viruses typically spread by fooling a user into running a suspicious application or by exploiting a helper process used to handle an attachment. Unfortunately, the structure of e-mail handling programs makes it difficult for BDB to track the causal relationship between incoming messages and subsequent actions. E-mail servers that receive messages and e-mail clients that display them for users tend to be long-lived and to read multiple messages at a time. Because BDB tracks causality at the granularity of a process, it assumes conservatively that all messages that have been read affect all subsequent actions. Tracking causality at a more precise granularity requires program-specific instrumentation, and this section demonstrates how one can enhance BDB in this way to track e-mail viruses.

We made two application-specific changes to support e-mail tracking across multiple hosts. First, we modified the exim e-mail server to link the network messages it receives with e-mail messages IDs. Second, we modified the pine e-mail client to inform BDB of the "current" e-mail message ID being read at a given time. We assume that the saving of an attachment or launching of a helper process is caused by the current e-mail message being read by the user. These changes allow us to track (backward or forward) the effects of an e-mail message as it is received by the server, transferred to the client, and viewed by a user. Resulting actions on a host, such as writing files, starting new processes, and sending messages, are tracked via BDB's normal mechanisms.

We tested the ability of BDB to track an e-mail virus by installing our instrumented e-mail server and client on our testbed, then releasing a virus. Figure 5 shows how

BDB's resulting causality graph is able to track the virus backward from the initial detection on Host A (Figure 5a), back to when the virus was received by the exim server (Figure 5b), back to the upstream Host B (Figure 5c).

5. Correlating multiple alerts within and across nodes

In this section, we discuss the benefits of correlating multiple IDS alerts using causality, and we show how correlating alerts with causality can reduce false positives on a testbed system.

5.1. Overview

A major problem with many IDSs is false positives. One approach to reduce these false positives is to combine multiple host or networks IDS alerts into a single, higher-confidence alert. Prior approaches connect distinct alerts through statistical correlation on various features of the alert, such as the destination IP address or time of the alert. BDB makes it possible to correlate alerts in a new way by revealing which alerts are related causally.

The main benefit of relating alerts causally is its potential for increased accuracy—statistical correlation may suffer from coincidental events, whereas causal relationships are determined by chains of specific OS and network events.

In addition, using causality to correlate alerts can reduce dramatically the amount of data each IDS needs to process. For example, in Section 4.3, we showed how BDB revealed which outgoing network packets were causally related to a host IDS alert. BDB can thus narrow the search for suspicious network activity to a handful of packets, even in the midst of a busy network. In the same way, BDB can reduce the amount of data that a host IDS must examine. For example, a host IDS need examine only those processes and files that are related causally to packets that are flagged as suspicious by a network IDS. This allows one to use intensive host IDSs that would otherwise be too slow [12].

Two IDS alerts may be related causally in a variety of ways. One alert may causally precede or follow the other. Or, two alerts may share a common ancestor, such as when a single shell process executes two child processes that each perform a suspicious action. It is up to higher-level policies to determine how to score these different causal relationships, and we leave this to future work. We speculate that alerts are more likely to be correlated if one is the direct descendant of the other than if they simply share a common ancestor.

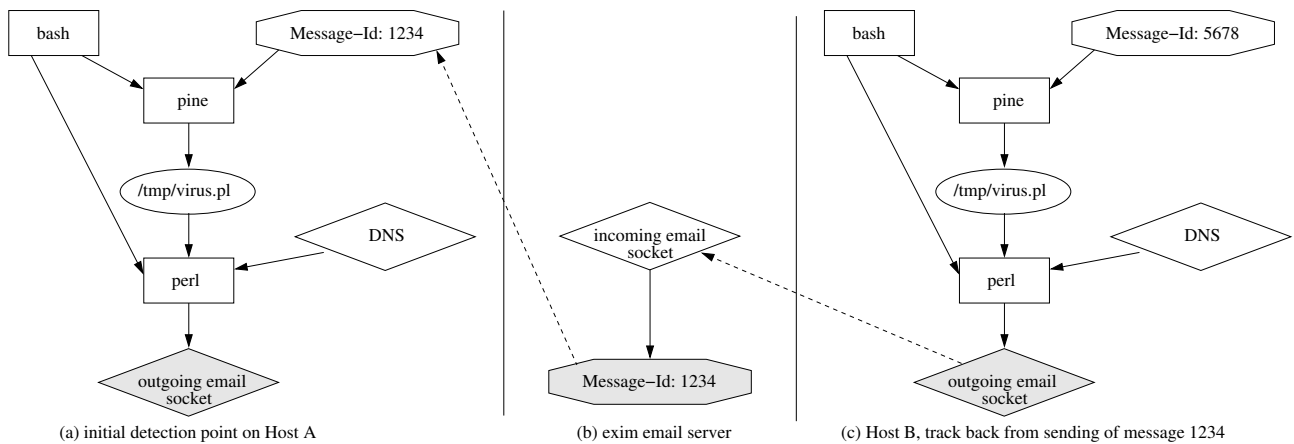


Figure 5. BackTracking an e-mail virus. In (a), BDB uses its normal tracking plus an instrumented e-mail client to track back from an outgoing network connection to the causally related incoming e-mail message. In (b), BDB uses the instrumented e-mail server to link the suspicious e-mail message with the incoming network message that delivered that e-mail message. This procedure identifies the host that sent the e-mail virus, where the backtracking can be repeated.

5.2. Results

To test how effectively causality can be used to correlate alerts, we ran BDB on two test systems that were exposed to the Internet. One system was running the default RedHat 6.2 installation; the other was running the default RedHat 7.0 installation. Both systems had several vulnerable servers that were accessible from the Internet, including the bind, ftp, and web servers. We configured the system to use a network IDS and a host IDS. We used Snort with its default rules as a network IDS, and we used a host IDS that flagged as suspicious any process that runs as root. While both IDSs are expected to generate many false positives, correlating alerts from these two IDSs using causality can increase our confidence that alerts result from actual compromises. To correlate alerts, we started with Snort alerts and generated causal graphs on the host for each suspicious network message. We then reported any processes in the resulting causal graphs that ran as root.

We ran the system for two days.² During this period, Snort generated 39 alerts, and the host IDS detected numerous root processes. Of these alerts, BDB detected two that were connected causally: an outgoing message flagged by Snort and two root processes. Figure 6 shows that the alerts were connected causally through a common

²We also used BDB on one of our desktop computers to correlate Snort and host IDS alerts. Although we were not attacked successfully over this period, BDB was effective at filtering out the numerous false positives generated by Snort.

ancestor (the httpd process). We confirmed by hand that these two alerts were indeed the result of a successful attack. Snort's alert resulted from a suspicious outgoing packet (triggering the "uid=" rule that indicates the output of the id command) that was sent by a shell process executed by a compromised web server. The shell process also downloaded various tools and gained root access through a local `modutils` exploit, which triggered the host IDS alert.

We examined the other Snort alerts by hand to see if there were any other true positives. We found one other true positive, in which the attacker compromised the web server but did not gain root privilege. This illustrates the tradeoffs involved in correlating IDS alerts. Reporting only correlating alerts reduces the number of false positives, but it may mask some true positives as well. Correlating Snort alerts with a more sophisticated host IDS may have reported both true positives.

Overall, BDB effectively reduced the number of false positives on our testbed. It also reduced administrative overhead by allowing us to use Snort's default rules, rather than customizing the set of rules by hand to reduce false positives.

6. Prototype implementation and performance

BDB is a working prototype implemented for the Linux operating system. Causal tracking is implemented as two

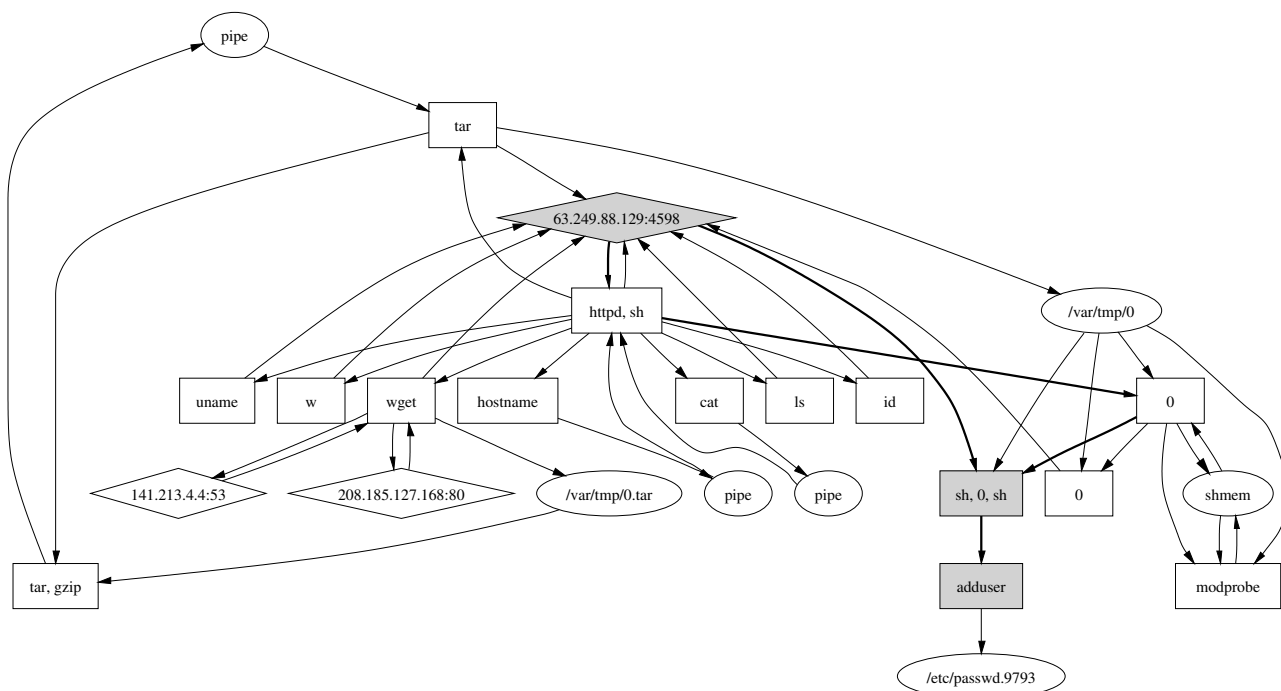


Figure 6. Correlating alerts. This figure shows how a causal graph can connect two distinct IDS alerts. The two alerts are shown as shaded boxes. One alert is generated by Snort, and the other is generated by a host IDS that detects root processes. The bold lines highlight the events that most directly relate the flagged IDS alerts.

separate modules: one for logging the specific causal events and another for generating forward and backward graphs. The logging requires modifications to the Linux operating system; the resulting logs can be written to a local file or streamed out over the network. The graph generator then parses the logs to create the graphs. BDB stores logs in a MySQL database; this allows it to generate graphs quickly even for large logs. For example, generating a graph from a log containing 2,187,963 objects and 55,894,869 events takes about 26 seconds. The time to generate graphs is proportional to the number of objects in the final causal graph rather than the total number of objects in the log. As a result, the multi-hop example discussed in Section 4.3 took less than 45 seconds to perform the complete analysis over all 12 nodes.

7. Limitations

As with most security enhancements, using causal tracking to enrich IDS alerts has its limits. First, BDB can track events only on hosts that are using its kernel modifications. Both backward and forward tracking stop when they reach a host that is not running BDB. For max-

imum effectiveness, an administrator should run BDB on all hosts in her administrative domain.

Second, BDB tracks only a subset of the many events that form dependencies, and events outside this set lead to covert channels. For example, BDB ignores the dependency that is formed if one process creates a file in a directory and another process lists the contents of that directory. [7] describes why it is difficult for an attacker to use these types of covert channels to carry out an attack without being tracked by BDB, but it is still possible. An attacker may also create a covert channel by leveraging an unmonitored host. BDB cannot track causality on hosts that lack instrumentation, and an attacker can break the causal chain by traversing such a host.

Another style of attack against BDB is to deliberately create noise in the causal graphs, either by generating a large number of events, or by implicating innocent processes and hosts. However, these actions may make it easier for an IDS to notice the attacker.

8. Related work

Other research projects have used the idea of causality to identify the effects of an intrusion. The Repairable File System [16] identifies potentially tainted files by tracking the flow of information across operating system events. While the Repairable File System performs forward causal analysis, it does not track dependencies across the network or filter the resulting graph to prioritize the likely paths of an intrusion. In the database area, work by Ammann, Jajodia, and Liu tracks the flow of contaminated transactions through a database and rolls data back if it has been affected directly or indirectly by contaminated transactions [3].

In addition to our approach of using causality, other techniques have been used to track multi-hop attacks. Work by Zhang and Paxson detects “stepping stones” used to carry interactive communication across multiple hops by seeing which packets have correlated size and timing characteristics [15]. Wang and Reeves extend this approach by manipulating the timing of packets to more easily correlated packets across multiple hops [14]. Our approach has advantages and disadvantages compared to prior approaches. The main disadvantage is that our approach requires one to monitor each host in the chain. Hence our approach is suitable mainly for communication within a single administrative domain. The main advantage of our approach is that it is independent of timing because it can track actual cause-and-effect chains, rather than relying on less robust characteristics such as timing. Hence our approach can track non-interactive multi-hop attacks, such as worms (even stealthy ones).

Many prior researchers have sought to correlate IDS alerts to reduce false positives [11] [4] [9]. Most prior projects correlate IDS alerts based on shared features, such as source or destination IP addresses of packets or the time at which the IDS alert occurred. Other projects use prior knowledge about sequences of actions to connect IDS alerts together into an attack scenario. Our work adds a new way to connect IDS alerts, by tracking actual cause-and-effect chains to connect prior alerts with later ones.

9. Conclusions and future work

Causality provides a new mechanism for enhancing IDS alerts. Instead of seeing each IDS alert in isolation, one can use causality to see which events and state led up to the IDS alert, and which events and state were influenced by the state detected by the IDS. BDB follows causal relationships both forward and backward and follows causal relationships across multiple hosts.

The mechanism of causality can be used in a variety of ways to increase the accuracy and reach of IDS alerts. We

demonstrated how causality was able to track the Slapper worm as it spread within a local network. We also showed how causality tracked a manual attack that spread via different attack vectors, and we showed how causality could be enhanced with application-specific knowledge to track an e-mail virus. Finally, we showed how causality can correlate distinct network and host IDS alerts to reduce the number of false positives.

In the future, we plan to explore how the causal graph generated by BDB can be used as a standalone host IDS. An administrator can specify or profile a set of causal graphs for each network service, then restrict the system to generate an alert if it sees a causal graph outside of this set. We also plan to study what types of causal relationship are most likely to connect related IDS alerts.

In summary, we believe that causality is a useful way to gain additional information from existing IDS alerts. By leveraging true positives found by existing an IDS, BDB can find other hosts that were also compromised and can correlate distinct alerts to form fewer, higher-confidence alerts.

10. Acknowledgments

The ideas in this paper were refined during discussions with Michael Bailey, Evan Cooke, David Watson, and Farnam Jahanian. The anonymous reviews provided valuable feedback that helped improve the quality of this paper. This research was supported in part by ARDA grant NBCHC030104, National Science Foundation grants CCR-0098229 and CCR-0219085, and by Intel Corporation. Samuel King was supported by a National Defense Science and Engineering Graduate Fellowship. Any opinions, finding, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

- [1] Snort: The Open Source Network Intrusion Detection System. <http://www.snort.org/>.
- [2] Common Vulnerabilities and Exposures CAN-2002-0656, July 2002.
- [3] P. Ammann, S. Jajodia, and P. Liu. Recovery from Malicious Transactions. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1167–1185, September 2002.
- [4] F. Cuppens and A. Mieke. Alert Correlation in a Cooperative Intrusion Detection Framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, May 2002.
- [5] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing, January 1998. RFC 2267.
- [6] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: a file system integrity checker. In

- Proceedings of 1994 ACM Conference on Computer and Communications Security (CCS)*, pages 18–29, November 1994.
- [7] S. T. King and P. M. Chen. Backtracking Intrusions. In *Proceedings of the 2003 Symposium on Operating Systems Principles (SOSP)*, pages 223–236, October 2003.
 - [8] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.
 - [9] P. Ning, Y. Cui, and D. S. Reeves. Constructing Attack Scenarios through Correlation of Intrusion Alerts. In *Proceedings of the 2002 ACM Conference on Computer and Communications Security (CCS)*, pages 245–254, November 2002.
 - [10] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *Proceedings of the 2000 ACM SIGCOMM Conference*, pages 295–306, August 2000.
 - [11] A. Valdes and K. Skinner. Probabilistic Alert Correlation. In *Proceedings of the 2001 International Workshop on the Recent Advances in Intrusion Detection (RAID)*, October 2001.
 - [12] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of 2001 IEEE Symposium on Computer Security and Privacy*, 2001.
 - [13] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the 2004 ACM SIGCOMM Conference*, August 2004.
 - [14] X. Wang and D. S. Reeves. Robust Correlation of Encrypted Attack Traffic Through Stepping Stones by Manipulation of Interpacket Delays. In *Proceedings of the 2003 ACM Conference on Computer and Communications Security (CCS)*, October 2003.
 - [15] Y. Zhang and V. Paxson. Detecting Stepping Stones. In *Proceedings of the 2000 USENIX Security Symposium*, August 2000.
 - [16] N. Zhu and T. Chiueh. Design, Implementation, and Evaluation of Repairable File Service. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN)*, pages 217–226, June 2003.

APPENDIX C



Princeton Computer Science

<http://www.cs.princeton.edu/>



First-year grad students, 2004.

Welcome to the website of the Department of Computer Science at Princeton University. Princeton has been at the forefront of computing since Alan Turing, Alonzo Church and John von Neumann were among its residents. The department has experienced significant growth over the last few years, and is now home to 30 faculty, with strong groups in theory, systems, graphics/media, programming languages, computational science, security, AI, and computational biology.

News & Events

April 6, 2006 Event:
[Lecture: Contextual Integrity: A Conservative Approach to Privacy](#)

April 5, 2006 Event:
[Lecture: The Future of the Web](#)

Today (April 4, 2006) Event:
[SEAS Departmental Open House](#)

Today (April 4, 2006) Event:
[Princeton Laptop Orchestra \(PLOrk\) Concert April 4](#)

Today (April 4, 2006) Event:
[Departmental Open House for Computer Science and Applications of Computing](#)

March 29, 2006 (posted date)
[Who's afraid of Alex Halderman?](#)

[more news...](#)

Colloquia

Wed, April 5, 2006
Colloquium: [Stefan Roth](#)
Brown University

Mon, April 10, 2006
Colloquium: [Sam King](#)
University of Michigan

[RSS](#)

[full colloquia schedule...](#)

Colloquia and Seminar Schedule

[View Current Schedule] [Browse the archives] [Sign up Speakers]

All colloquia are in the Computer Science Building unless noted.

Monday, April 10, 2006, 4 :30pm

Colloquium

Small Auditorium (Room
Host: Larry Pet

Analyzing Intrusions Using Operating System Level Information Flow

Sam King, University of Michigan

Abstract:

Computers continue to get broken into, so intrusion analysis is a part of most system administrators'job description. System administrators must answer two main questions when analyzing intrusions: "how did the attacker gain access to my system?", and "what did the attacker do after they broke in". Current tools for analyzing intrusions fall short because they have insufficient information to fully track the intrusion and because they cannot separate the actions of attackers from the actions of legitimate users.

This talk will focus on how system administrators can use information flow graphs to help analyze intrusions. BackTracker is used to help answer the question "how did the attacker gain access to my system?". BackTracker starts with a suspicious object (e.g., malicious process, trojaned executable file) and follows the attack back in time, using causal OS events, to highlight the sequence of events and objects that lead to the suspicious state. Showing an information flow graph of these causally connected events and objects provides a system wide view of the attack and significantly reduces the amount of data an administrator must examine in order to determine which application was originally exploited. ForwardTracker helps answer the question "what did the attacker do after they broke in?". ForwardTracker starts from the application which was exploited and tracks causal events forward in time to display the information flow graph of events and objects that result from the intrusion. Finally, Bi-directional distributed BackTracker (BDB) continues the backward and forward information flow graphs across the network to highlight the set of computers on a local network which are likely to have been compromised by the attacker.

web.archive.org/web/20060908233057/http://www.cs.cornell.edu/colloquium/2006sp/

Internet Archive Wayback Machine 47 captures 18 Jun 2006 – 19 Jan 2025

Go JUN SEP DEC 08 2005 2006 2008 About this capture

Computer Science Colloquium Series, Spring 2006

Regular time and location: Thursdays at 4:15pm in B17 Upson unless otherwise indicated.

Date	Speaker	Title	Host
1/26/06	Lee Giles Pennsylvania State University	Next Generation CiteSeer: CiteSeer^x Co-sponsored by Information Science	Halpern
Friday 1/27/06	Linda Petzold UC Santa Barbara	Multiscale Simulation of Biochemical Systems ** 4:30 PM 5130 Upson Hall **	Van Loan
Tuesday 1/31/06	Ioana Dumitriu UC Berkeley	Toward Accurate Polynomial Evaluation in Rounded Arithmetic	Van Loan
2/2/06	Raghu Ramakrishnan University of Wisconsin-Madison	Salton Lecture Series Discovering Interesting Subsets of Data in Cube Space	Gehrke
2/9/06	Dan Suciu University of Washington	Managing Imprecisions with Probabilistic Databases	Gehrke
2/16/06	Raffaello D'Andrea Cornell University	Design and Control of Autonomous Systems	
Tuesday 2/21/06	Yael Tauman Kalai MIT	Limits of Obfuscation	Tardos
3/2/06	Utkarsh Srivastava Stanford University	Efficient Query Processing for Modern Data Management	Gehrke
Monday 3/6/06	Chris Bischof UT Aachen	Automatic Differentiation and Beyond ** 4:15 PM 655 Rhodes Hall **	Van Loan
3/9/06	Sam King University of Michigan	Analyzing Intrusions Using Operating System Level Information Flow	Schneider
3/16/06	Boon Loo UC Berkeley	Declarative Networking: Extensible Networks with Declarative Queries	Sirer
3/21/06 & 3/23/06 Spring Break			
Monday 3/27/06	Itzik Pe'er MIT	Data, technology and populations for genomewide association studies Co-Sponsored by Biological Statistics and Computational Biology	Elber

Computer Science Colloquium x +

web.archive.org/web/20060910211057/https://www.cs.cornell.edu/colloquium/2006sp/king.htm

INTERNET ARCHIVE
waybackmachine

4 captures
10 Sep 2006 - 22 Sep 2015

Go AUG SEP AUG
2005 10 2006 2008 About this

Thursday, March 9, 2006
4:15 pm
B17 Upson Hall

Computer Science
Colloquium
Spring 2006

Sam King University of Michigan

Analyzing Intrusions Using Operating System Level Information Flow

Computers continue to get broken into, so intrusion analysis is a part of most system administrators' job description. System administrators must answer two main questions when analyzing intrusions: "how did the attacker gain access to my system?", and "what did the attacker do after they broke in?". Current tools for analyzing intrusions fall short because they have insufficient information to fully track the intrusion and because they cannot separate the actions of attackers from the actions of legitimate users.

This talk will focus on how system administrators can use information flow graphs to help analyze intrusions. BackTracker is used to help answer the question "how did the attacker gain access to my system?". BackTracker starts with a suspicious object (e.g., malicious process, trojaned executable file) and follows the attack back in time, using causal OS events, to highlight the sequence of events and objects that lead to the suspicious state. Showing an information flow graph of these causally-connected events and objects provides a system-wide view of the attack and significantly reduces the amount of data an administrator must examine in order to determine which application was originally exploited. ForwardTracker helps answer the question "what did the attacker do after they broke in?". ForwardTracker starts from the application which was exploited and tracks causal events forward in time to display the information flow graph of events and objects that result from the intrusion. Finally, Bi-directional Distributed BackTracker (BDB) continues the backward and forward information flow graphs across the network to highlight the set of computers on a local network which are likely to have been compromised by the attacker.

Bio: Sam King is a PhD student in the CSE Division of the Department of Electrical Engineering and Computer Science at the University of Michigan at Ann Arbor and will be graduating in the summer of 2006. His research interests include experimental software systems, computer security, and operating systems. His dissertation work focuses on computer forensics, and he has also explored various other topics while at Michigan including advanced malware defenses, intrusion detection, and debugging operating systems using time-traveling virtual machines. Before arriving at Michigan, he received a BS from UCLA and an MS from Stanford University. He also worked for two years as a software developer for an embedded systems company in Santa Clara, CA. He is married (his wife's name is also Sam) and has a son, Eli.

APPENDIX D

Peter Chen's Publications

web.archive.org/web/20070214125111/http://www.eecs.umich.edu/virtual/papers/index.html

Internet Archive Wayback Machine

17 captures

15 Jul 2006 - 28 Jan 2012

Go

JUL FEB SEP

2006 14 2007 2008

About this capture

2006

- Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, "[Rethink the sync](#)", *Proceedings of the 2006 Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006 ([presentation](#) by Ed Nightingale). Best Paper Award.
- George W. Dunlap, "[Execution Replay for Intrusion Analysis](#)", October 2006. This is George Dunlap's Ph.D. thesis.
- Samuel T. King, "[Analyzing Intrusions Using Operating System Level Information Flow](#)", September 2006. This is Sam King's Ph.D. thesis.
- Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, Jacob R. Lorch, "[SubVirt: Implementing malware with virtual machines](#)", *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006 ([presentation](#) by Sam King).

2005

- Ashlesha Joshi, Samuel T. King, George W. Dunlap, Peter M. Chen, "[Detecting past and present intrusions through vulnerability-specific predicates](#)", *Proceedings of the 2005 Symposium on Operating Systems Principles (SOSP)*, October 2005 ([presentation](#) by Ashlesha Joshi).
- Edmund B. Nightingale, Peter M. Chen, Jason Flinn, "[Speculative Execution in a Distributed File System](#)", *Proceedings of the 2005 Symposium on Operating Systems Principles (SOSP)*, October 2005 ([presentation](#) by Ed Nightingale). Award paper.
- Dominic Lucchetti, Steven K. Reinhardt, Peter M. Chen, "[ExtraVirt: Detecting and recovering from transient processor faults](#)", *2005 Symposium on Operating Systems Principles (SOSP) work-in-progress session*, October 2005 ([presentation](#) by Dominic Lucchetti).
- Samuel T. King, George W. Dunlap, Peter M. Chen, "[Debugging operating systems with time-traveling virtual machines](#)", *Proceedings of the 2005 Annual USENIX Technical Conference*, April 2005 ([presentation](#) by Sam King). Best Paper Award.
- Samuel T. King, Peter M. Chen, "[Backtracking Intrusions](#)", *ACM Transactions on Computer Systems*, February 2005. This is the journal version of our SOSP 2003 paper.
- Samuel T. King, Z. Morley Mao, Dominic G. Lucchetti, Peter M. Chen, "[Enriching intrusion alerts through multi-host causality](#)", *Proceedings of the 2005 Network and Distributed System Security Symposium (NDSS)*, February 2005 ([presentation](#) by Sam King).

2004

- Ashlesha Joshi, Peter M. Chen, "[The ghost of intrusions past](#)", *2004 Symposium on Operating Systems Design and Implementation (OSDI) work-in-progress session*, December 2004 ([presentation](#) by Ashlesha Joshi).
- Murtaza Basrai, Peter M. Chen, "[Cooperative ReVirt: Adapting Message Logging for Intrusion Analysis](#)", *University of Michigan CSE-TR-504-04*, November 2004.
- Peter M. Chen, "[An Automated Feedback System for Computer Organization Projects](#)", *IEEE Transactions on Education*, May 2004.

2003

- Samuel T. King, Peter M. Chen, "[Backtracking Intrusions](#)", *Proceedings of the 2003 Symposium on Operating Systems Principles (SOSP)*, October 2003 ([presentation](#) by Sam King). Award paper.
- Samuel T. King, George W. Dunlap, Peter M. Chen, "[Operating System Support for Virtual Machines](#)", *Proceedings of the 2003 Annual USENIX Technical Conference*, June 2003 ([presentation](#) by Sam King).

2002

- George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, Peter M. Chen, "[ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay](#)", *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002 ([presentation](#) by George Dunlap).
- Subhachandra Chandra, Peter M. Chen, "[The Impact of Recovery Mechanisms on the Likelihood of Saving Corrupted State](#)", *Proceedings of the 2002 International Symposium on Software Reliability Engineering (ISSRE)*, November 2002 ([presentation](#) by Subhachandra Chandra).

2001

APPENDIX E

BorderPatrol: Isolating Events for Black-box Tracing

Eric Koskinen
Department of Computer Science
Brown University
115 Waterman St., Providence,
Rhode Island 02912, USA
ejk@cs.brown.edu

John Jannotti
Department of Computer Science
Brown University
115 Waterman St., Providence,
Rhode Island 02912, USA
jj@cs.brown.edu

ABSTRACT

Causal request traces are valuable to developers of large concurrent and distributed applications, yet difficult to obtain. Traces show how a request is processed, and can be analyzed by tools to detect performance or correctness errors and anomalous behavior.

We present BorderPatrol, which obtains precise request traces through systems built from a litany of unmodified modules. Traced components include Apache, tthttpd, PostgreSQL, TurboGears, BIND and notably Zeus, a closed-source event-driven web server. BorderPatrol obtains traces using *active observation* which carefully modifies the event stream observed by modules, simplifying precise observation. *Protocol processors* leverage knowledge about standard protocols, avoiding application-specific instrumentation.

BorderPatrol obtains precise traces for black-box systems that cannot be traced by any other technique. We confirm the accuracy of BorderPatrol's traces by comparing to manual instrumentation, and compare the developer effort required for each kind of trace. BorderPatrol imposes limited overhead on real systems (approximately 10-15%) and it may be enabled or disabled in at run-time, making it a viable option for deployment in production environments.

Categories and Subject Descriptors

C.4 [Performance of Systems]; D.2.5 [Software Engineering]: Testing and Debugging—Distributed debugging, testing tools, tracing; K.6.1 [Project and People Management]: Systems analysis and design; K.6.4 [System Management]

General Terms

Performance, Measurement

Keywords

Performance debugging, black box systems, distributed systems, performance analysis, causal paths

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EuroSys'08, April 1–4, 2008, Glasgow, Scotland, UK.

Copyright 2008 ACM 978-1-60558-013-5/08/04 ...\$5.00.

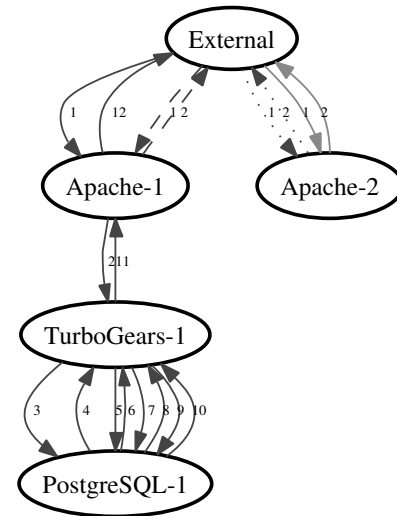


Figure 1: A BorderPatrol trace of a three-tier web application. Four external requests (each shown with a different arrow type) are handled by Apache in two processes. The first request, shown in solid black, is passed on to a web-application written in Python which makes several database calls to fulfill the request. The programs were not modified to obtain the trace.

1. INTRODUCTION

Today's large-scale applications consist of many independent modules that leverage concurrency for performance. In many cases, the components are developed by different groups and in different languages. Individual components may exploit concurrency with threaded, multi-process, or event-driven designs.

Regardless of this heterogeneity, developers want answers to questions about their entire applications. "What path through the system do search requests take, and where do they spend the most time?" or "What resources are used by clients reading email, as compared to sending email?" The principals of interest in these queries are requests, not individual modules. Developers would benefit from tracing tools that follow single requests as they are passed between modules, including third-party binary modules, even as those requests are passed and returned from remote, unmodified systems. Figure 1 shows a trace produced by BorderPatrol, our tool for tracing unmodified systems of black-boxes.

Beyond being used as an aid to developers, recent work has shown that request traces can be valuable input to au-

tomated tools. Systems such as Pinpoint [7], Pip [13], and Stardust [19] analyze precise request traces to identify faulty modules, discover anomalous request paths, and make capacity plans. However, these systems rely on simple trace gathering techniques that are unable to trace complex systems.

Obtaining precise request traces in a heterogeneous, concurrent system is difficult. It is insufficient to obtain traditional trace data, such as system call or function call logs, since these logs do not indicate when high-level requests have been handed off from module to module. Instead, most tracing systems have advocated module-specific programmer-supplied instrumentation which requires source-level access and an understanding of the module architecture. While some [1] have avoided instrumentation, they have sacrificed precise traces for statistical information.

When an application spans multiple modules, or when a module multiplexes several requests, request flow does not follow module control flow. Instead requests are executed in fragments by modules that multiplex their time among many such requests. These modules may use operating system abstractions such as processes or threads, or they may manage concurrency themselves, using an event loop or user-level threading package.

BorderPatrol follows requests as they move through this cacophony of modules, written by disparate teams, loosely aggregated with protocols that do not share a unifying request abstraction. We understand, before we begin, that perfection is impossible. In the general case, precise black-box tracing is impossible, since modules may act in arbitrary ways inside their “black-boxes” particularly when presented with simultaneous requests. However, our observation is that real applications are not arbitrary. Through careful observation and a light-weight form of module isolation, causal paths can be reconstructed in real-world systems.

Previous tracing systems have also recognized the impossibility of precise tracing without programmer assistance. Some have required programmer assistance in the form of a “bread crumb” trail emitted by instrumentation, or by stashing request identifiers inside of datastructures that are carried throughout a framework. Neither solution is satisfying in a heterogeneous environment. Others have sacrificed precision in favor of statistical results. In these systems, common paths through unmodified modules can be found with some probability, but precise traces of specific anomalous paths cannot be determined. We discuss all of these approaches in Section 8.

Our compromise, and therefore our contribution, is different. We present a tracing technique that *actively isolates black-box inputs* so that request paths can be precisely observed, without materially affecting the overall application’s ability to multiplex requests. *Event isolation* (Section 3.3) unbundles concurrent input events in order to allow the observation of a module’s behavior on a per-event basis. When event isolation is impossible or undesirable, we identify request propagation by inspection. *Message witnesses* (Section 3.2) identify matched messages, usually request/response pairs. Event isolation and message witnesses are provided by *protocol processors* (Section 3.1), an abstraction that allows developers to implement protocol-specific, rather than implementation-specific, tracing. A single HTTP protocol processor can be used to trace various web servers, web proxies, or even XML-HTTP services.

The techniques described in this paper are realized in a tool called BorderPatrol, which is publicly available. BorderPatrol uses library interposition to insert protocol processors between the unmodified black-boxes of a multi-module system. For BorderPatrol to be effective, the black-boxes must follow certain assumptions that we outline in the following section. We argue that these assumptions are reasonable because they follow from common software architectures.

Our evaluation consists of case studies (Section 6) and a performance evaluation (Section 7). We show that BorderPatrol reconstructs causal paths through a range of diverse servers including Apache, thttpd, Zeus, BIND, PostgreSQL, and TurboGears, without modifications to server source code or the use of statistical methods. Further, we show that the overhead of tracing is about 10-15% and can be activated at runtime, making it a viable technique for production environments.

2. BLACK BOX MODEL

BorderPatrol seeks to follow the repeated transfer of a request from one black-box module to another in order to construct causal paths that show which modules handled a given request, in what order, and for how long. For example, when a web application queries a database, we want to associate the computation in the database with the original HTTP request. Although BorderPatrol treats modules as “black boxes,” it makes some assumptions about the way real-world applications work that allow it to follow request transfers.

Request traces can be thought of as chains that are made up of two types of links. *External* links connect the output of one black-box module to the input of another. *Internal* links connect a module input to a module output.

External links can be observed by monitoring communication channels using any number of techniques, *i.e.* network snooping, virtual machine monitoring, and system or library call interposition.

The internal links of black boxes cannot be observed directly. Instead, BorderPatrol makes inferences based on the assumption that the internals of black boxes are *honest*, *immediate*, and *independent*. A black box is honest if it faithfully implements the basic structure of the protocols it participates in. It is immediate if, when presented with a single input event, it processes the input event before requesting another input. Finally, black boxes are independent if they process concurrent input events in the same way that they would have processed the events if they arrived sequentially, except for timing effects. The remainder of this section details these assumptions, and describes why we expect that the operation of real-world black boxes operate within them.

2.1 Honesty

Sometimes, internal links can be established by observing the contents of input and output messages. This is common when a request is passed out of a module using the same protocol that passed the request in, so that an identifier is visible in both messages, for example in the nonce of an RPC request/reply, or the URL in a proxy server request that is forwarded to the origin server. We refer to these identifiers as *witnesses* and BorderPatrol assumes they are accurate if they exist. We expect that bugs affecting witnesses are so basic that they are unlikely in systems mature enough to

consider tracing. Witnesses are used only to patch paths when BorderPatrol’s request following techniques cannot be used, such as when building a path through a remote, untraced module.

In effect, a “dishonest” module is violating its specification so badly that it’s hard to define what BorderPatrol *should* do. If a module changes the identifiers associated with a request in an unrecognizable violation of the protocol, is it really forwarding the original request? BorderPatrol does *not* trace black-box systems, it traces systems made up of black-boxes. The definition of a black-box is an unknown module that adheres to a specification, and honesty requires that adherence.

2.2 Immediacy

Usually, determining internal links is more difficult than matching witnesses. An internal link matches a module’s input messages to its output messages. However, the protocols used may be unrelated and the follow-on messages may carry no identifying information that can be tied to the original request. For example, a web page request arrives as an HTTP request, but the application server may issue SQL requests to the database server that bear no resemblance to the original HTTP request. No witness oriented approach could produce internal links for such modules.

In addition, modules multiplex requests. Tracing cannot be accomplished simply by maintaining a map between requests and the module (or even the thread within the module) which will be handling it, then associating any further communication with that request. For example, event-driven systems rotate among outstanding requests using a single thread. Additionally, a single process may collect multiple inputs (via `read` for example), and then work on them consecutively with no externally identifiable break between them.

Our model assumes that black boxes are immediate—they are composed of pieces we refer to as *fragments* that do *not* multiplex requests. A fragment is an internal control path that handles individual inputs and processes them to completion. These fragments usually do not process an entire request. The execution of the fragment runs from one input event (such as data becoming available on a socket) to another, not from request start to finish.

Since fragments immediately begin work on the request associated with their input event, BorderPatrol can determine internal links by supplying that input, noting the output caused by the fragment, and connecting the two. BorderPatrol takes a general view of *output* that includes any interaction with an outside module, such as connection creation. We describe this process in detail in Section 5.5.

Immediacy is illustrated in Figure 2. On the left, two concurrent requests enter a black-box module, and since nothing is known about the module internals, it is impossible to match the inputs to the outputs. However, the right side of the diagram illustrates the module’s true structure. Although BorderPatrol cannot determine this structure, it is easy to see that if the events are supplied independently, the output can be matched with the input.

2.3 Independence

Our final assumption about black-box modules is independence. We assume that modules will respond to two sequential inputs the same way they would have treated those

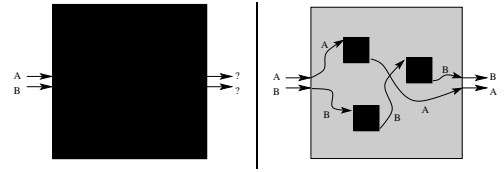


Figure 2: An illustration of the *immediacy* and *independence* assumptions. Immediacy tells us that when A is supplied, the black box’s next observable action will be to create the output labeled A. Independence tells us that the black box would not have treated A differently had it been supplied simultaneously with B.

inputs if they had been received simultaneously. In a concrete example, we assume that an application that uses `poll` will not behave differently if it must call `poll` twice to obtain two ready file descriptors.

Like immediacy, the independence assumption says nothing of entire requests, only the individual *events* that comprise them. We believe that inside real-world modules, multiple input events are immediately separated by the structure of well-written software. Libraries such as `libevent` [10] and `libasync` [11], dispatch to independent event handlers for each event. If multiple events are supplied to an event-driven module, the internal event loop dispatches these events serially to event handlers. In threaded applications, independence is even clearer. Each thread in these applications blocks waiting for a single next event (such as the completion of a read) to proceed. There is no danger that the behavior of these applications will change when event arrival is serialized.

Although batch-oriented interfaces are a common performance optimization, we explain in Section 4 how BorderPatrol is able to take advantage of independence without foiling these optimizations. Briefly, the batch interface is still used across the slow kernel-user boundary, but individual events are supplied from BorderPatrol’s interposition layer to the original module.

Independence does not imply that modules will function internally in an identical fashion when events are reordered. For example, a server that uses caching techniques will respond with an alternate code path when the cache is populated by an earlier request. BorderPatrol will faithfully follow the new path. The independence assumption says that the alternative path is *correct*, not identical to what would have occurred without BorderPatrol. This change means that BorderPatrol is not the optimal debugging tool for finding errors due to race conditions or locking mistakes that require fine-grained replay of identical traces. However, correct implementations of black-box specifications will exhibit independence, and remain correct while running under BorderPatrol.

3. ACTIVE OBSERVATION

BorderPatrol employs *active observation* to observe and subtly modify the event streams sent and received by monitored modules. Active observation, allows BorderPatrol to precisely trace modules that follow the assumptions of the previous section without materially affecting module behavior.

Protocol processors implement active observation in a modular way, encoding protocol knowledge, not implementation

knowledge. Protocol processors are used to understand and separate multiple messages on a single channel. These processors also record witnesses that allow path reconstruction when external modules are used that are not traced by BorderPatrol, and protocol specify attributes that may be interesting to distinguish requests for user queries.

3.1 Protocol Processors

BorderPatrol uses library interposition to pass input to protocol processors before passing it on to unmodified modules. The protocol processors identify message boundaries, log protocol-specific attributes that users may wish to query, and track message witnesses (see below). Although the development of protocol processors requires more specialized knowledge than pure black-box approaches, the knowledge is not application-specific, but protocol-specific. We have used the same HTTP protocol processor to trace many different web servers with wildly varying implementations. Furthermore, these protocol processors do not fully implement the protocol, they usually understand little beyond the basic “envelope” of the protocol messages.

Protocol processors operate by looking for message delimiters or length counts in the data stream, and understand messages only enough to log application-specific identifiers such as URLs, SQL queries, or sequence numbers. The interface from the interposition library to the protocol processor has been designed to make these tasks easy. As a result, the protocol processors we have implemented are between 30-150 lines of code. We discuss the details of the protocol processor interface and implementation in Section 5.2.

3.2 Message Witnesses

Message witnesses are the simplest way to establish the internal links from module inputs to outputs. A *message witness* is data that can be extracted from input messages and output messages to allow direct linking. Unfortunately, witnesses are unlikely when input and output messages are of different protocols, so they are useful mainly for linking requests and replies, particularly for requests into modules that cannot be traced by BorderPatrol, such as remote web services.

Protocol processors find and log message witnesses such as URLs and RPC identifiers. When these witnesses reappear in responses, BorderPatrol can stitch together the request trace that might have otherwise been lost when the request was being handled by a remote module.

3.3 Event Isolation

In order to directly follow internal links without witnesses, BorderPatrol supplies input events to modules one at a time. BorderPatrol then monitors the module’s output, and assumes (due to immediacy) that any activity can be attributed to the same request as the input event. As a concrete example consider `poll`, an interface that modules use to obtain events for any number of file descriptors. At the time `poll` is called, BorderPatrol has tracked the input that is available on each channel, and can attribute each potential input to a high-level request. By returning only one event at a time to the black-box module, BorderPatrol can attribute the following fragment of computation and outputs to the request associated with the event.

The events returned by `poll` are indivisible, they can be attributed to only one request. However, when a module

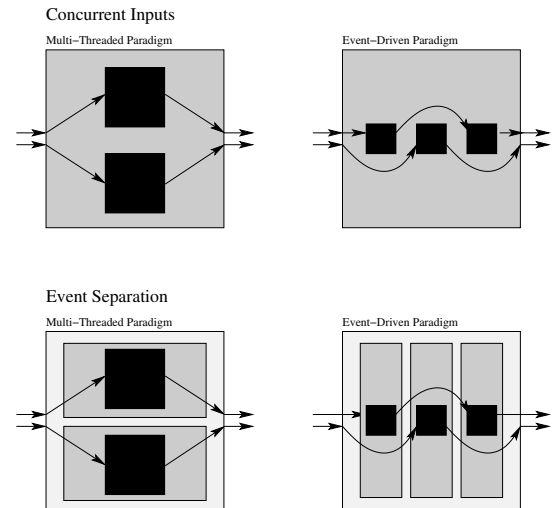


Figure 3: BorderPatrol works because real-world servers have straightforward internal structure. Multi-threaded servers dispatch events independently, to separate threads (left). Event-driven servers execute in fragments that can be pieced together by running them sequentially (right).

reads input data, there is the danger that input from several messages, and therefore several requests, is combined. Protocol processors allow BorderPatrol to isolate events at the protocol level, preventing reads containing multiple messages.

When data arrives on a channel with a protocol processor, the data may contain multiple input events that should be isolated. The protocol processor examines the data in search of message boundaries. If a boundary is found, only the data leading up to the boundary is returned. On the next read, the processor considers the cached data first, though it may find yet another boundary, and again return a “short read.”

4. WHY DOES BORDERPATROL WORK?

Do real-world applications follow our assumptions and decompose cleanly to code fragments that operate on individual events? Can BorderPatrol obtain that decomposition? This section explores typical application architectures and explains when and why applications can be decomposed and traced accurately.

Fundamental to real-world interactive programs, of which servers are a subset, is the ability to handle concurrent requests. Therefore, these applications must be able to accept new requests continuously, even as previous requests are still being processed. There are several common paradigms for multiplexing requests. Using the taxonomy presented by Pai *et al.* [12], we consider some of the most popular.

Multi-process or Multi-threaded. Servers written in the MP/MT style maintain a pool of individual threads (or processes). These threads loop, continuously accepting new requests, processing each one to completion. In pseudo-code:

```
while (fd = accept())
  while (req = read(fd))
    handle_request(req);
  close(fd);
```

Such a server is immediate and independent. While inside `handle_request`, the server will service only a single

request. It may interact with additional modules to aid in servicing the request, but BorderPatrol's tracing job is easy. For example, the request might be an HTTP request for a page containing user customized data obtained via an RPC interface. BorderPatrol attributes the RPC to the top-level request, and continues path reconstruction in the destination module. If the destination module is not running BorderPatrol, a witness in the RPC response can reestablish the request path, treating the entire remote module as a single black box. BorderPatrol does *not* simply assume that sequential behavior across the RPC call, in two separate fragments, is related. BorderPatrol follows the request back into the web server from the remote module. The link is established by active observation, not by assumptions about single-threaded code.

Single Process Event-driven. SPED servers multiplex requests across a single thread. In pseudo-code:

```
while(1)
  events = poll();
  for e in events
    handler = find_handler(e);
    execute(handler, e);
```

The handling of a single request is divided into many smaller stages. The equivalent of `handle_request()` might consist of several handlers: (1) parse the request and initiate a connection to the RPC server (2) complete the connection to the RPC server (3) write a message to the RPC server (4) read the response from the RPC server and (5) compute the HTML response and write it to the client. Further, each of these stages might re-register the same handler to complete a lengthy operation.

BorderPatrol ensures that the SPED process receives only one event at a time, so all of the following actions, until the next input, can be attributed to the input event's request. An illustration of this architecture appears on the right-hand side of Figure 3.

Asymmetric Multi-Process Event-Driven AMPED servers are similar to SPED servers, with the addition of helper processes used to simulate asynchronous I/O. BorderPatrol observes the requests from the main process and attributes the work of the helper to the high-level request that initiated contact with the helper. BorderPatrol will require a protocol processor in the case that the communication between the main process and helper persists on a single channel. Simpler interactions with subprocesses that span a single request can be handled by BorderPatrol's default "One-shot" protocol processor. We expect that these ad-hoc protocols are conveniently delimited or use fixed frame sizes. We expect implementations in the 10s of lines.

Work Queues. Applications that make use of "hidden" work queues to pass requests from module to module will present a problem for BorderPatrol's tracing because of unobservable fragment interactions. Work queues may be implemented with internal data structures that cannot be observed without more invasive techniques. However, some work queue implementations do have standardized interfaces, and if they are implemented as shared libraries or via IPC, fragment interactions might be observed by an "API processor" akin to BorderPatrol's protocol processors. Regardless we were somewhat surprised, but pleased, not to find this model in the many modules we examined.

Whodunit [6] aims to derive information from (nearly) unmodified servers that pass requests in this manner. There

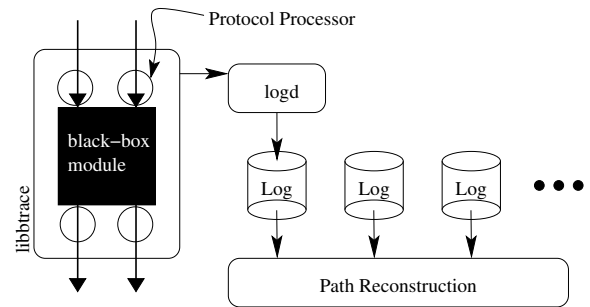


Figure 4: System Overview. The solid black box represents a traced application module. Communications (messages, IPC, and signals) are monitored by the protocol and/or API processors. Events are relayed through the Logging Daemon to a raw database. Databases from multiple hosts may be aggregated, and causal paths are reconstructed.

is potential synergy with BorderPatrol's mechanism, though Whodunit's output is statistical making it inappropriate for use in precise tracing without further adaptation.

User-level Scheduling. User-level threads may also present difficulties for BorderPatrol, depending on implementation. Cooperative thread packages context switch only when the current thread attempts a blocking system call. A non-blocking version is substituted, and the thread context is switched. The thread may be resumed when an OS notification indicates the operation would not block. This architecture is identical to SPED for BorderPatrol's purposes and presents no difficulties. However, other thread packages use asynchronous signals in order to support preemption at regular intervals. BorderPatrol does not currently support the interception of these signals. Even if it did, these signals are periodic and indicate only that time has passed. Treating the package as a black box would prevent BorderPatrol from understanding the user-level scheduling code and knowing which thread has been activated. BorderPatrol could resume tracing when an interaction with a known resource is observed.

5. IMPLEMENTATION

BorderPatrol collects trace data from unmodified modules through the extensive use of library interposition which can be supplemented with data from a kernel module. The trace data is aggregated and processed in one forward pass to obtain request paths. An overview of the architecture is given in Figure 4.

5.1 Library Interposition (libtrace)

BorderPatrol's interception framework is a series of wrapper functions for roughly 20 standard library functions. Our library of wrappers, `libtrace`, is pre-loaded before system libraries using the `LD_PRELOAD` mechanism. `Libtrace` intercepts calls to `libc`, isolates events, invokes protocol processors, and emits logging events. Often, an intercepted function invokes the real `libc` routine as part of its work.

`Libtrace` tracks the requests associated with each connection in a process. BorderPatrol tracks all connection creation operations (`open`, `socket`, `pipe`, *etc.*) and modification operations (`close`, `dup`, `fcntl`).

`Libtrace` also tracks messages as they flow through read and write operations. Many connections do not need to be

monitored since request causality doesn't flow across them. For example, the work involved in opening a file should be attributed to the current request, but the request typically doesn't flow into the file (although we believe BorderPatrol could be enhanced to track architectures such as mail servers in which requests pass through the file system). For simple file operations, BorderPatrol simply logs the interaction.

By contrast, requests *do* flow over many connections in a distributed system (*e.g.* FastCGI connections and database connections). In these cases, BorderPatrol (a) identifies the protocol involved (b) invokes the protocol processor on **read/write** operations on the connection, and (c) buffers data and events when event isolation requires it.

Currently, BorderPatrol does not trace some interfaces that it ought to in order to gain the most comprehensive coverage. For example, signals, the kevent API, and the AIO system calls are all ignored. These interfaces do not appear to pose fundamental challenges, they have been neglected thus far only because of their rarity.

5.2 Protocol Processors

Libbtrace also contains the protocol-specific code that implements protocol processors. These processors provide tracing for any application that implements the protocol, regardless of architecture. While we envision the possibility of dynamically selecting the appropriate protocol processor by observing passing data, BorderPatrol currently selects the appropriate processor based on conventions such as port number, Unix domain path, or executable name.

The protocol processor interface consists of four functions, two of which are used for initialization and tear-down. The following descriptions use pseudo-code data types to elide the details of C typing and buffer handling.

pp_state pp_init() Processors allocate and initialize a structure to store protocol specific state for a given communication channel in between invocations of the processor. The allocated state is passed as the first argument to all other functions.

void pp_shutdown(pp_state) When an application closes a communication channel, processors are called to allow the deallocation of resources obtained in **pp_init**.

int pp_read(pp_state, buffer) When data arrives on an input channel, **pp_read** is invoked to log and demarcate requests. The processor returns the number of bytes from the buffer that may be passed safely to the application without crossing a protocol message boundary. If the border between two requests is found in the buffer, the processor returns the offset of the boundary. If there is no message boundary, the entire buffer may be passed through to the application even if the buffer only represents a partial message. However, for the convenience of protocol processors, BorderPatrol can buffer partial messages in order to supply **pp_read** with the complete message when more data arrives. The processor indicates the desire for buffering by returning **PP_NEED_MORE**.

int pp_write(pp_state, buffer) When data is being written to an output channel, **pp_write** demarcates and logs, just as **pp_read**. However, when BorderPatrol writes data, there is no need to perform event isolation. The protocol processor is invoked only to log events and witnesses.

Two protocol processors must be written for most protocols. The read and write functions are used to process the messages for a protocol in one direction. The write func-

```
int pp_http_read(pp_http_t state, buffer buf) {
    switch(state->s) {
        case DONE_1_0:
            return buf.length();

        case AWAIT_HEADER:
            i = find_re(buf, "GET.*?HTTP/1.1\r\n.*?\r\n\r\n")
            if (i==0) return PP_NEED_MORE;

            url = extract_url(buf)
            httpv = extract_version(buf)
            log(http_req, url, state->seq++)
            state->s = httpv == 1.1 ? AWAIT_HEADER : DONE_1_0;
            return i
    }
}
```

Figure 5: Example protocol processor for client to server communication using HTTP. **pp_http_read** illustrates an HTTP protocol processor for client to server communication. Due to the simplified interface, **pp_http_read** always operates from the start of the message.

Protocol Processor	Lines of Code
HTTP (1.0 & 1.1)	105
FastCGI	118
PostgreSQL	147
X11 (client-side only)	50
DNS (client-side only)	27
One-shot	28
Line-oriented	37

Figure 6: Protocol processor line counts. Each count includes *both* the client- and server-side of the protocol, except where noted. “One-shot” is used to handle any protocol with one request/response per connection. “Line-oriented” handles any protocol that uses newline to delimit sequential messages.

tion is invoked at the sender, and read at the receiver, but they perform nearly the same work, except for a difference in logging a receive or send. To process a protocol in both directions, a second protocol processor is used that understands the format of response messages.

An example **pp_read** for HTTP is shown in Figure 5. HTTP is a simple, sequential protocol in which each request is separated by two pairs of line-feed/newline characters. This example is organized in a state transition style. The **DONE_1_0** state only applies to HTTP/1.0 clients. Once a request header is received, the protocol processor enters this final state since HTTP/1.0 forbids reusing a connection for multiple requests. In the alternate state **AWAIT_HEADER**, the processor looks for the request separator. If it isn't found in the current data, it returns **PP_NEED_MORE**, indicating that the processor should be invoked again when more data has arrived. While the partial request is cached for the benefit of the processor, BorderPatrol also passes it through to the application because there is no danger that the partial request contains a request boundary. Finally, when the complete request is recognized, attributes are parsed from the header and logged.

BorderPatrol's actual HTTP processor is somewhat more complex, but not much, weighing in at 105 lines total for all functions. Figure 6 shows line counts for several other protocol processors, each less than 150 lines long.

BorderPatrol tracks data flowing through each file descrip-

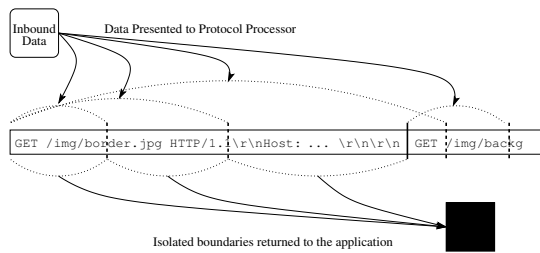


Figure 7: Data may arrive in chunks (dotted lines) on a multiplexed channel without regard to message boundaries. Protocol processors ensure that this data is supplied to applications in chunks that respect message boundaries (solid line), allowing BorderPatrol to track the behavior of applications from the point that a message has been provided, without confusion from further additional messages.

tor and maintains cursors to indicate which portions have been sent to the application, presented to the protocol processor, and not yet considered. Figure 7 demonstrates this tracking for HTTP.

If there is any data that the application has not yet collected, as much data as possible is passed to the application, considering the application's buffer size and the position of the protocol processor. Otherwise, there may be additional data, collected during a previous `read`, that the protocol processor has not yet seen. This happens whenever a protocol processor consumes a partial message or one of two contiguous messages. Finally, in the event that the protocol processor has been presented with all data in the buffer (even if it contains a portion of the next protocol chunk) and there is no cached data that can be passed to the application, the real version of `read` is used to refill the internal buffer.

Using the protocol processor on *out-bound* data is far simpler. Application `writes` are never shortened. Instead, the protocol processor is called repeatedly until all messages in the stream have been identified and logged. The remaining data is buffered until the next time `write` is invoked.

When BorderPatrol retains data in order to perform event isolation, it must also modify the result of any future call to `poll`. The buffered file descriptor is labeled readable regardless of its actual condition. In this way the application will call `read` again, which can be fulfilled from the buffer.

5.3 Kernel Page fault Monitor

User-level library interception is insufficient for capturing entry and exit from some kernel-related processes. BorderPatrol can install monitoring points in the Linux kernel in order to observe page-fault activity. Some processes use `mmap` to allow the operating system to page in data on demand without an explicit call to `read`. Library interposition cannot be used to observe I/O that results from page faults on `mmap`-ed pages, because page faults cause a transparent trap to the kernel. BorderPatrol includes a kernel process, `pfttrace` that logs page faults in specified processes. `pfttrace` uses `kprobes` to register call-backs whenever page faults occur. The process ID and time-stamps are passed through `relayfs` to a user-space daemon, which forwards them to the logging daemon.

5.4 Logging

Traces collected from the interposition library and the kernel page fault monitor are sent across a named pipe to the

per-host logging daemon (`logd`). The logging daemon exists to collect events from traced processes, buffer them, and write them in batches to disk. Each thread maintains a separate connection to the log daemon, so events from different threads may be received out of order. However, events from any particular thread are ordered by the pipe.

The volume and frequency of events motivated a binary logging format to limit space requirements and avoid repeated calls to expensive formatting functions. Each event consists of a fixed-length header, optionally followed by a character string and a number of integers. The event header record includes process and thread identifiers, a cycle count time-stamp, and event details, such as system call arguments and return values.

5.5 Recovering Request Paths

Events are collected from each module and sorted by clock cycles. The correlation of external links between modules with the internal links within modules provides the causal path of a request. Two rules allow the construction of paths while scanning forward in time:

1. When a module receives a message associated with request r , a fragment initiates computation for r .
2. When a fragment computing r sends a message, that message is associated with r .

Moving forward through an event stream, BorderPatrol reconstructs the history of modules, the communication channels they engage in, and messages transmitted. As virtual time proceeds, BorderPatrol maintains a mapping from file descriptors to communication channels as they are created, duplicated or destroyed. During the execution of fragments, a module *designation* identifies which request the module is currently processing. Finally, events from protocol processors indicate when *messages* are transmitted or received. In accordance with Rule 2 above, these messages are associated with the sender's current designation. An event signaling receipt of such a message updates the recipient's designation.

BorderPatrol can only determine the single most direct cause of a given fragment's executions. For example, consider a module that invokes two other modules and proceeds when both have responded. BorderPatrol will recognize that the request has moved into each of the called modules, but when the first returns, no further action will be observed. When the second module returns, BorderPatrol will determine that the request has returned and will continue within the original module, perhaps to move on yet again. In summary, BorderPatrol correctly follows forks in the request processing path, but cannot directly detect joins because they reflect information about what might have happened (if a different module had returned last), rather than what did happen in the particular trace.

In addition to explicit module communication through IPC or data streams, causal paths also continue across process creation. Often a module will spawn a helper module to assist computation. For example when a web server receives a request for a CGI URL, it will `fork` a process which then `execs` the CGI. Spawned modules consist of an implicit initial fragment which is associated with the same request that the parent was processing the moment it called `fork`.

The rules we use to recover request paths are similar in spirit to the work of Isaacs *et al.* [5] in which *temporal joins* correlate events in accordance with an application-specific join schema to reconstruct paths. BorderPatrol obtains ex-

licit internal and external causal links, so it is immediately known when requests enter and exit modules.

As a result, BorderPatrol is application independent. In contrast to join schemas, protocol processors exist solely to identify request boundaries, and contain no application-specific information.

Events on a single host use the cycle count as a total order, but these clocks will not be perfectly synchronized across multiple hosts. Since messages are logged at transmission and receipt, an approximate mapping between clocks on each host can be obtained. This mapping is sufficient to order messages, though one-way delays will not be measured precisely.

6. CASE STUDIES

Before considering performance overhead in the next section, we first show how BorderPatrol copes with two typical scenarios that require manual instrumentation to obtain precise paths with previous tracing systems.

6.1 Multi-Threaded Tiers (dearinter.net)

dearinter.net is a social networking web site which invites users to post and vote on public questions. dearinter.net consists of a multi-threaded Python application tier (TurboGears [20]) between an Apache web server [3] front-end and a PostgreSQL database back-end.

The tiers of dearinter.net communicate using several standard protocols. Web requests arrive as HTTP requests, Apache forwards application requests to TurboGears as FastCGI messages, and TurboGears issues queries to the database through the PostgreSQL protocol. BorderPatrol contains protocol processors for each of these protocols. The processors are straight-forward, and none is longer than 150 lines of code.

Examining an access log excerpt from a typical page load motivates the need for event isolation using protocol processors. Here we see that a top-level “question” page is loaded, followed by almost simultaneous requests for several embedded images.

```
9:32:42.03 /question/521 HTTP/1.1 200 1949
9:32:42.24 /img/House.jpg HTTP/1.1 200 19317
9:32:42.30 /img/Mark2.jpg HTTP/1.1 200 18820
9:32:42.34 /img/Meter.jpg HTTP/1.1 200 19947
```

Figure 8 illustrates a portion of the events logged by BorderPatrol during this page load. Only the log entries for the Apache process are shown, in order to minimize details while motivating protocol processors. First, the client establishes a connection. The HTTP protocol processor recognizes a request for the URL /question/521. To service the request, Apache connects to the FastCGI server (not shown), which responds with data that is returned to the client. The images are also served through the dearinter.net application server.

BorderPatrol separates protocol messages when multiple messages occur over a single connection, a property shared by all of the protocols used by dearinter.net. Apache supports HTTP/1.1 pipelining over persistent connections, FastCGI allows multiple outstanding requests over a single socket, and the PostgreSQL protocol allows clients to issue multiple outstanding queries.

Notice the **ProtocolIsolate** event just after the request for *House.jpg*. As the application is reading, the HTTP processor notices the boundary between two HTTP requests.

KCycles	Event
2,000,585	ProtocolInit(3) → https
2,000,592	Accept(16,0) → (3.:60983-:80)
2,000,860	ProtocolMsgRecv(3,https) [/question/521]
2,002,447	Socket() → 5
2,002,524	ProtocolInit(5) → fcgic
2,002,526	Connect(5,0) → (:40682-:9797)
2,002,591	ProtocolMsgSend(5,fcgic) [URI=/q...]
2,432,164	ProtocolMsgRecv(5,fcgic)
2,432,201	Close(5)
2,432,260	ProtocolMsgSend(3,https) [200]
2,435,414	ProtocolMsgRecv(3,https) [House.jpg]
2,435,462	ProtocolIsolate (67,161)
2,436,817	Socket() → 5,
2,436,914	ProtocolInit(5) → fcgic
2,436,916	Connect(5,0) → (:40683-:9797)
2,436,969	ProtocolMsgSend(5,fcgic) [House.jpg]
2,559,082	ProtocolMsgRecv(5,fcgic)
2,559,135	ProtocolMsgSend(3,https) [200]
2,560,658	Close(5)
2,560,808	ProtocolMsgRecv(3,https) [Mark2.jpg]
2,562,252	Socket() → 5
2,562,348	ProtocolInit(5) → fcgic
2,562,351	Connect(5,0) → (:40684-:9797)
2,562,391	ProtocolMsgSend(5,fcgic) [URI=Mark2...]
2,596,653	ProtocolMsgRecv(5,fcgic)
2,596,703	ProtocolMsgSend(3,https) [200]
2,598,234	Close(5)
...	...

Figure 8: Log of events relevant to the Apache process with Event Isolation enabled on dearinter.net. Dashed lines indicate the beginning of a code *fragment*. Fragments begin at every input event and when poll indicates that a file descriptor has become writable.

Rather than passing the compound request to the application, it isolates and passes through the first of the two. Apache continues immediately by contacting the FastCGI server and relaying *House.jpg*. Afterward, Apache calls **read** again to collect the second request for *Mark2.jpg*.

To illustrate *independence*, we generated the same workload with event isolation disabled in BorderPatrol. Now a single call to **read** fetched multiple image requests. Regardless, Apache handled the requests sequentially—it created a connection to the FastCGI server, relayed the first image, and then repeated the process for the second image. This serial behavior is an artifact of Apache’s architecture, not BorderPatrol.

This scenario is a concrete example of a module interaction that cannot be precisely deciphered without instrumentation using any other tracing tool. If Apache were to read in both requests it would be impossible to correlate which FastCGI connection corresponded to which client request. In this example, both requests are for images that are handled quite similarly, and we might happen to know that Apache handles requests sequentially. In general, the requests might be quite different, and require several module interactions to service. An error in constructing the causal path might, for example, attribute database access to a request for a static image rather than a dynamic Python page.

Validating Traces. We used BorderPatrol to create traces from a workload generated by actual dearinter.net clients from an access log provided by its developers. We validated the correctness of the traces in two ways: by detailed discussion with the developers and by comparison with ex-

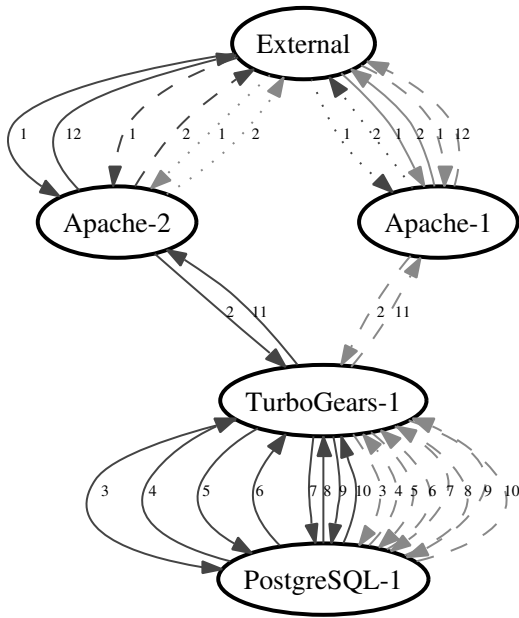


Figure 9: The original trace of the application shown in Figure 1. BorderPatrol made it easy to see that two requests were being handled by the database in a workload that should only require one. The application was reconfigured to serve images from Apache.

PLICIT instrumentation. During these discussions, the developers confirmed even the most surprising aspects of the trace, and found multiple opportunities for optimization.

We used the trace to create two types of user digestible output. First, we generated a graphical view of the paths through each module, as shown in Figure 9. Using this figure alone, the *dearinter.net* developers noticed their suboptimal configuration was causing access to the database for certain static images. With a minor modification, the application was reconfigured and yielded Figure 1, shown earlier.

Second, we found all PostgreSQL messages associated with specific URL requests. A request for `/tag/rabbits`, caused the following cycle-stamped queries:

```

316264 BEGIN; SET TRANSACTION ...
316522 SELECT NEXTVAL('tg_visit_id_seq')
317336 INSERT INTO tg_visit (id,visit_key,expiry)
VALUES (419704,'5c4...', '2007-03-18...
335990 SELECT expiry,... FROM tg_visit WHERE id = 419704
336605 END
479741 BEGIN; SET TRANSACTION ...
479891 SELECT id,user_id FROM tg_visit_identity
WHERE visit_key = '5c4...'
484013 SELECT id,tag,count FROM tag WHERE tag = 'rabbits'
485311 SELECT id,tag,count FROM tag WHERE tag = 'rabbits'
485928 SELECT id FROM qu_tag WHERE exttag_id = 1528
487024 SELECT question_id FROM qu_tag WHERE id = 2914
487778 SELECT title,sum,weight,user_id,numcomments,...
FROM question WHERE question_id = 1107
511741 SELECT user_name,email,... FROM tg_user ...
514104 SELECT id FROM qu_tag WHERE question_id = 1107
514841 SELECT tag_id FROM qu_tag WHERE id = 2911
515353 SELECT tag_id FROM qu_tag WHERE id = 2912
515782 SELECT tag_id FROM qu_tag WHERE id = 2913
516238 SELECT tag,count FROM tag WHERE id = 1525
516874 SELECT tag,count FROM tag WHERE id = 1526
517362 SELECT tag,count FROM tag WHERE id = 1527
539487 END

```

The first transaction generates a new unique browser cookie to act as a temporary user identifier. The second transaction generates the content of the page using several `selects`. First an identifier for the “rabbits” tag is obtained, from which the list of questions associated with that identifier can be loaded. The developers confirmed that the subsequent identical query was due to a previously unknown inefficiency in the structure of the application. Finally, statistics and related questions are loaded for each question (here, only 1107). BorderPatrol followed internal and external links to obtain the request path without knowledge of the internals of *dearinter.net*.

Indeed it would have been possible to obtain the above query log by simply inspecting PostgreSQL access logs. However, examining access logs is only effective when a single request is being served by the system. As soon as multiple requests arrive the log begins to conflate requests. Even if a given server’s log has unique per-request identifiers (such as a session ID) it would not be possible to correlate the activity in one server’s log with the activity generated by the same request in another server. To find the bug mentioned above, one would have to set up an instance of *dearinter.net* outside of a production environment and then manually inject requests one at a time. By contrast, BorderPatrol can automatically accumulate per-request information across all subsystems involved.

We also manually instrumented all tiers of the *dearinter.net* application. Modifications were necessary in Apache’s accept loop, HTTP processing, and FastCGI implementation; in TurboGears’ FastCGI library; in Python’s SQLAlchemy [16] library; and in PostgreSQL’s connection handling. Explicit instrumentation matched BorderPatrol’s inferred traces precisely. Of course, each modification consisted of only a single line of code, but we were struck by the difficulty in two ways. First, three very different code bases were involved, spanning two languages. Second, it was difficult to know for sure that our instrumentation was correct and complete.

The instrumentation would have missed communication that we were not explicitly aware of, such as DNS lookups. BorderPatrol’s interposition can observe all communication. Worse, our instrumentation would have been incorrect if the internal architectures of these programs had used multiple threads to satisfy a single request, or multiplexed requests on a single thread. Instrumentation code must explicitly observe the internal links that BorderPatrol infers.

To observe these links, request information can be placed in a data structure that is passed throughout the module and logged during communication. Unfortunately, real systems rarely pass a request structure through their entire code base. For example, SQLAlchemy is not tightly tied to TurboGears. When instrumenting SQLAlchemy, the request object is not accessible. In this case, it is convenient to keep the request object in thread-local storage. However, this technique is prone to error if the module to be instrumented does not have a one-to-one mapping of threads to requests. In these cases, we believe manual instrumentation would require pervasive understanding of the entire module to manage knowledge of the current request.

6.2 Event-Driven Web Server (Zeus)

Zeus [21] is an enterprise-scale commercial web server, only available to the public in binary form. We have no direct knowledge of the internals of Zeus, though we are

aware it is a high-performance event-driven design. Without source, instrumentation is impossible, and with an event-driven architecture, binding requests to threads will not succeed.

BorderPatrol traced Zeus, including its use of FastCGI, using the same protocol processors that traced `dearinter.net`, plus a DNS processor for Zeus’s reverse lookups. A sample access pattern is shown below.

```
7:58:10.03 GET /.../index.fcgi?... HTTP/1.1" 200
7:58:10.16 GET /.../statimg.fcgi HTTP/1.1" 200
7:58:10.17 GET /.../1t.gif HTTP/1.1" 200
```

KCycles	Event
1,137,563	ProtocolInit(8,41170:80) → https
1,137,567	Accept(4) → 8
1,137,756	Socket() → 9
1,137,758	ProtocolInit(9) → dnsc
1,137,780	Connect(9) → (:32784:53)
1,137,817	ProtocolMsgSend(9,dnsc,3668)
1,140,325	ProtocolMsgRecv(9,dnsc,3668)
1,140,350	Close(9)
1,140,387	ProtocolMsgRecv(8,http,0) [GET,index.fcgi]
1,141,262	Socket() → 9
1,141,342	ProtocolInit(9) → fcgic
1,141,346	Connect(9) → (/tmp/s.zeus)
1,141,540	ProtocolMsgSend(9,fcgic)
1,405,294	ProtocolMsgRecv(9,fcgic)
1,405,297	ProtocolIsolate (8,11683,0)
1,405,625	ProtocolMsgSend(8,https,0) [200]
1,407,236	ProtocolMsgRecv(9,fcgi)
1,407,238	ProtocolIsolate(8,16,0)
1,409,622	ProtocolMsgRecv(8,http,1) [GET,statimg.fcgi]
1,409,687	ProtocolIsolate(101,193,0) → 0
1,409,811	ProtocolMsgSend(9,fcgi)
1,409,862	ProtocolMsgRecv(8,http,2) [GET,1t.gif]
1,421,876	ProtocolMsgRecv(9,fcgi)
1,421,878	ProtocolIsolate(8,9716,0) → 0
1,422,567	ProtocolIsoPoll(0,0,0) → 2
1,422,590	ProtocolMsgSend(8,https,1) [200]
1,422,666	ProtocolMsgRecv(9,fcgi)
1,422,668	ProtocolIsolate(8,16,0)
1,422,927	Open(1t.gif) → 10
1,422,953	Close(10)
1,422,980	ProtocolMsgSend(8,https,2) [200]
...	...

Figure 10: Log of events with Event Isolation enabled on Zeus. Bold events show protocol processors demarcate message borders, detect data parameters, and perform event isolation (**ProtIsolate**).

The events collected are listed in Figure 10. As in the previous case study, activity begins with the arrival of a client connection. Zeus then connects to a name server to reverse resolve the client IP address. The DNS protocol processor tracks the outstanding DNS request using a witness that consists of the UDP 4-tuple and DNS request ID. BorderPatrol properly constructs paths and attributes time spent in remote, unmonitored modules.

After the name is resolved, Zeus reads an HTTP request from the client for `index.fcgi`. A FastCGI subprocess is forked and a connection is established via a Unix domain socket. Finally, Zeus writes a FastCGI message to the FastCGI server, receives the response, and relays it to the client. Next, the client requests a dynamic GIF (`statimg.fcgi`).

Although the requests are nearly simultaneous, overlap in time, and are handled by a single thread in Zeus, BorderPatrol can correctly correlate the FastCGI activity with `statimg.fcgi` rather than with `1t.gif`. This tracking does not require serialized requests, as Figure 10 illustrates. Event isolation supplies Zeus with the message for `statimg.fcgi` first, and Zeus immediately contacts the FastCGI server. With that connection in progress, Zeus returns to its event loop and receives the request for `1t.gif`. With the second request in hand, Zeus receives the response from the first request which is forwarded to the client. Finally, Zeus reads the static image and forwards it to the client.

Validating Traces. Zeus is a binary module, and we have no relationship with the Zeus developers, so validating BorderPatrol’s traces was challenging. We carefully examined request paths to ensure that, for example, the expected files were opened on the path associated with the associated URLs and that paths reconstructed during a heavy, concurrent workload matched the individual paths reconstructed from single requests.

6.3 Other Cases

In addition to the above case studies that include traces through Zeus, Bind, Apache 1.3, TurboGears, and PostgreSQL, we have traced many other modules in combination with these components. BorderPatrol has successfully traced Perl scripts used as CGI and FastCGI components, multiple web servers such as `thttpd` and a Java web server, and several simple protocols using our generic “one-shot” and “line-oriented” processors.

7. PERFORMANCE EVALUATION

Our methodology introduces overhead. In this section, we quantify this overhead both as absolute micro-benchmarks, and under realistic workloads for the case studies examined in Section 6. Our experiments were conducted on a server with a single 2.0GHz Athlon CPU and 512MB of RAM. In all experiments the overhead of collecting and writing logs is included. The logging daemon is run on the local machine with the experimental application, although it could be run remotely.

Libbtrace is interposed between the application and `libc` at run-time. Library interposition by itself has negligible overhead: less than 1%. However, `libbtrace` contains initialization code that produces startup overhead. For example, when the first application call is trapped, pointers are initialized to the real `libc` versions of the routines that are interposed upon. Then BorderPatrol initializes various data structures and connects to the logging daemon. The first and all subsequent trapped application calls typically involve some logging, invoking protocol processors (in the case of I/O) and implement event isolation.

7.1 Micro-benchmarks

A series of micro-benchmarks is shown in Figure 11. We ran experiments measuring how the request latency of a web server (Apache) degrades under various workloads as the number of concurrent clients are increased. In each graph, the solid line indicates the control scenario – measurements of a pure Apache server. The dotted line indicates the measurement of Apache wrapped with our tracing layer `libbtrace`. All workloads were generated by closed-

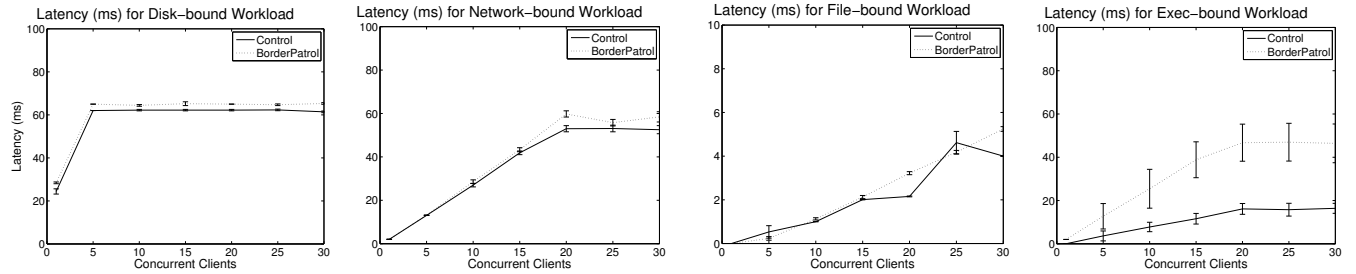


Figure 11: Latency overhead for three different micro-benchmark workloads. Each graph shows the untraced web server as a solid line, and the BorderPatrol traced version as a dotted line. The workloads are *disk-bound*, *network-bound*, small-*file bound*, and *fork/exec-bound*.

loop feedback clients, so performance reaches a plateau at saturation.

The leftmost benchmark shows latency degradation under a disk-bound workload. We generated a variety of files at 10MB each, and the clients fetched random subsets. At around 5 concurrent clients, the server becomes saturated as it cannot serve files faster than they can be loaded from disk. The overhead was roughly the same for any level of concurrency, with an overall mean of 5.37%.

In the second benchmark, a workload was generated consisting of a single 1MB file, repeatedly fetched by increasingly many clients. The file is immediately loaded into the buffer cache, so this test measures the overhead of a network-bound workload. As compared with the disk-bound benchmark, it takes longer to reach a plateau but does reach a plateau when Apache maximizes its ability to use the network. Most of the overall 7.65% overhead came from the highly concurrent workloads.

The third benchmark shows the overhead for a workload consisting of one small file repeatedly fetched by concurrent clients. The file immediately is loaded into the buffer cache and so the workload is representative of system-call intense scenarios. Across the entire range of concurrency, the mean overhead was 37.2%.

Finally, the far right benchmark illustrates the overhead when workloads involve `fork` and `exec` operations. In this benchmark, clients access a URL served by a bare-bones C program. During `fork`, our implementation performs several initializations, such as connecting the child process to the logging daemon and allocating our bookkeeping data structures. Our wrapped `exec` call performs additional initialization such as looking up the `real libc` calls through dynamic linking, and more extensive bookkeeping initialization. The mean overhead for this benchmark was 307.7%, an enormous penalty, but one that is paid only for `exec`. Real workloads, particularly in performance critical applications, will not be `exec-heavy`.

In summary, our methodology generates the most overhead for workloads that involve a large amount of process creation because each time a new process is created some initialization routines must execute. However, if the process is subsequently used to load data from disk (such as a database) or communicate with other processes (such as a web or application server) the cost is negligible. We will now turn to realistic workloads that illustrate this point.

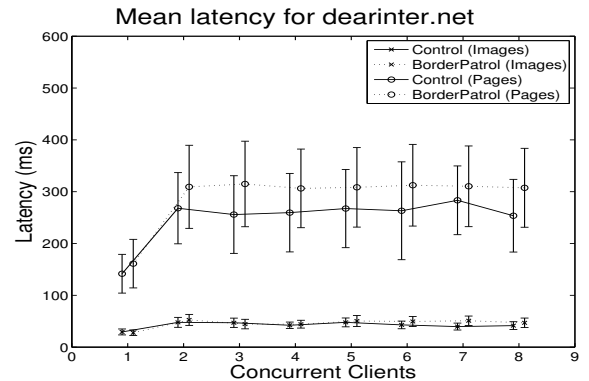


Figure 12: Latency and bandwidth overhead versus concurrent clients for *dearinter.net* (Apache, TurboGears and PostgreSQL) under a representative workload generated by replaying actual access logs.

Case Study	Events	Log (MB)	Time (s)
dearinter.net	603,962	21.63	46.29
Zeus	268,973	10.84	203.45

Figure 13: *dearinter.net* consumed approximately 470kB of log space per second during our benchmark runs. Zeus consumed approximately significantly less – 53kB per second – as the communication channels had few attributes to be logged.

7.2 Case Studies

We now revisit the case studies discussed in Section 6, and analyze the overhead for a more realistic day-to-day workload. In addition to the computational overhead that we discuss below, executing the application with BorderPatrol accumulates log entries as summarized in Figure 13. The logs are not particularly large and, of course, could be deleted when their likely value has declined.

dearinter.net. The overhead of our implementation on *dearinter.net* is shown in Figure 12. Here the workload involves more computation and random disk access than in the micro-benchmarks and so it quickly reaches capacity. Additionally, the workload includes both static and dynamic content, so we show the overhead for each in Figure 12. For the higher-latency dynamic pages, the overhead of our implementation is 16.96%, whereas the overhead is 8.4% for static images and JavaScript. The variance profile was unchanged with our tracing methodology enabled.

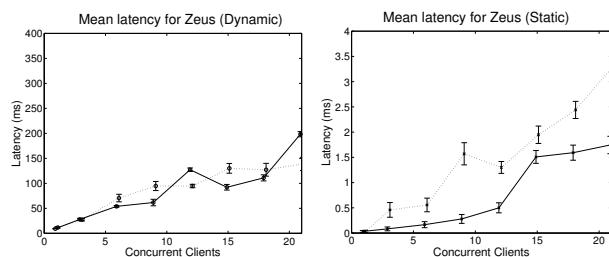


Figure 14: Latency and bandwidth overhead versus concurrent clients for Zeus (closed-source high-performance event-driven web server) under a workload of mixed images.

Application Component	System calls	
	Traced	Untraced
Apache 1.3 and TurboGears	357248	324227
PostgreSQL	16194	14582

Figure 15: BorderPatrol introduces a modest overhead in terms of system calls. Apache and TurboGears execute 10% more system when traced. PostgreSQL executes 11% more.

Zeus. Figure 14 illustrates the latency overhead of our implementation. We generated two workloads: dynamic FastCGI pages to the left and static images on the right. The mean overhead for dynamic pages is 2.0%, while static images have a 96.4% overhead. Zeus is highly tuned for serving static pages that fit in memory, so it is unsurprising that BorderPatrol imposes a larger relative penalty. When serving dynamic content through FastCGI (a workload more reminiscent of the systems we are focused on) the overhead is lost in the noise. The 100% error bars also show that BorderPatrol has not negatively affected Zeus’ concurrency profile by causing some requests to be queued excessively.

8. RELATED WORK

Previous work in request tracing has generally focused on either simplifying the instrumentation burden, or the use of statistical methods that eliminate instrumentation but also lose the ability to trace precisely.

Instrumentation. The most accurate way to correlate concurrent inputs with outputs is to leverage application-specific knowledge and explicitly declare which input corresponds to which output.

Magpie [5] seeks to provide precise traces of applications while minimizing the burden on developers. Their approach is two-pronged. First, they simplify instrumentation requirements by applying a general *temporal join* to logged events. A temporal join allows a submodule to emit trace events without knowledge of the global request that invoked it. Magpie builds a path by joining locally significant attributes across modules to produce a path. In addition, Magpie takes the pragmatic step of modifying an application framework, Microsoft’s IIS and SQL Server. Modules written within this framework require no further modification, though calls to external libraries would not be traced.

TraceBack [4] uses program analysis to inject runtime instrumentation into modules that enables a source-statement reconstruction of program execution. From that reconstruction, Traceback attempts to reconstruct paths using techniques similar to [1].

A variety of commercially available products use similar techniques. These products instrument application frameworks (such as WebSphere, WebLogic, Oracle E-Business, and Siebel) with logging calls to annotate the nodes of a causal path. The products range from the simplistic 2-tier reconstructions in [17] to many-tier reconstructions in [18, 15].

As we found in our experience with `dearinter.net` (see Section 6.1), a shortcoming of instrumentation is a practical one: all points in the application where inputs arrive must be modified. In large-scale applications where components span developer groups, are written in multiple languages, and may lack source code, modifying the application (or the frameworks) is not always possible. Further, developers may need to modify the application to make necessary information available at the time it is needed for logging, adding to their burden. Magpie’s general temporal join seeks to reduce this requirement though it is not clear that a join-able attribute is easily accessible in applications that multiplex requests on threads.

More distantly related, King [9] shows how information flow can be used to detect intrusion. Like BorderPatrol, the work tracks the inputs and outputs of processes. However, where BorderPatrol uses knowledge about protocols to distill a causal tree into per-request branches, King’s work simply collects the causal tree entirely so that a root cause can be found.

Pervasive Frameworks. Alternatively, some approaches enforce infrastructure change. Specifically, the interface of all modules is widened to include request information. This work assumes that all participating modules will be modified to implement the new interface.

Pinpoint [7] is designed specifically for J2EE web applications that associate each request with exactly one thread. This association allows any module to record the request it is working on by examining a thread-local variable. By contrast, BorderPatrol, allows applications to be written in almost any language, to use a variety of execution models (multi-threaded/event-driven), and to cross process and machine boundaries.

Causeway [2] advocates pervasive changes to applications and protocols in order to bundle meta-data alongside existing module communication. X-Trace [8] is philosophically similar work that focuses on debugging paths through many network layers. Each layer must be modified to carry X-Trace meta-data that allows path reconstruction. BorderPatrol focuses on tracing without changing applications.

Probabilistic Correlation. An alternative approach avoids augmenting the control- or data-flow by compromising on precision: the correlation between inputs and outputs can be done statistically. HP Labs has used this approach on network traffic [1], and more recently [14] on a per-process granularity using library interposition. In both cases, causality is inferred from the relative time-stamps of input arrivals and output departures.

Whodunit [6] obtains *transactional profiles* that follow request hand-offs that occur in shared memory, invisible to BorderPatrol’s tracing mechanism, by observing and analyzing module lock usage. Whodunit obtains aggregate performance information, rather than precise traces of individual requests. Nonetheless, it represents an interesting complementary approach that could augment BorderPatrol in the context of difficult to trace architectures.

Analysis From Causal Paths. Analysis of causal paths is an emerging area of research. These analyses [13, 7] *assume* causal paths can be obtained and perform higher-order analysis such as failure detection or capacity planning. BorderPatrol provides the opportunity to take advantage of this research in systems that are otherwise untraceable. Pip [13] finds bugs by dynamically detecting request paths that deviate from paths specifications provided by developers. Pinpoint [7] finds faulty modules by recording the modules involved in handling each request and applying data mining techniques to failure cases. BorderPatrol could be used to ease the adoption of these promising tools.

9. CONCLUSIONS

The lesson of BorderPatrol is that traces can be obtained for unmodified programs without sacrificing precision. We present a model for understanding the behavior of black-box distributed systems. Our model allows us to safely employ a mechanism that isolates conflated input events arriving at a module, without preventing the application from multiplexing requests. Rather than requiring systems to adopt new conventions to make request paths explicit, we are able to automatically extract causal paths through active observation.

BorderPatrol is freely available from
<http://cs.brown.edu/research/borderpatrol/>.

Acknowledgments

The authors would also like to thank Yanif Ahmad, Shriram Krishnamurthi, Kiran Pamnany, Steve Reiss, and the anonymous reviewers for their valuable feedback on improving the presentation.

This research supported by the National Science Foundation under Grant No. CNS-0614944. We thank the NSF for their support.

10. REFERENCES

- [1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, October 2003.
- [2] Khaled Elmeleegy Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Causeway: Operating system support for controlling and analyzing the execution of distributed programs. In *Proc. of the 10th Workshop on Hot Topics in Operating Systems (HotOS-X)*. IEEE Computer Society Technical Committee on Operating Systems, 2005.
- [3] Apache HTTP server. <http://httpd.apache.org/>.
- [4] Andrew Ayers, Richard Schooler, Chris Metcalf, Anant Agarwal, Junghwan Rhee, and Emmett Witchel. TraceBack: First fault diagnosis by reconstruction of distributed control flow. In *Proc. of the ACM SIGPLAN 2005 conference on Programming Language Design and Implementation (PLDI'05)*, 2005.
- [5] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *Proc. of the 6th Symposium on Operating Systems Design and Implementation (SOSP'04)*, December 2004.
- [6] Anupam Chanda, Alan Cox, and Willy Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In *Proc. of the 2nd European Conference on Computer Systems (EuroSys'07)*, March 2007.
- [7] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proc. of the International Conference on Dependable Systems and Networks (IPDS Track)*, 2002.
- [8] Rodrigo Fonseca, George Porter, Randy h. Katz, Scott Shenker, and Ion Stoica. X-Trace: A Pervasive Network Tracing Framework. In *Proc. of the 4th USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI'07)*, April 2007.
- [9] Samuel King Analyzing Intrusions Using Operating System Level Information Flow. Ph.D. thesis. September 2006.
- [10] libevent. <http://www.monkey.org/~provos/libevent>.
- [11] David Mazières. A toolkit for user-level file systems. In *Proc. of the General Track: 2001 USENIX Annual Technical Conference*, 2001.
- [12] Vivek Pai, Peter Druschel, and Willy Zwaenepoel. Flash: an efficient and portable web server. In *Proc. of the USENIX 1999 Annual Technical Conference*, June 1999.
- [13] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. PIP: Detecting the unexpected in distributed systems. In *Proc. of the 3rd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI'07)*, May 2006.
- [14] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat. WAP5: Blackbox performance debugging for widearea systems. In *Proc. of the 15th International World Wide Web Conference (WWW'06)*, May 2006.
- [15] Quest Software®. PerformaSure®. <http://www.quest.com/performasure/>.
- [16] SQLObject. <http://www.sqlobject.org/>.
- [17] Symantec. Indepth. <http://www.symantec.com/enterprise/products/category.jsp?pcid=1021>.
- [18] Wily Technology. Introscope®. <http://www.wilytech.com/solutions/products/Introscope.html>.
- [19] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. Ganger. Stardust: Tracking activity in a distributed storage system. In *Proc. of the ACM SIGMETRICS Conference*, June 2006.
- [20] TurboGears. <http://www.turbogears.org/>.
- [21] Zeus web server. <http://www.zeus.com/>.