CERIAS Tech Report 2011-24

Practical Automatic Determination of Causal Relationships in Software Execution Traces

by Sundararaman Jeyaraman
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Sundararaman Jeyaraman

Entitled
PRACTICAL AUTOMATIC DETERMINATION OF CAUSAL RELATIONSHIPS IN SOFTWARE EXECUTION TRACES

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Mikhail Atallah
Chair

Eugene H. Spafford

Samuel S. Wagstaff, Jr.

H. E. Dunsmore

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Mikhail Atallah

Approved by: Sunil Prabhakar 11/22/2011
Head of the Graduate Program Date

Graduate School Form 20
(Revised 9/10)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:

PRACTICAL AUTOMATIC DETERMINATION OF CAUSAL RELATIONSHIPS IN SOFTWARE EXECUTION TRACES

For the degree of Doctor of Philosophy

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22,* September 6, 1991, *Policy on Integrity in Research.**

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Sundararaman Jeyaraman

Printed Name and Signature of Candidate

12/01/2011

Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html

PRACTICAL AUTOMATIC DETERMINATION OF CAUSAL RELATIONSHIPS IN SOFTWARE EXECUTION TRACES

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Sundararaman Jeyaraman

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2011

Purdue University

West Lafayette, Indiana

To my dearest Velumma

ACKNOWLEDGMENTS

I would like to sincerely thank my advisor, Professor Mike Atallah, who has been a source of inspiration, a gentle and guiding presence during my long tenure as a graduate student. I would like to thank Professors Eugene Spafford, Suresh Jagannathan, Dongyan Xu, Cristina Nita-Rotaru and Chris Clifton: It was a pleasure and honor working with and learning from you.

I am greatly indebted to a great many colleagues who made my stay tolerable and enjoyable: Umut Topkara, Mercan Topkara, Rick Kennell, Abhilasha Bhargav-Spantzel, Mahesh Tripunitara, Pattie Chambers, Brian Carrier, Florian Buchholz, Ed Cates, Adam Hammer, Dr. William Gorman and Sandra Freeman.

I have been most fortunate to have an extraordinary support network of friends and family, who were instrumental in my finishing the dissertation and defending it: Velu, Amma, SK, Meena, Ram, Mukesh, Ramki, Shri, Eas, Lakshmi. Thanks for being part of my life and helping me get this proverbial monkey off my back.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

ABSTRACT

Jeyaraman, Sundararaman. Ph.D., Purdue University, December 2011. Practical Automatic Determination of Causal Relationships in Software Execution Traces. Major Professor: Mikhail Atallah.

From the system investigator who needs to analyze an intrusion ("how did the intruder break in?"), to the forensic expert who needs to investigate digital crimes ("did the suspect commit the crime?"), security experts frequently have to answer questions about the cause-effect relationships between the various events that occur in a computer system. The implications of using causality determination techniques with a low accuracy vary from slowing down incident response to undermining the evidence unearthed by forensic experts.

This dissertation presents research done in two areas: (1) We present an empirical study evaluating the accuracy and performance overhead of existing causality determination techniques. Our study shows that existing causality determination techniques are either accurate or efficient, but seldom both. (2) We propose a novel approach to causality determination based on coarse-grained observation of control-flow of program execution. Our evaluation shows that our approach is both practical in terms of low runtime overhead and accurate in terms of low false positives and false negatives.

1 INTRODUCTION

This chapter provides the motivation, general background material and the contributions of this dissertation. Section 1.1 describes how the questions about causality are fundamental to solving many information security challenges. Section 1.2 provides the definition of causality and related terms necessary to make our thesis statement. Section 1.3 provides the thesis statement, contributions and an outline of the reminder of the dissertation.

1.1 Motivation

The question of how the various events that occur in a computer system are causally related arises frequently in a variety of contexts in information security. In this section, we describe how some of the fundamental questions that arise in those fields are causal in nature and how the ability to reliably and practically answer those questions can improve the state-of-art of those fields.

1.1.1 Intrusion analysis and forensic analysis

The number of security incidents and intrusions have been rapidly on the rise over the past few years [1]. Given that it is difficult to completely secure computing infrastructure, and the heavy financial loss inflicted by intrusions [1], the importance of incidence response and recovery mechanisms can hardly be overstated. An effective intrusion response and recovery strategy is heavily dependent on the ability to analyze and answer questions regarding the events related to the intrusions in a timely and efficient manner.

Consider, by way of example, the following security incidents:

1. The security policy of an organization is violated by one or more unknown insiders (e.g., misusing the system to send spam, send confidential material to outsiders);

2. a digital crime is committed (e.g., storing illegal material on the system, or using the system to launch cyber-attacks on other systems);

3. a hacker breaks into a host inside the internal network of an organization and installs back-doors and other malware.

In all of the above cases, the ability to identify and reconstruct the sequence of events that led to each incident is critical to the success of effective response and recovery measures: In the first kind of incident, the system administrators of the organization need to determine the identity of the insiders and the underlying causes for the violation. It might even be the case that the insiders had no malicious intentions, but that the original policy had been set too tight.

For the second kind of incident, the digital investigators and the prosecution need to reliably attribute the digital crime to a particular suspect. In the third kind of incident, the administrators need to identify the attack vector of the hacker (how did the break-in occur?), to secure their systems against any future attacks that use similar techniques.

Questions of a similar nature also arise when a digital forensic expert examines digital evidence during the digital investigation process [49, 50]. Collectively, answering such questions has been referred to as the process of "event reconstruction" [48, 49]. A critical component of event reconstruction is the determination of causal relationships between events in a computer system. The knowledge of causal relationships allows an investigator to accurately "backtrack" from the effect to the cause (or causes) and to "forward track" from a cause to all its effects.

Historically, intrusion analysis systems have used causality determination schemes that are sound, have a very low overhead, but suffer from high false-positive rates [56]. Examples include BackTracker introduced by King et al. [35,36] and the process-labels scheme introduced in Bucholz et al. [42, 43].

One of the milder negative consequences of a high false-positive rate is that a security practitioner has to waste time and resources in investigating events that are completely unrelated to the security incident being investigated. At their worst, high false-positive rates make the evidence obtained susceptible to successful legal defense tactics such as the Trojan Horse defense [47].

Recently developed techniques such as Dynamic Taint Analysis (*DTA*) [57] and Virtual machine introspection [46] could be used to improve the precision of causality determination. However each of those techniques have drawbacks of their own limiting their deployability. The generalized version of *DTA* [74] results in a severe degradation in performance (50x slowdown [74]) making it impractical to be deployed. Virtual machine introspection is more precise than the traditional techniques (though not as precise as *DTA*), but requires hardware assisted virtualization to keep the CPU overhead to around 18% (in addition to the virtualization overhead) [46]. The need for a hypervisor and hardware assisted virtualization limits its applicability to resource constrained devices (e.g., smartphones).

This leaves the developers of intrusion analysis systems with the dilemma of having to choose between efficiency (low-overhead) and precision for determining causal relationships. In this dissertation, we provide a novel approach to causality determination that resolves this dilemma for analyzing certain classes of security incidents.

1.1.2 Intrusion detection

An Intrusion Detection System (IDS) can be informally thought of as a burglar-alarm for detecting security violations in a computer system. Based on their detection

methodologies, most of the intrusion detection systems fall into one of the following categories: (a) signature-based systems (b) anomaly-detection systems. Signature-based systems require a rule-base, based on which they determine if the ongoing activities in a system constitute an attack or not [18]. Anomaly-detection systems build a model of normal behavior of the system and flag any anomalous behavior as a possible attack [12–17]. Both types of systems have some fundamental limitations that limit their usefulness and practicability. Signature-based IDSs cannot detect any novel attacks that are not already present in the rule-base. Also, even simple variations of the attacks that are present in the rule-base can go undetected. Anomaly-detection systems suffer from a high false-alarm rate.

Recently, "policy-based" intrusion detection systems have been proposed as a promising alternative to both anomaly and signature-based detection systems [19–22]. The key idea is that, an intrusion is nothing but a violation of a well-defined security policy e.g., Information-flow policies. The policy-based approach is very promising and appealing because:

1. Most sites already have some form of a well defined high-level security policy e.g., Discretionary Access Control (DAC) permissions in Unix-like systems.

2. Most of the attacks result in violations of such simple, but well defined policies. For example, an intruder reading the `/etc/shadow` file as a result of a remote buffer-overflow exploit in `sendmail`, is violating the DAC policy that states that non-root users cannot access `/etc/shadow`. If the same intruder modifies `/var/log/syslog`, she violates the policy that non-root users cannot modify that specific file.

An important roadblock for the success of policy-based IDSs is the "semantic-gap" between the high-level policy statements and the low-level events that occur in a system. Consider the example mentioned in the previous paragraph. A policy-based IDS that observes the system calls that are executed in the system would observe the following sequence of system calls related to the attack: `... receive(), execve(),`

`read(), write() ...` all of which are executed by `root`. It is unclear how to accurately determine if the `read()` or `write()` actually violates the DAC policy. But if the same sequence of system calls are translated into the following form:

```
Cause:  receive()
Effects:  execve(), read(), write()
```

then, it is easier to see how an outsider has "influenced" the `read()` and `write()` calls, (through the packet `receive()`ed by `sendmail`) thereby potentially violating the DAC policy. Causal relationships provide a convenient bridge for the semantic-gap existing between the low-level events and the high-level policy statements.

1.1.3 Intrusion alert correlation

A security conscious organization typically deploys a large number of intrusion detection systems, that differ from each other based on a variety of factors such as place of deployment (network or host based), the event streams on which they operate (system calls or application-level logs) and methodology of detection (anomaly-based or signature-based). Therefore, the system administrators of an organization are typically overwhelmed with a profusion of intrusion-alerts emanating from diverse detection systems. Correlating alerts [25, 26, 28] from such heterogeneous sources could be useful due to the following reasons:

1. *Correlation as aggregation:*

   If multiple alerts could be aggregated and identified as being a result of the same attack, then the number of alerts that are actually viewed by the administrators is reduced by a great amount.

2. *Correlation for understanding:*

   An attack can be completely studied and understood only by correlating alert information from heterogeneous detection sensors. This could result in improved context-sensitivity and a decrease in the false-alarm rate.

3. *Correlation for recognizing attack scenarios:*

   There are cases where a series of attacks are first launched in preparation for future intrusions. If the earlier attacks could be correlated with the later ones, then complex attack scenarios could be reconstructed [25, 28, 29]. This could tremendously aid subsequent response and recovery efforts. Also, future detection of similar attack scenarios becomes possible.

Knowledge of causal relationships between the low-level events that generated the alerts would tremendously aid all three aforementioned functions. In fact, work by King et al. [36] has found that enriching intrusion alerts with some contextual information based on even a very simple notion of causality is quite valuable.

1.2 Background and definitions

In this section, we discuss the background concepts that are necessary for stating our thesis and discussing the main results of this dissertation. We use the theory provided by Carrier [52] as the ground work for our definitions. While our definitions do not have the same formalism and syntax used in [52] they remain faithful in semantics.

1.2.1 Digital system

A *digital system* is defined as a connected set of digital storage and event devices. Digital storage devices are physical components that can store one or more values and a digital event device is a physical component that can change the state of a storage location [52]. In this dissertation, we use the terms *digital system*, *computer system*, *system* and *host* interchangeably. Purely for expository purposes, we restrict the discussion to computer systems that run Unix-like operating systems. This is not a fundamental limitation as the concepts and approaches described should be applicable to other systems with only minor modifications.

1.2.2 State, event

The *state* of a system is the discrete value of all its storage locations and an *event* is an occurrence that changes the state of the system. The *history* of a digital system describes the sequence of states and events between two times. For the purpose of this dissertation, we consider an event to be an *action* that is performed by a *process* (or an application) on behalf of a *user*.

The rest of the dissertation focuses on a subset of events viz. system calls. We focus on system calls due to the following reason: The security-relevant behavior of any application is likely to consist of system calls. In order for a perpetrator of a security incident to accomplish anything meaningful (e.g., compromising system integrity), interaction with the operating system is necessary [1]. However, the techniques proposed in this dissertation are not restricted to system calls alone. They can be extended to any event that conforms to our definition.

In terms of Carrier [52], a system call can be thought of as a *complex event* that is composed of many *primitive events* or lower level events. Henceforth, the terms *computer event*, *system event*, *event* and *system call* are used interchangeably.

1.2.3 Event causality

Historically, various theories have been proposed to formally define, explain and reason about what is intuitively referred to as causation or cause-effect relationships e.g., Regular theories [5], Counterfactual theories [7, 8], Probabilistic theories [9–11].

In this dissertation, we focus on the standard notion of causality as defined by Hume [5], Mill [6] and Lewis [8], that captures the notion of a "necessary cause". A necessary cause can be expressed using the counterfactual: would E (effect) have occurred if it were not for C (cause)? [3, 4]. An example counterfactual query about causation in

[1] There are some exceptions to this claim. E.g., denial of service attacks might not require interaction with the underlying operating system [23].

computer events could be: Would the *root kit be still installed* (effect), if it were not for the *email received by the email-server* (cause)? [2].

Causal relationships between events are enabled by causal mechanisms [3, 4]. For example, consider a user *Alice* deleting a file *foo*. In this case, the executable code that was invoked as part of the system call `unlink()` is the mechanism that enables *Alice* to delete *foo*. Broadly, causal mechanisms that enable causality between computer system events are of the following two types:

1. The operating system

   Causal relationships between system events could be enabled through various subsystems of the operating system, e.g., the file system, the Inter-Process Communication (IPC) system. Consider the example in Figure 1.1, where process-1 and process-2 execute a sequence of system calls in the specified order. The `write()` system call of process-1 is a cause of the `read()` system call of process-2, because the result of the `read()` system call is dependent on the `write()` system call. In other words, the data that are used by `read()` are dependent on the data produced by `write()`. This causal relationship is enabled by the file system component of the OS. Similarly, other subsystems such as the process subsystem and the IPC subsystem also enable causal relationships between events [35].

```
Process-1: fd = open(foo, O_WRONLY);
Process-1: write(fd, ''hello'', 5);
Process-1: close(fd);
Process-2: fd = open(foo, O_RDONLY);
Process-2: read(fd, buffer, 4);
```

Figure 1.1.: Example to illustrate causal relationships through the operating system kernel.

[2]The installation of the root kit can be expressed as a series of system calls that copy the necessary files. Similarly, the reception of the email can be captured by system calls that received the corresponding network packets.

2. Program Address Space

   Causal relationships between two events could be enabled by the address space of a process (code and data) if both events are executed by the same process. For example, in the piece of code listed in Figure 1.2, the causal relationship between the `read()` and the `write()` calls is enabled by the `strncpy()` library call and the data buffers `buffer1 and dest`.

```
fd_r = open(foo, O_RDONLY);
fd_w = open(bar, O_WRONLY);
read(fd_r, buffer1, 10);
read(fd_r, buffer2, 10);
if(buffer1[0] == 1) {
    strncpy(dest, buffer1, 10);
}
write(fd_w, dest, 5);
```

Figure 1.2.: Sample source code to illustrate causality through Program Dependences.

In this dissertation, we focus on causation that is enabled by the process address space. We use program dependences (data dependences and control dependences) to capture causal relationships that are enabled by the process address space. We assume that program dependences are a conservative approximation of causal relationships i.e., if event C causes E, then the program statement representing E is "dependent" on the statement representing C.

For example, in Figure 1.2, the causal relationship between the `read()` and the `write()` calls is enabled by a chain of program dependences between the two calls. The `write()` call uses a value (dest) produced by the `strncpy()` library call (*data dependence*). The call to `strncpy()` is dependent on the truth value of the `if` condition (*control dependence*). The truth value of the `if` condition is in turn dependent on the `read()` system call.

1.2.4 Event reconstruction

A security incident happens as a result of a chain of events (or multiple chains of events if there are multiple causes for the incident). An event chain is an ordered sequence of events $(e_0, e_1, ., e_k)$ where event e_i is the cause of event e_{i+1} (in other words e_{i+1} is dependent on e_i). The process of identifying the chain(s) of events that result in a security incident is called event reconstruction.

1.3 Thesis statement

This dissertation describes the work done to validate the following hypothesis:

It is practical to automatically and accurately determine causal relationships between system calls in software execution traces.

In this dissertation, we focus on causal relationships enabled through program dependences and causality determination techniques used in intrusion analysis and event reconstruction systems.

1.4 Thesis contributions

This dissertation makes the following contributions:

- First, we empirically study the effectiveness of existing approaches for causality determination in event reconstruction systems. As part of this study:
  - We develop a systematic approach for evaluating the effectiveness of causality determination techniques.
  - We develop a suite of real world applications and testcases for benchmarking the effectiveness of causality determination. The suite allows us to identify the source of inaccuracy and performance overhead of the various causality determination techniques that we study.

  - Using our approach, we provide experimental data quantifying the accuracy and the overhead (time, space, memory) of each technique. Some of our results are enlightening and surprising. For example, the rate of false positives is very high for all the techniques that we evaluate, sometimes as high as 96%. The legal ramifications of this result are substantial and this highlights the need for more accurate techniques.
  - We analyze the experimental data and shed light on the conditions that lead to the inaccuracies and the overhead of the techniques we evaluate. For example, we found that BackTracker and the Static-slicing techniques do not work well in applications that exhibit recursive and iterative workflow characteristics.

- Second, based on the insights that we gain from our empirical study, we describe a new approach to causality determination. Our approach involves developing a "causality prediction model" to determine causal relationships based on observations of control-flow of a program. Our models provide efficient and accurate causality determination when the following conditions are met: (1) The program was not subject to control-flow modification or code-injection attacks and (2) The executable code of the program is available apriori.

- Third, We evaluate the effectiveness of our approach using the same systematic approach we used to study the existing causality determination techniques. We show that our approach based on causality models has a low false-positive rate and low false-negative rate (less than 5%) with a low runtime overhead.

- Finally, we analyze the experimental data from our evaluation and provide insights on improving the accuracy of causality determination even more.

1.5 Thesis organization

The rest of this document is organized as follows. Chapter 2 provides a survey of the research literature of work that is closely related to this dissertation. Chapter 3 presents the experimental evaluation and analysis of existing causality determination techniques. Based on the insights gained from the experimental evaluation, Chapter 4 describes our approach to causality determination and its experimental evaluation and analysis. Chapter 5 summarizes the contributions of this dissertation and gives directions for future research work.

2 RELATED WORK

This chapter discusses the research related to our thesis statement.

2.1 Intrusion and forensic analysis systems

Determination of causal relationships is critical for intrusion analysis and digital forensics investigations. In this section, we provide a survey of a subset of the tools available for intrusion and forensic analysis and the causality determination techniques used by them.

2.1.1 Tools using ex post evidence

Often, the only source of evidence available to an investigator is the hard disk image of a host. In addition, logs of network traffic might be available occasionally. Tools such as TCT [31], Sleuth Kit [30] and Guidance Softwares EnCase [34] help the investigators in collecting and analyzing the evidence from hard disk images. The primary focus of these tools is the discovery of evidence that might be of use to a digital forensic investigator. The investigator is still left to manually reconstruct the event sequences that fit the unearthed evidence.

In addition to evidence discovery, some tools attempt to improve the ease of evidence analysis and event reconstruction. For example, Zeitline [32] imports logs from disparate sources (e.g., system MAC times, system and firewall logs, and application data) and allows the investigator to group low-level events into "super events". Wireshark [33] can interpret the network traffic logs and provide a higher semantic view (application level view) of the network events. Despite these improvements, au-

tomatic event reconstruction is largely not possible where these tools are employed, primarily because of the limited nature of the available evidence.

2.1.2 Ex ante logging

There are scenarios where it is possible for the investigators to log events in a host prior to the occurrence of a security incident. For example, system administrators of organizations can install host-based logging mechanisms in the hosts under their supervision. If a security violation occurs in any of the hosts, then the corresponding logs can be utilized for analyzing the violation. Intrusion analysis tools such as BackTracker [35] and Forensix [39] use this approach of combining ex ante logging with ex post intrusion analysis for event reconstruction.

BackTracker

BackTracker is an automatic event reconstruction tool that identifies chains of events that could have influenced a security incident [35]. At runtime, BackTracker records system events that induce dependence relationships between operating system objects. A dependence relationship induced by an event consists of a *source* object (the cause), a *sink* object (the effect) and the time interval during which the event took place. Once a security incident is detected, BackTracker constructs a dependence relationship graph using the dependence relationships inferred from the recorded events. The nodes of the graphs are operating system objects such as files, processes and filenames. The edges represent dependence relationships between the objects. Given a set of objects that are involved in a security incident (detection points), BackTracker reconstructs the event chains by traversing the dependence graph backwards from the detection points using the dependence edges.

BackTracker takes a coarse-grained conservative approach when it comes to dependences between events executed by the same process (PD causal relationships). Given

a pair of events E_i and E_j executed by the same process, BackTracker marks E_i to be the cause of E_j if E_i was an "input" event (e.g., `read(), recv(), readdir()`) and happened before E_j. This coarse-grained approach is *sound* (i.e., no false negatives), but results in many false positives [56]. Our proposed approach to causality determination, takes a finer grained approach that reduces the false positives significantly without introducing many false negatives.

Forensix

Forensix is a forensics and intrusion analysis tool similar to BackTracker [39] . It uses the SNARE framework [40] (an event logging mechanism) for recording the events that happen in a system. System events are observed at the granularity of OS system calls. Auxiliary information such as the parameters and return values of the system calls are also recorded.

There are two main differences between BackTracker and Forensix: (1) Forensix provides support for tamperproof logging by streaming the system call information in real-time to append-only storage in a separate, hardened logging machine. (2) Unlike BackTracker which uses a dependence graph, Forensix facilitates reconstruction by providing a database query language (SQL) interface to the recorded logs, i.e., event reconstruction can be performed in an iterative fashion using a series of SQL queries.

Despite these differences, the dependence relationships captured during the analysis phase are similar to those captured by BackTracker. Specifically, Forensix treats PD causal relationships in the same coarse-grained way as BackTracker and hence suffers from a high false positive rate when determining PD causal relationships [56].

Process Labels

Though not originally intended for event reconstruction purposes, the Process Labels scheme proposed by Buchholz and Shields [42] and further expounded in Xuxian et

al. [43] and Buchholz [44], possesses the same capabilities as BackTracker. Buchholz and Shields propose a model of *pervasive binding of processes labels* to track the impact of *principals* in a system. A principal is defined as an *active agent* that performs actions in a system and interacts with other principals. Principals create, access and modify other principals and objects in the system. Every principal is associated with a unique *label* and labels are propagated from a cause to its effect. Using their model, causal relationships can be identified by tracking labels.

For determining PD causal relationships, they use the same technique as that of BackTracker: "... if an output can be observed for a principal at time t, we consider all previous inputs of time $t_i \leq t$ as potentially having caused the output. Thus any information exchange between principals (direct or indirect) has a potential effect on successive outputs of a principal. This approach will yield false positives as certain inputs may not have been the cause of an output. However, this ensures that any input that did cause an output will be considered" [44]. As a result, the Process Labels approach suffers from the same issue of high false positive rate as that of BackTracker [56].

Improved BackTracker

Sitaraman and Venkatesan [41] propose the following improvements to BackTracker (we refer to their approach as *Improved BackTracker*):

- *Offset intervals.* BackTracker treats files as atomic objects If a process modifies a file, it influences all future reads of the file regardless of which portion of the file is modified. This might lead to false dependences.

  To overcome this, Improved Backtracker records the arguments of the read, write system calls. The arguments help in observing the files at a finer granularity by providing the offsets at which each read and write system call operates. For instance, consider a process A writing to file *foo* from bytes 1 to 50. Con-

sider another process B that reads from *foo* but only the bytes 51 to 100. In this case, BackTracker falsely implicates the `read()` system call as an effect of the `write()` system call, whereas Improved BackTracker does not.

This approach is targeted at improving the accuracy of OS enabled causal relationships, while the approaches presented in this dissertation are geared towards improving causality determination in PD causal relationships.

- *Program slicing.* Sitaraman and Venkatesan propose the use of static slicing and dynamic slicing to reduce the false positives incurred by BackTracker in determining PD causal relationships.

Tracking memory mapped files

This approach improves the precision of previously discussed BackTracker-like systems by adding the ability to observe memory-mapped files at a finer granularity. The event reconstruction systems discussed so far do not consider read and write events to files that are mapped in memory. A process changing the contents of a shared memory-mapped file may affect the behavior of a legitimate process that reads the modified shared memory area later on. Memory operations cannot be logged by tracing system calls because a process makes use of pointers to reference its memory address space.

Reconstruction systems typically establish an unconditional dependence between two processes that share memory, no matter what type of operations are carried out on it, if any, leading to false positives. Sarmoria and Chapin [45] propose a runtime monitor to log read and write operations in memory-mapped files. The basic concept of their approach is the use of page faults to monitor memory access (read and write) of memory-mapped files and to log those accesses. Once the accesses have been logged, they use a BackTracker (or more precisely the Improved Backtracker with file offsets) like approach to determine dependence relationships.

This approach is limited to improving the accuracy of determining OS enabled causal relationships and does not extend to PD causal relationships.

Virtual machine introspection

A virtual-machine monitor (VMM) is a layer of software that emulates faithfully the hardware of a complete computer system [38]. The abstraction created by the virtual machine monitor is called a virtual machine. The operating system running in the virtual machine is called the guest operating system [37]. Introspecting and logging the events that occur in the guest OS from within the VMM is suitable for creating logs that are resistant to malicious tampering: VMMs represent a smaller trusted computing base than operating system kernels. Moreover the interface between the guest OS and the VMM is smaller and hence easier to secure than the interface between applications and the guest OS.

ReVirt [37] was the first intrusion analysis system to take advantage of VMMs to log system events. ReVirt creates data checkpoints and records sources of non-determinism in a system (such as user inputs) so that the system could be replayed in the future. To investigate the system, the investigator would stop the replay and install and execute tools to collect data about the system state. While ReVirt improves the *completeness* of the audit logs and their integrity, it still does not completely automate the process of causality determination and consequently event reconstruction.

Kannan et al. [46] propose a virtual machine introspection system that transparently monitors and logs data-flow in the guest OS using just the abstractions provided by the VMM. In addition to the logging mechanism, they also provide a mechanism to "query" the audit logs that facilitates quick reconstruction of events. While this approach is certainly superior to BackTracker like systems (and the proposed improvements to BackTracker), in the sense that PD causal relationships are tracked at a finer granularity (at the level of a virtual page), there are several limitations when compared to our approach: (1) Causality is tracked through data dependences alone

and control dependences are ignored. As shown by Clause et al. [74], ignoring control dependences leads to a significant drop in precision. (2) Data flow is tracked through the granularity of a virtual machine page whose size is typically 4 KB. This leads the system to falsely identify causal relationships between two system calls even if they are not causally related. (3) The virtual machine introspection technique relies on the presence of hardware assisted virtualization to reduce the overhead of monitoring and logging. This makes the approach unsuitable for constrained embedded computing environments such as network devices (e.g., routers) and smartphones where virtualization hardware is not typically present.

2.2 Information flow analysis

Information flow policies specify the way information may legally flow through a computer system. For example, a confidentiality policy might specify the set of users that are allowed to access a particular information. Examples of information flow policies include the Bell-LaPadula model [61], the Biba model [62] and the Chinese wall model [63].

The way information flows through a system also describes causal relationships. For example, if a process A *writes* into a file *foo*, which is subsequently *read* by a process B, information flows from A to B and the *write* is a cause of the *read*. Hence, past research done in analyzing how information actually flows through a computer system can potentially be leveraged to determine causal relationships in the system.

Research in the area of information flow analysis can be broadly categorized into (a) Static information flow analysis and (b) Dynamic information flow analysis.

2.2.1 Static information flow analysis

Research in static information flow analysis has focused on developing language-based approaches for detecting information flow policy violations [64]. Given the source code

of a program, these approaches try to determine if it satisfies a given information flow policy [65, 66]. Static information flow analyses suffer from the same drawbacks as using static slicing to determine causal relationships between system calls in a program. They tend to suffer from imprecision and a large number of false positives as discussed in Section 3.5. Furthermore, language-based approaches are limited in their applicability as they either require the programs to be written in specialized languages or to be manually annotated [64].

2.2.2 Dynamic information flow analysis

Dynamic information flow control mechanisms [68–70] on the other hand are both precise and do not restrict themselves to programs written in special languages. However, most of the dynamic flow control mechanisms need some form of architectural support [68, 69]. To precisely track information flow, dynamic flow systems have to examine every instruction issued by an application. The overhead involved with examining every instruction is so high that special architectural modifications are needed. The techniques we propose have a very low overhead without significantly sacrificing precision.

2.2.3 Dynamic taint analysis

Dynamic taint analysis (DTA) is one of the most commonly employed dynamic analysis techniques in security research. DTA runs a program and observes which computations are affected by predefined taint sources such as user input. It has been employed in a variety of contexts such as automatic prevention of code injection attacks and malware analysis [57].

DTA can be used as a dynamic information flow analysis system. Intuitively, dynamic tainting tracks the information flow within a program by (1) associating one or more markings with some data values in the program and (2) propagating these markings

as data values flow through the program during execution (through both data and control dependences) [74]. Similarly, causality determination between system calls can be naturally modeled as a dynamic taint analysis problem: Each system call in a program is considered a source of a taint mark. The taint marks are then propagated by control and data dependences through the program execution. Each system call also acts as a sink, i.e., the taint marks that impact that system call are logged. Historically, DTA has been formulated such that the tainting information is represented using a single bit. However, causality determination demands multiple taint sources and hence several bits of tainting information. We refer to the formulation requiring multiple taint sources and multiple bits of tainting information as *generalized DTA*.

As it was in the case of other dynamic information flow analysis systems, the early DTA techniques suffered from a high CPU overhead [73]. Several attempts have been made to improve the runtime of DTA – [75, 76, 78–81]. Many of those attempts such as [79–81] require specialized hardware limiting their applicability. Amongst the purely software based approaches, the work by Chang et al. [75] is the one with the lowest overhead for performing DTA. They report an impressive low overhead of 13% for CPU intensive applications. However the formulation of DTA used by Chang et al. to derive the 13% is not generic enough to determine causal relationships between all system call pairs in a program execution. Moreover, their approach does not propagate taints through control-flow dependences leading to significant degradation of precision [74].

In comparison, our approach offers the generality of *generalized DTA* (multiple taint sources, control-flow dependence tracking) for a certain class of security incidents and has a low overhead of under 5% on the average without significantly sacrificing the accuracy of causality determination.

2.3 Misuse detection systems

Host-based *misuse detection systems* (MDS) detect an attacker's attempts to hijack processes running in a system. An anomaly based MDS achieves this by identifying program behaviors that deviate from a known specification of normal behavior [87]. Specifications of normal behavior can be either provided manually [14] or can be derived automatically [13, 16, 17, 23, 87].

Among those MDSs that derive their specifications automatically, some derive the specifications from *event traces* of program executions [13] while others derive the specifications directly from either the program source code or binary [16, 17, 23, 87]. Those that derive their specifications from the program source are closely related to our approach to causality determination.

A major insight we gained from those MDSs is that even simple models of program execution (e.g., the control-flow of an application) can capture a significant portion of the program semantics. While data-flow definitely adds fidelity to program specifications, it was surprising to note the degree of success achieved by modeling the control-flow alone. This inspired us to explore the relationship between control-flow and causality. Specifically we ask the research question "Can we make deductions about causal relationships in a program execution, by merely observing its control-flow?". As we show in Chapter 4, it is certainly possible to do so under certain conditions (e.g., when there are no control-flow modification attacks).

3 EMPIRICAL STUDY OF CAUSALITY DETERMINATION TECHNIQUES

In this chapter, we present an empirical study of existing causality determination techniques. Section 3.1 provides a general introduction and motivation to study causality determination techniques and summarizes the contributions of this chapter. Section 3.2 gives the necessary background to understand this chapter. Section 3.3 explains our methodology for evaluating causality determination techniques. Section 3.5 presents the empirical results from our study and an analysis of the results and finally Section 3.6 discusses limitations of our approach and concludes.

3.1 Introduction

Causality determination techniques are typically employed by automated reconstruction systems such as BackTracker [35, 36], Forensix [39], Improved BackTracker [41], the virtual machine introspection scheme [46] and the Process Labels scheme [42, 43]. Despite the growing body of literature of causality determination schemes, there is hardly any work that quantifies their effectiveness. A rigorous study that quantifies their effectiveness is essential for the following reasons:

- For a system administrator and a analyzer of security incidents, a study that sheds light on the accuracy and effectiveness of causality determination techniques is useful in choosing the right event reconstruction system for their circumstances. As we shall see, techniques suited for certain programs misbehave in others.
- For a forensic investigator, the importance of reliability and accuracy metrics for techniques employed in his/her analysis cannot be overstated [53]:

  - All too often, individuals who are indicted for digital crime successfully exploit the lack of such metrics by using tactics such as the Trojan horse defense [47]. A forensic expert providing testimony in a court of law could buttress his/her conclusions by citing studies that evaluate the effectiveness of the causality determination techniques that they used in event reconstruction.
  - Event reconstruction systems often provide multiple hypotheses regarding the possible causes of a security incident. If false-positive rates are available, they can be used as priors for calculating the likelihood of each hypothesis, allowing investigators to order or prioritize the different hypotheses.

- For an information security researcher, such a study offers a guide in identifying the challenges that need to be tackled in order to build more accurate and efficient causality determination schemes.

In this chapter, we present an experimental study that evaluates the effectiveness of causality determination techniques used by most event reconstruction systems. Our contributions are the following:

- We develop a systematic approach for evaluating the effectiveness of causality determination schemes.
- We develop a suite of real world applications and testcases for benchmarking various causality determination schemes. The suite allows us to identify the source of inaccuracy and performance overhead of those techniques.
- Using our approach, we provide experimental data quantifying the effectiveness of the causality determination techniques and the overhead (time, space, memory) of each technique. Some of our results are enlightening and surprising. For example, the rate of false positives is very high for some of the commonly used

techniques, sometimes as high as 96%. The legal ramifications of this result are substantial and this highlights the urgent need for greater accuracy.

- We analyze the experimental data and shed light on the conditions that lead to the inaccuracies and overhead of the techniques we evaluate. For example, we found that BackTracker and the static slicing techniques do not work well in applications that exhibit recursive and iterative workflow characteristics (more on this in Section 3.5).

3.2 Background

This section presents a background on the terms used in the rest of the chapter.

Program slicing. Intuitively, a program slice [58, 59] of any statement S in a program is the set of other program statements that influence the execution of S and the values used in S. The process of building a program slice is referred to as program slicing or just slicing.

Static slicing. Computing the program slice of a program statement in a static fashion is referred to as static slicing. Static slicing computes the parts of the program that could influence the given statement, over all possible execution paths of the program.

Dynamic slicing. On the other hand, dynamic slicing computes a program slice for a particular execution of the program. Dynamic slicing, by definition, tracks program dependences in the most accurate fashion. However, the accuracy comes at a huge cost of space, memory and time [59].

3.3 Evaluation strategy

In this section, we explain our approach for measuring the effectiveness of causality determination techniques.

3.3.1 Metrics for causality determination

The first challenge in measuring the effectiveness of causality determination techniques is to decide upon a set of metrics. We propose to use the rates of false positives and false negatives as metrics to measure the accuracy of causality inference. False positives arise when two events e_i and e_j are implicated in a causal relationship when there is actually no such relationship. If the false positives of a technique are high, an investigator using that technique has to waste time investigating and eliminating the spurious relationships. In the worst case, the existence of spurious relationships could be leveraged by defense attorneys as part of a Trojan horse defense. Similarly, false negatives arise when the reconstruction process misses causal relationships between events. False negatives result in the investigators completely missing some (or all) of the actual causes of a security incident. Hence, we use both the rate of false positives and the rate of false negatives to evaluate the effectiveness of the techniques under consideration.

3.3.2 Measurement methodology

The next challenge in evaluating the effectiveness of causality determination techniques is to develop a suite of benchmarks to measure the metrics defined in Section 3.3.1. Initially, we considered using a suite of "scenarios" – a collection of security incidents along with corresponding audit logs, disk and memory images. The causality determination techniques would then be used to identify causal event chains for each scenario and the resulting false positives and false negatives could be measured. In fact, previous work on intrusion analysis systems described in Chapter 2 adopted a combination of qualitative reasoning and scenarios to evaluate the systems therein. However, we quickly concluded that it is non-trivial (and very expensive) to develop a comprehensive benchmark suite of scenarios that is not inherently biased or inaccurate. Our conclusion is primarily based on the experience of researchers developing

benchmark suites for Intrusion Detection Systems. Despite many attempts, there is still no consensus on the best way to benchmark IDS systems [54].

Fortunately, the following observation allows us to develop a benchmark suite for causality determination techniques that is less biased and is more scientific than a suite of scenarios:

Observation 1 *The accuracy of reconstruction systems is predicated entirely on their ability to infer causal relationships enabled through program dependences.*

Because the semantics of system calls are well defined (the effect of each system call on system objects is well understood), it is possible to accurately determine OS-enabled causal relationships. On the other hand, the causality determination techniques used by reconstruction systems vary in their ability to infer *PD causal relationships*. Hence to make an assessment of the effectiveness of causality determination, it is sufficient to measure the effectiveness in inferring PD causal relationships.

By definition, the most accurate way to determine PD causal relationships is to use dynamic slicing [59]. False positives arise when a particular technique infers a causal relationship between two events, but dynamic slicing does not. Similarly, false negatives arise when a particular technique fails to infer a PD relationship that is inferred by dynamic slicing.

3.3.3 Causality determination techniques

We study the effectiveness of the following causality determination techniques that are deployed by intrusion analysis systems described in Chapter 2:

BackTracker

BackTracker, Forensix and Process Labels treat PD causal relationships similarly. They simply consider processes as black boxes. Their causality determination policy

is simple: Any *input* event is a cause for future events. Henceforth, we refer to this technique as simply the BackTracker technique.

Static slicing

This is an improvement employed by Improved BackTracker. The inference policy can be summarized thus: An event C is a cause of another event E if, the program statement S_C corresponding to C belongs to the backward static slice of the program statement S_E.

Dynamic slicing

This is another improvement employed by Improved BackTracker. This is the dynamic variant of static slicing. An event C is considered a cause of another event E if, the instruction I_C corresponding to C belongs to the backward dynamic slice of the instruction I_E. Dynamic slicing, by definition, is the most accurate technique for detecting PD relationships and hence does not have any false-negatives or false-positives.

3.4 Experimental evaluation

3.4.1 The benchmarks

Our benchmark suite consists of a collection of open source applications and a suite of testcases for each application. Table 3.1 provides a short description of each of the applications in our test suite. The application that is smallest in terms of lines of code (LOC) is `ls` with 2,939 LOC. `GnuPG` is the largest application with 68,081 LOC. We have taken care to include both CPU-intensive applications (e.g., `gzip`) that do not frequently execute system calls, and system call-intensive applications such as `wget`.

Table 3.1: List of the applications in the benchmark suite

Application	Description	Lines of code
GnuPG 1.4.2	GNU replacement for PGP	68,081
gnu wget 1.10	Program for retrieving files through HTTP(S), FTP	22,268
find (findutils 4.2.25)	Search for files in a directory hierarchy	19,217
ls (coreutils 4.5.3)	List directory contents	2,939
cp (coreutils 4.5.3)	Copy files	3,321
wc (coreutils 4.5.3)	Print the number of bytes, words and lines in a file	3,226
tar 1.15.1	Archiving software	8,425
gzip 1.3.3	A popular data compression program	4,296
grep 2.5.1	Search files for a given input pattern	7,485

For each application in our suite, we develop a set of testcases (test suite), designed to maximize the coverage of the functionality of the respective application. Some of the applications, have publicly available regression test suites, e.g., `GnuPG`. In such cases, we borrow those regression test suites. If no such suite is publicly available for an application (e.g., `gzip`, `wget`), we develop our own test suite. In this study, we consider causal relationships between the system calls listed in Table 3.2. All the tests were run on a 2.8 GHz Pentium 4 Linux workstation with 512 MB RAM and 1 GB swap space. The number of system calls that were executed by each application as well as the number of instructions executed are presented as part of Table 3.1.

Table 3.2: List of the system calls considered in this study

File I/O	Network I/O
open	socket
open64	connect
opendir	select
read	send
write	recv
seek	recvfrom
chdir	
getdents	
access	
close	

3.4.2 Implementation of the causality determination techniques

BackTracker

Identifying causal relationships using the BackTracker technique is straightforward. We implement this functionality as a simple table lookup.

Static slicing

We use CodeSurfer [55], a program analysis tool for implementing the static slicing technique. Every system call executed by an application has a corresponding callsite in its source code. The callsite could be either a direct invocation of the system call or an indirect invocation through a library call. We use CodeSurfer to obtain static program slices of all such callsites in the source code of an application.

Dynamic slicing

We implement dynamic slicing through its functional equivalent of generalized DTA [74]. A brief description of our implementation of generalized DTA is as follows:

For every memory location (main memory, registers etc.,) potentially accessed by the instructions executed by a program, an alternate memory called *shadow memory*[1] is maintained. The shadow memory for any location A contains information about the set of system calls the current value of location A is dependent on.

For each system call instruction, the program is instrumented to generate a unique label for each specific instance of the system call. The label contains information about the system call type, the program counter value and the specific instance of the system call. The uniquely generated label is propagated to the shadow memory

[1] This is very similar to the shadow memory used by Taintbochs [72]. But, the information stored in the shadow memory is different.

locations corresponding to all the locations that are modified by this system call. Additionally, for each system call, the program is instrumented to log the labels stored in the shadow memory locations corresponding to the read operands of the system call.

For instructions that are not system calls, the program is instrumented to propagate the label set information from the read operands of the instruction to the write operands. For example, if an instruction adds two registers and writes the result into a third register, the instrumented code performs a union of the label sets corresponding to the read registers and propagates them to the shadow memory of the write register. In addition to tracking causality through program data-flow as explained, we also track causality through control-flow dependences the same way as Dytan [74].

3.5 Results

For each application in the benchmark suite, we run the testcases in the applications test suite. Each testcase produces a trace of system calls. For every pair of system calls (S_a, S_b) present in a trace, we use BackTracker, static slicing and dynamic slicing to determine if S_a is a cause of S_b as explained in Section 3.3. We calculate the rate of false positives and false negatives as explained in Section 3.3.1.

A total of 110,882 system calls and approximately 11 billion instructions are executed as part of the testcases. We report the rate of false positives and false negatives for BackTracker and static slicing in Table 3.3 and Table 3.4 respectively. Both techniques are conservative in their inference of causality and hence result in a 0% false-negative rate. The false-positive rate and the false-negative rate for the dynamic slicing technique are 0 by definition.

The time and memory overhead associated with dynamic slicing is reported in Table 3.5. Both static slicing and Back Tracker had negligible dynamic runtime overhead

($O(1)$ table lookups). Static slicing incurred a one-time cost for computing the static backward slices of the callsites, which was well within previously reported results [60].

Table 3.3: The rate of false positives for BackTracker. Avg and Std stand for average and standard deviation, respectively.

Application	System calls	Instructions	False Positives	
			Avg	Std
GnuPG	45,762	9,735,745,189	95.60	12.88
wget	49,239	168,432,151	31.78	28.99
find	2602	11,640,606	59.34	27.27
tar	7659	109,540,215	93.01	20.16
gzip	1775	824,574,115	35.32	30.43
wc	266	441,862,104	36.61	43.35
ls	2936	17,563,651	85.01	20.79
cp	464	3,511,599	67.44	31.21
grep	101	332,571	48.30	41.58

Table 3.4: The rate of false positives for static slicing. We were unable to obtain the results for `GnuPG` in the case of static slicing owing to limitations of CodeSurfer. Avg and Std stand for average and standard deviation, respectively.

Application	System calls	Instructions	False Positives	
			Avg	Std
GnuPG	45,762	9,735,745,189		
wget	49,239	168,432,151	64.94	12.44
find	2602	11,640,606	52.99	25.13
tar	7659	109,540,215	92.59	22.92
gzip	1775	824,574,115	30.21	38.41
wc	266	441,862,104	36.75	43.51
ls	2936	17,563,651	83.23	25.92
cp	464	3,511,599	54.74	41.08
grep	101	332,571	14.13	28.76

Table 3.5: Time overhead associated with dynamic taint analysis. Time overhead is the ratio of the dynamic slicing time to the normal application execution time. Avg and Std stand for average and standard deviation respectively.

Application	Avg	Std	Minimum	Maximum	Overhead
GnuPG	787.489	585.39	49.96	4953.9	8458
wget	162.808	65.55	31.32	427.72	4933
find	49.97	5.82	40.45	74.26	648.96
tar	38.40	30.95	15.06	263.1	12,802
gzip	180.91	478.55	28.02	2530.32	32,894
wc	178.06	303.92	36.68	1132.69	28,719
ls	38.32	18.73	17.47	78.60	22,153
cp	15.54	4.32	10.44	32.35	10,502
grep	28.15	9.85	16.26	53.76	53.31

Table 3.6: Memory overhead associated with dynamic taint analysis. Memory is presented in MiBs.

Application	Avg	Std	Minimum	Maximum
GnuPG	450.25	78.34	275.23	788.87
wget	431.56	84.73	274.50	638.01
find	313.99	22.78	275.03	378.9
tar	308.95	27.86	276.29	385.97
gzip	431.98	96.91	3.61	502.62
wc	345.46	21.86	274.21	357.22
ls	310.56	47.57	3.71	393.20
cp	23.62	.87	22.36	25.29
grep	159.30	156.10	4.14	384.85

We note some significant results:

1. The rate of false positives is high for both techniques. For BackTracker, the maximum false-positive rate is in the case of `GnuPG` 95.6%. For static slicing, it is 92.59% in the case of `tar`.

2. Contrary to plausible expectations [41], in most applications (except `grep`), static slicing does not provide a significantly better precision than BackTracker. In some cases such as `wget` it is actually much worse than BackTracker.

   Based on our analysis of the results for `wget` and `ls` we believe that the improvements provided by static slicing are limited when the program exhibits an "iterative" behavior with the same set of system calls repeating multiple times.

   However, one must exercise caution while interpreting the results for static slicing. It is well known that the results of static slicing depend on a variety of parameters (e.g., context-sensitivity, precision of pointer-analysis) [60]. We used the tool CodeSurfer with its default settings for static slicing. Alternate settings of CodeSurfer and alternate implementations of static slicing might produce different results.

3. The rate of false positives varies significantly across applications. For instance, in the case of BackTracker, the rate of false positives varies from 31.78% (`wget`) to 95.6% (`GnuPG`). This suggests that the nature of an application plays a crucial rule in determining its amenability to causality inference.

   We find that that the iterativeand recursive workflow nature of certain applications could result in high false positives. For instance, consider the application `ls`. A high-level overview of `ls` can be given as follows: When the `ls` command is executed, it iterates over a list of directories (supplied through command line). For each directory, ls extracts the files residing in the directory and prints the files. This is an example of iterative workflow. The extraction of informa-

tion about a file from a directory involves a `readdir()` system call and printing information about a file involves a `write()` call.

Now, consider the case of the `ls` command being invoked with the arguments dir1 dir2 over the directory structure presented in Figure 3.1. In this case, both BackTracker and static slicing declare the `readdir()` calls associated with dir1 to be causes of the `write()` calls associated with both dir1 and dir2, though there is no actual causal relationship (as determined by dynamic slicing).

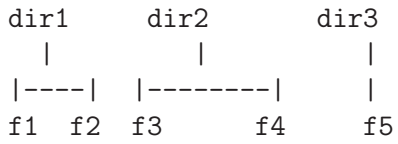


Figure 3.1.: The directory structure used in the discussion of `ls`

4. Dynamic slicing (or generalized DTA) has a lower CPU overhead for I/O-intensive applications such as wget (4,933x) when compared to CPU intensive applications such as gzip (32,894x). The overhead in Table 3.5 was calculated by adding the results from the user and sys components of the Unix time command. However, the user and sys components measure only the CPU usage of a process and do not take into account the time waiting for completion of I/O. If we account for the time taken for I/O completion (provided by the real component of time), the overhead for wget drops dramatically to 45x.

   The reasons are two-fold: (a) The worst-case time complexity of dynamic slicing for a given trace T is $O(nm)$, where n is the number of instructions executed in the trace and m is the number of system calls in the trace. For I/O-intensive applications, the increase in m is easily offset by the dramatic decrease in n. For instance, in the case of `wget`, every trace has an average n of approximately 3 million and m of approximately 1000. On the other hand, `gzip` has an average n of approximately 8 million and m of 177. The difference in the m values is very

small when compared to the difference in n; (b) For I/O-intensive applications the wall-clock time of completion is dominated by the I/O waiting time which mitigates the effect of the dynamic slicing CPU overhead.

3.6 Limitations and future work

- In this study we did not evaluate the virtual machine introspection approach used in Krishnan et al. [46]. Their approach tracks PD causal relationships at the granularity of data-flow through virtual memory pages. We believe that this approach has the potential to be more precise than both BackTracker and static slicing but less precise than dynamic slicing. One avenue for future work would be to extend our study to include this approach.

- The suigte of applications in our benchmark might not be a good representative of applications that are frequently encountered during security incidents. For example, our benchmark does not contain any multi-threaded server applications. In future work, we would like to expand the benchmark to a more comprehensive one.

- We were also constrained by the fact that our applications should be compatible with both PIN and CodeSurfer. We found that CodeSurfer was the bottleneck due to its limitations in handling applications of large size (greater than 100 KLOC). For instance, the version of CodeSurfer we used could not handle the slicing queries for `GnuPG`. Also, as indicated in Section 3.5, the results of static slicing vary depending on the precision of the underlying program analyses. A more comprehensive analysis of this effect is needed before arriving at conclusions regarding the effectiveness of static slicing as a causality determination technique.

- The testcases developed for each application were designed to exercise maximum coverage of each applications source code. However, we have not studied the

coverage statistics of the testcases. In future work, we would like to obtain the coverage statistics and use them for improving the testcases.

- Program dependences are not the only means through which causal relationships can be enabled between events that occur in the same process. Research in information-flow has proven that causality can be enabled through implicit dependences which are not captured using program dependences alone [68]. Exploring the impact of implicit dependences and the relation between information-flow and causal relationship is another promising area for future inquiry.

3.7 Conclusion

In this study, we propose an approach to evaluate the effectiveness of automatic causality determination techniques We use our approach to evaluate a suite of causality determination techniques and conclude that Back Tracker, Forensix and Process Labels have a very high rate of false positives. Based on our preliminary analysis, we posit that the recursive and iterative workflow structure of applications is a crucial reason for the high rate of false positives. Additionally, we also document the time and memory overhead of generalized DTA (dynamic slicing).

4 CAUSALITY DETECTION THROUGH CONTROL-FLOW MONITORING

In the previous chapter, we studied the effectiveness of causality determination techniques used by existing reconstruction systems. Based on the insights gained from that study, in this chapter we propose a new approach for causality determining by monitoring the control-flow of a program. Section 4.1 provides the motivation and introduces our approach. Section 4.2 describes the framework of our approach. Sections 4.3, 4.4 and 4.5 provide a detailed exposition of the various aspect of our approach. Section 4.6 presents experimental evaluation and results. Section 4.7 discusses the general applicability and limitations of our approach and Section 4.8 concludes.

4.1 Introduction

Identifying cause-effect relationships between system calls is a fundamental feature of automatic intrusion analysis and event reconstruction systems. Based on the results from Chapter 3, it is clear that existing causality determination techniques leave security practitioners and researchers in a dilemma of having to choose between accuracy and efficiency. On one hand, techniques such as BackTracker and static slicing have very low runtime overhead, but are very imprecise. On the other hand, techniques such as dynamic slicing and generalized DTA are precise but have runtime overheads that make them impractical for deployment.

In this chapter, we describe a novel approach to causality determination that resolves this dilemma for analyzing certain classes of security incidents. Broadly speaking, our approach provides efficient and highly accurate causality determination when the following conditions are met:

1. The incident does not involve control-flow modification or code-injection attacks.

2. The executable binaries of the processes that were involved in the incident are available apriori.

We note that a wide range of security incidents, such as unauthorized data access by malicious insiders, unintentional data leakage, and storage of illegal material in enterprise computer systems do not involve sophisticated tactics such as code-injection and control-flow modification of programs. They typically involve executing programs that are benign in their own right (e.g., email clients, archiving software, web browsers) but in a way that violates security policy. In such cases, our approach is able to determine causal relationships between system calls both accurately and efficiently. In the presence of control-flow modification attacks or unknown applications, we can be at least as accurate and efficient as Backtracker.

Our approach involves developing a "causality prediction model" for programs, a runtime monitor that generates augmented audit logs and an offline analyzer that uses the audit logs in conjunction with the prediction model to determine causal relationships. We exploit the fact that the control-flow of a program is a good predictor of causal relationships. We evaluate the effectiveness of three control-flow properties viz., *program counter, call stack* and *dynamic call stack* in predicting causal relationships. Based on our evaluation, our most accurate model, the *dynamic call stack* model has a low false-positive rate and false-negative rate (both under 5%) and a low runtime overhead.

Our approach is not a panacea and is not meant to be a replacement for other causality detection techniques such as generalized DTA. It is an attractive addition to the toolkit of causality detection techniques at the disposal of intrusion analysis and reconstruction systems. We envision that an intrusion analysis system could leverage our approach wherever applicable and apply heavy-weight schemes such as *generalized*

DTA in a more selective manner (e.g., in applications that are targets of control-flow modification attacks and previously unseen malware).

The primary contributions of this chapter are fourfold:

- First, we describe a new approach for determining causal relationships that leverages the control-flow properties of a program. We propose the usage of three control-flow properties viz., *program counter, call stack* and *dynamic call stack* to predict causal relationships between system calls.
- Second, we evaluate the performance of our approach over a broad suite of open source applications. Our evaluation reveals that causality determination by leveraging control-flow information is both efficient (less than 5% of additional time for program exectuion) and accurate (false-positive rate and false-negative rate under 5%).
- Third, we analyze the experimental results and explain the sources of false-positives and false-negatives in our approach (e.g., causal flow through data-flow and coverage in our testsuite) and point out ways to improve the accuracy of our models.
- Finally, we discuss the limitations and applicability of our approach and suggest future avenues of research.

4.2 General approach

We would like to determine causal relationships between system calls executed by a program. To that end, we propose a three-stage approach:

1. First, we build a model of causal relationships between every pair of system call in a program. The models consist of control-flow properties that predict the causal relationships.

2. Second, we instrument the program to generate augmented audit logs containing the control-flow properties for the system calls executed by the program.

3. Finally, we analyze the audit logs using the causal model to determine the causes of the system calls.

In the following sections, we describe each of the aforementioned steps in greater detail.

4.3 Building causal models

In the past, several host-based intrusion detection systems have used simple aspects of the control-flow of a program to derive high-fidelity specifications of valid program behavior([82–87]). Similar to those approaches, our philosophy is to see how far we can use simple control-flow properties to predict causal relationships between the system calls of a program.

A causal model of a program is a specification of the causal relationships between every pair of system calls in that program. It lists the control-flow conditions that must occur for each causal relationship to happen. Control-flow conditions are properties of the dynamic control-flow trace of a program. For example, in the source code snippet in Figure 4.1, the `write()` system call in line `8` is an effect of the `read()` system call in line `3` but not the `read()` system call in line `4`. This could be modeled as a condition based on the program counter of the `read()` and `write()` system calls (see Table 4.1). The conditions are expressed as logical formulae using control-flow properties.

Causal models are built by scanning "causal traces" of a program. For a given execution of the program, the causal trace contains the actual causes of system calls in that execution and the control-flow properties of the system calls. We use generalized DTA to generate the causal traces: Each system call acts as a new "taint source" and as a "taint sink". We execute the program over a set of training test cases and

collect the taint information along with the control-flow properties for each system call executed in the training. The taint information provides the actual causes of a system call.

```
1
2                       fd_r = open(foo, O_RDONLY);
3                       fd_w = open(bar, O_WRONLY);
4                       read(fd_r, buffer1, 10);
5                       read(fd_r, buffer2, 10);
6                       if(buffer1[0] == 1) {
7                           strncpy(dest, buffer1, 10);
8                       }
9                       write(fd_w, dest, 5);
```

Figure 4.1.: Sample source code to illustrate causality through Program Dependences.

Table 4.1: Dependence model for listing in Figure 4.1

Dependence Relationship	Control-flow conditions
$read \xrightarrow{Dep} write$	$PC(read) = 3$

4.4 Control-flow properties

In the following subsections, we discuss the three control-flow properties that we evaluate.

4.4.1 Callsite

The first and the simplest control-flow property that we evaluate is the callsite of a system call. We model the callsite using the program counter *(PC)* value of the system callsite. Let us say system call C at callsite $PC(C)$ is an actual cause of system call E at callsite $PC(E)$ during the training phase. After the training phase,

given two system call instances C_i and E_j, the callsite causal model predicts C_i to be the cause of E_j if $PC(C_i) = PC(C) \wedge PC(E_j) = PC(E)$. Table 4.1 provides an example of a causal model that is based on the callsites of the system calls.

4.4.2 Callstack

The next control-flow property that we evaluate is the callstack when a system call is invoked. When a system call A is executed its callstack $CS(A)$ is the list of return addresses of currently actively routines $\{A_n, A_{n-1} \ldots A_0\}$, where n is the number of frames in the callstack and A_n is the program counter of the system call A.

The callstack of a program contains information about the past and current states of the program. It captures more of the program semantics than just the program counter of a system call. This allows us to predict causal relationships more accurately than by just using the program counter. For example, consider the source code snippet in Figure 4.2. The `read()` system call in line 6 is the cause of the `write()` system call in line 13 only if the `read()` is executed under the context of function `g()` ($g() \rightarrow readfile() \rightarrow read()$). It is not a cause when executed under the context of function `f()` ($f() \rightarrow readfile() \rightarrow read()$). The call stack helps differentiate between the the two cases – `read()` is a cause of `write()` only if $CS(read)$ is $\{6, 27, 41\}$ and not if it is $\{6, 34, 40\}$.

The causal model using callstack information for the code snippet in Fig 4.2 is presented in Table 4.2. The first column lists the causal relationships and the second column lists the control-flow conditions that must be satisfied for the dependence relationship to hold. If a causal relationship is not present in the model, e.g., $write \xrightarrow{Cause} write$ then it is assumed that such a relationship could never happen.

```
1 #define BUFSIZE 100
2
3 void readfile (char * f, char *b)
4 {
5   int fd_r = open(f, O_RDONLY);
6   read(fd_r, b, BUFSIZE);
7   close(fd_r);
8 }
9
10 void writefile(char *f, char *b)
11 {
12   fd_w = open(f, O_WRONLY);
13   write(fd_w, b, BUFSIZE);
14   close(fd_w);
15 }
16
17 void prntscrn(char *b)
18 {
19   write(stdout, b, BUFSIZE);
20 }
21
22 void g()
23 {
24   char b[BUFSIZE];
25
26   readfile("foo", b);
27   writefile("bar", b);
28   return;
29 }
30
31 void f()
32 {
33   readfile("foo", b);
34   prntscrn(b);
35   return;
36 }
37
38 int main() {
39   f();
40   g();
41   return;
42 }
```

Figure 4.2.: Sample source code to illustrate causal models using callstack information

Table 4.2: An example of a causal model using callstack information

Causal Relationship	Control-flow conditions
$open \xrightarrow{Cause} read$	$CS(open) = \{6, 27, 41\} \wedge CS(open) = \{6, 34, 40\}$
$read \xrightarrow{Cause} write$	$(CS(read) = \{7, 27, 41\} \wedge CS(write) = \{14, 28, 41\}) \vee$ $(CS(read) = \{7, 34, 40\} \wedge CS(write) = \{20, 35, 40\})$

4.4.3 Dynamic callstack

The final and the most complex control-flow property that we evaluate is the dynamic callstack. A dynamic callstack contains information about not only the return addresses of active routines, but also the instance information of each of the return addresses. We define the dynamic call stack of system call A as $DCS(A) = \{A_n^i, A_{n-1}^j \dots A_0^1\}$ where n is the number of frames in the callstack, A_n^i is the i^{th} instance of the program counter A_n of the system call A, and A_0 represents the return address of the `main()` function.

The dynamic callstack has more information about the history and the current state of the program than the callstack alone. Specifically, the dynamic callstack allows us to differentiate between the same system call executed in different iterations of a loop. Consider the code snippet in Figure 4.3 that prints out the list of files for a list of directories (modeled loosely after `ls`). The `readdir()` system call in line 6 is a cause of the `write()` system call in line 7 only if they are both executed within the same "instance" of the `printfiles()` function. For instance, consider two directories `dir1, dir2` whose files are being listed out by the program. The `write` calls that belong to `dir1` should not be considered an effect of the `readdir()` system calls for `dir2` and vice versa. The dynamic callstack causal model allows us to capture this constraint. The dynamic callstack causal model for the code snippet in Fig 4.3 is provided in Table 4.3.

```
1
2  void printfiles(pendingdir)
3  {
4    struct dirent *f;
5    DIR *dr = opendir(pendingdir->name);
6    while((f = readddir(dr))) {
7      write(stdout, f->d_name);
8    }
9  }
10
11 void main()
12 {
13   ...
14   while (pendingdir) {
15     printfiles(pendingdir);
16     pendingdir = pendingdir->next;
17   }
18 }
```

Figure 4.3.: Sample source code to illustrate causal models using the dynamic call-stack information

Table 4.3: An example of a causal model using the dynamic callstack information. We use "*" in the superscript as a wildcard to indicate that the instance information does not matter.

Causal Relationship	Control-flow conditions
$opendir \xrightarrow{Cause} readdir$	$DCS(opendir) = \{5^*, 15^j\} \wedge DCS(readdir) = \{6^*, 15^j\}$
$readdir \xrightarrow{Cause} write$	$DCS(write) = \{7^*, 15^j\} \wedge DCS(readdir) = \{6^*, 15^j\}$
$opendir \xrightarrow{Cause} write$	$DCS(write) = \{7^*, 15^j\} \wedge DCS(opendir) = \{5^*, 15^j\}$

4.5 Augmented audit logs and offline analysis

An online monitor generates audit logs of system calls executed by applications. In addition to the information found in traditional "c2 compliant" audit logs [88], the monitor also logs the control-flow properties of system calls. Given the augmented audit logs of system calls for a program, we can generate the list of "causes" for each system call as follows:

- Let CM be the causal model of the program. CM could be the callsite, callstack or the dynamic call stack model.
- For each system call S_i in the audit log
  - Let $C(S_i)$ be the set of causes of S_i
  - Let the control-flow property of S_i be $CFP(S_i)$. CFP could be the callsite, callstack or dynamic callstack depending on the type of CM
  - For each system call S_j in the audit log where j varies from 0 to i
    - Let the control-flow feature of S_j be $CFP(S_j)$
    - If $CM(CFP(S_i), CFP(S_j))$ evaluates to $TRUE$ then add S_j to $C(S_i)$
  - Print out $C(S_i)$

The time complexity of the analysis algorithm is $O(N^2M)$ where N is the number of system calls in a program's audit log and M is the maximum size of the logical formulae used in CM.

4.6 Experimental evaluation and results

In this section we measure the performance of our causal models (the callsite, callstack and dynamic callstack models) in identifying causal relationships using a suite of

Table 4.4: List of the applications in the benchmark suite

Application	Description	Lines of code (LOC)
GnuPG 1.4.2	GNU replacement for PGP	68,081
gnu wget 1.10	Program for retrieving files through HTTP(S), FTP	22,268
find (findutils 4.2.25)	Search for files in a directory hierarchy	19,217
ls (coreutils 4.5.3)	List directory contents	2,939
cp (coreutils 4.5.3)	Copy files	3,321
wc (coreutils 4.5.3)	Print the number of bytes, words and lines in a file	3,226
tar 1.15.1	Archiving software	8,425
gzip 1.3.3	A popular data compression program	4,296
grep 2.5.1	Search files for a given input pattern	7,485

open source applications described in Table 4.4. We focus on measuring the following metrics: false-positives, false-negatives and runtime overhead.

For each application, we develop a set of testcases designed to exercise as much of their respective functionality as possible. Some of the applications (e.g., `GnuPG`) have a well defined regression testsuite. We reuse those tests where available and develop our own testcases otherwise [56]. We divide those testcases equally into a "training set" and a "test set". The training set is used to obtain the causal traces and build the causal models as described in Section 4.3. After the causal models are built, we use the test set to obtain the performance metrics for each of the model. All the tests were run on a 2.8 GHz Pentium 4 Linux workstation with 512 MB RAM and 1 GB swap space.

4.6.1 False positives

False positives occur when a causal model predicts that a system call S_i is a cause of another system call S_j when in reality it is not. The rate of false positives is calculated using the following formula:

$$\text{Rate of false positives} = \frac{\text{Number of false causal predictions}}{\text{Total number of causal relationships}} * 100$$

The false positives are calculated by comparing the results of causality determination against those of dynamic slicing: False positives arise when a particular causal model infers a causal relationship between two events, but dynamic slicing does not. In Figures 4.7, 4.8 and 4.9 we present the rate of false positives of the causal models in comparison to the BackTracker [35, 36] and static slicing techniques. We make the following observations:

1. Observing the control-flow of a program enhances the ability to predict causal relationships in that program. In general, the more we observe the control-flow, the better the accuracy of prediction:

    - Even the simplest of our causal models viz. the callsite model, has a dramatically lower false positive rate when compared to both BackTracker and static slicing. The callsite model has a false positive rate of 19.61%. In comparison, BackTracker has an average false positive rate of 55.97% and the static slicing model has an average false positive rate of 45.33%. At its best, the callsite model has a false positive rate of 1.51% for the program `grep` and at its worst the model has a false positive rate of 50.01% for the program `ls`.
    - As expected, with the additional context and state provided by the program callstack, the callstack model improves upon the callsite model with

an average false-positive rate of 10.54%. It has a very low false-positive rate for applications such as `GnuPG, find` and `grep`. In the worst case, it suffers from a false-positive rate of 43.48% for the program `ls`.

- The dynamic callstack model further improves upon the accuracy of the callstack model with an average false-positive rate of 4.45%. For most of the applications in our benchmark, the false-positive rate of the dynamic callstack model is less than 5% with zero false-positives for applications `grep, wc`. As it is with the other causal models, the worst case of false-positives occurs for `ls` (21.45%).

2. The rate of false-positives varies significantly from one application to another for all our causal models. Applications such as `grep` bring out the best in all our causal models while applications such as `ls` bring out the worst. The nature of the application has a significant impact on the accuracy of causality determination.

   We investigated the test cases of `ls` to understand the reasons for the large number of false positives for that program. Our analysis revealed that most of the false positives for `ls` resulted from testcases that exercised the "recursive" option for `ls` (`ls -r`) (similar to the cases of BackTracker and static slicing).

   A high-level overview of `ls` can be given as follows (reproduced from Section 3.5: When the `ls` command is executed, it iterates over a list of directories supplied through command line. For each directory, `ls` extracts the files residing in the directory and prints the files. The extraction of information about a file from a directory involves a `readdir()` system call and printing information about a file involves a `write()` system call. Figure 4.5 captures this functionality in the form of source code derived from `ls`.

   Now consider the case of the command `ls -r` being executed over the directory structure illustrated in Fig 4.4. Based on our analysis of the source code of `ls` we know that a `readdir()` call associated with one directory (say "foo") can

be a cause of the `readdir()` call associated with another directory (say "bar") only if "bar" is a descendant of "foo" in the directory hierarchy. Our causal models fail to capture this "descendant of" relationship and spuriously label the `readdir()` calls associated with the subdirectory "dir2" to be causes of the `readdir()` and `write()` calls associated with subdirectory "dir3".

The "descendant of" relationship is enabled primarily through the data-flow of the program. The Figure 4.6 lists the execution trace of `ls -r` (corresponding to the source code in Figure 4.5) over the directory structure in Figure 4.4. In Figure 4.6, the control-flow that immediately follows `readdir(‘‘dir2’’)` is identical to that following `readdir(‘‘dir3’’)`. Similarly the control-flow that precedes `write(‘‘f1’’)` is identical to that preceding `write(‘‘f3’’)`. However their respective data-flows (marked as colored arrows in Figure 4.6) are distinct. In order to eliminate false positives in this case the data-flow must be observed.

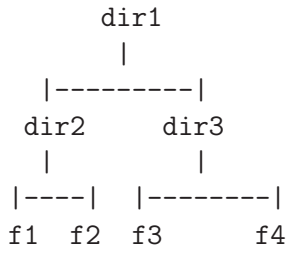


Figure 4.4.: The hierarchical directory structure used in the discussion of `ls`

```
1
2  // Queues a directory to the pending directories queue
3  void queue_directory(char *dir)
4  {
5    new->name = dir;
6    // Data flow to establish parent-child relationship
7    new->next = pendingdir;
8    pendingdir = new;
9  }
10
11 // Prints the names of the files in the given directory
12 void print_dir(pendingdir)
13 {
14   struct dirent *f;
15   DIR *dr = opendir(pendingdir->name);
16   while((f = readddir(dr))) {
17     write(stdout, f->d_name);
18     if (isdir(f)) {
19       queue_directory(f);
20     }
21   }
22 }
23
24 void main()
25 {
26   ...
27   while (pendingdir) {
28     thispend = pendingdir;
29     print_dir(thispend);
30     pendingdir = pendingdir->next;
31   }
32 }
```

Figure 4.5.: Source code snippet that captures the behavior of `ls -r`. The snippet was derived from the original source of ls (`ls.c`) available in the `coreutils-4.5.3` package. We have simplified and modified the source considerably to highlight only the relevant portions.

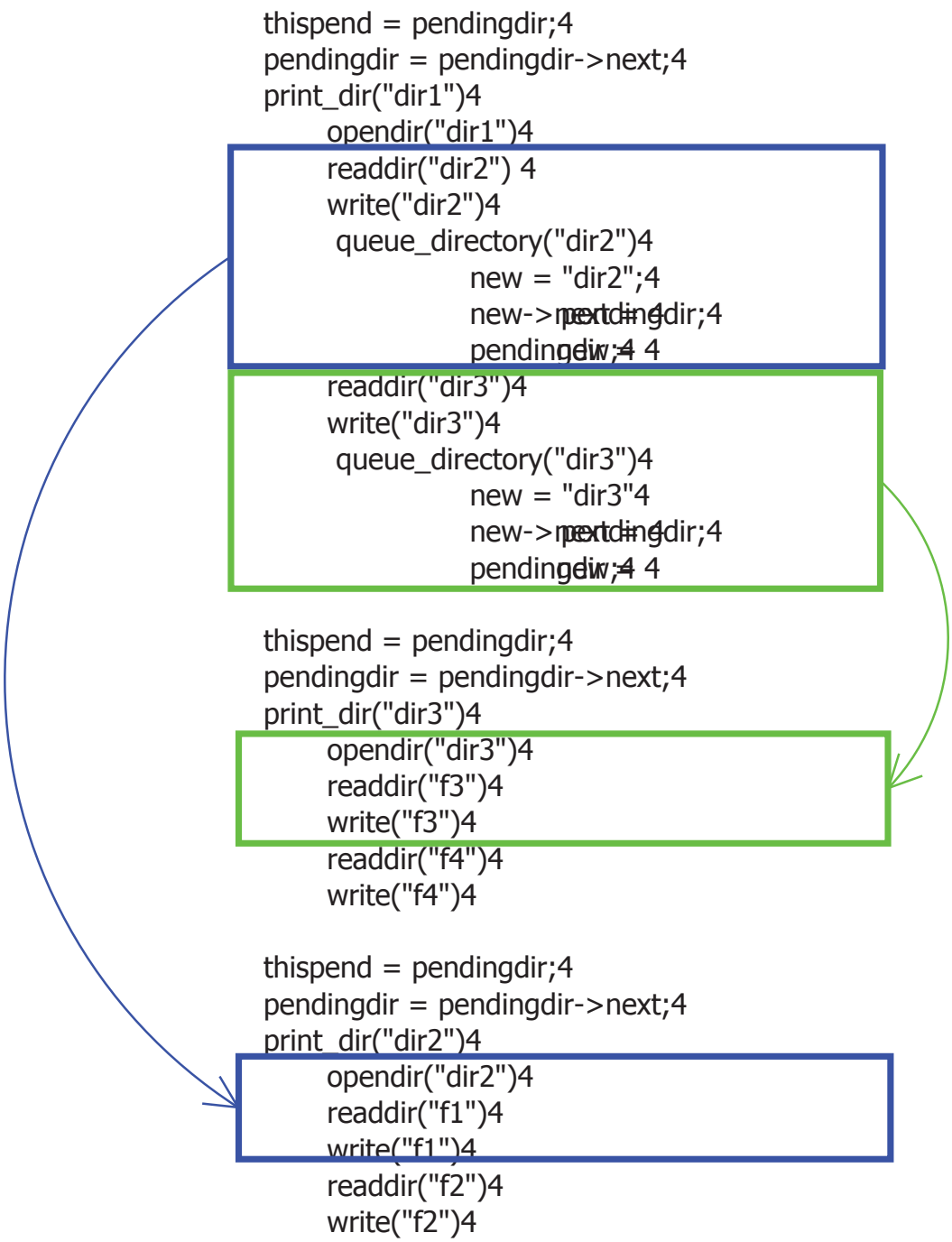
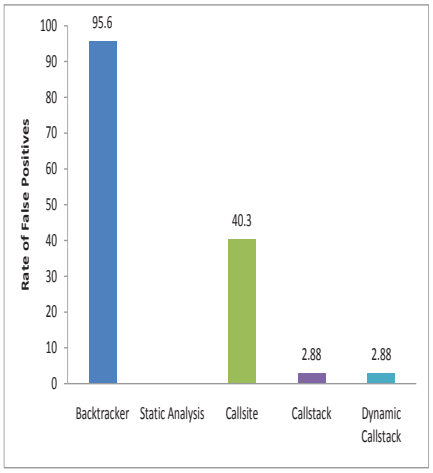
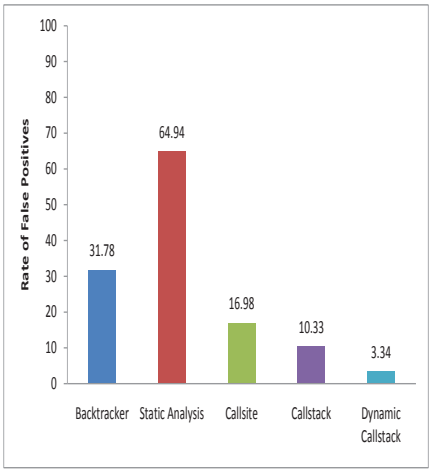


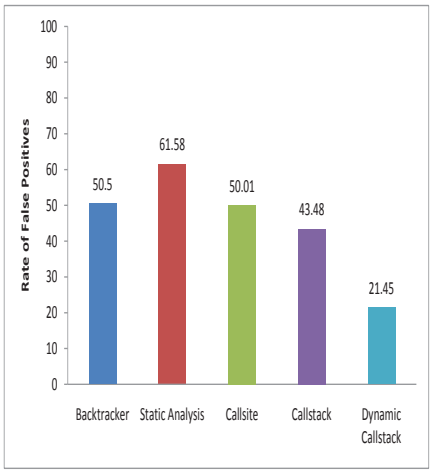
Figure 4.6.: The control-flow and data-flow trace of a sample execution of the code listed in Figure 4.5. Data-flow is depicted through colored arrows.



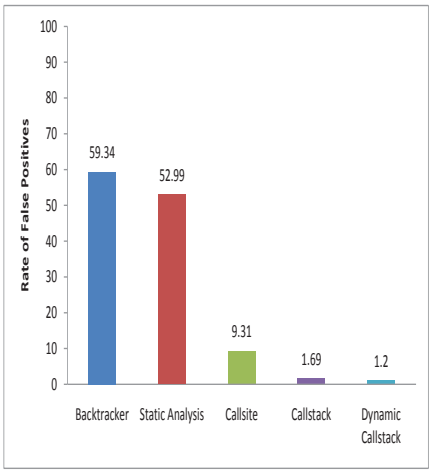
(a) GnuPG



(b) wget

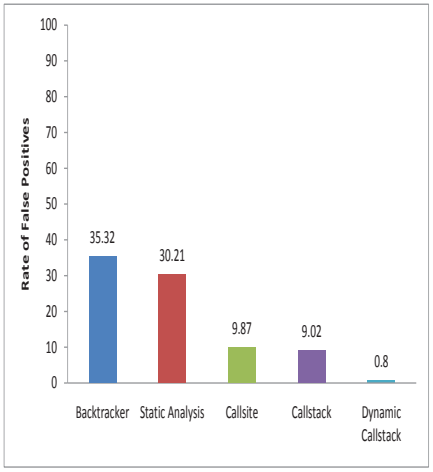


(c) ls

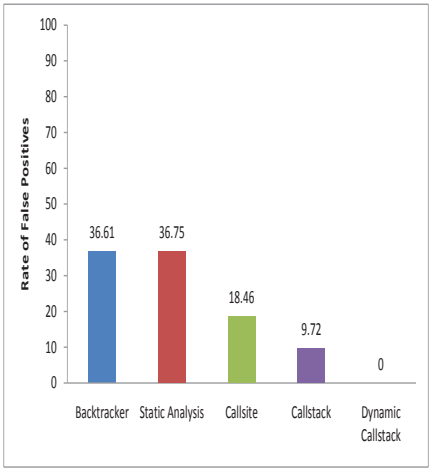


(d) find

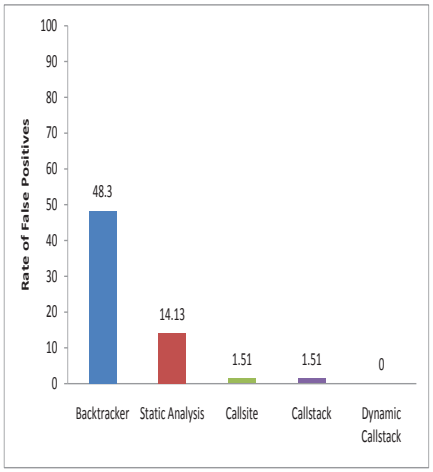
Figure 4.7.: Rate of false positives of causal models using Callsite, Callstack and Dynamic Callstack for `GnuPG`, `wget`, `ls`, `find`.



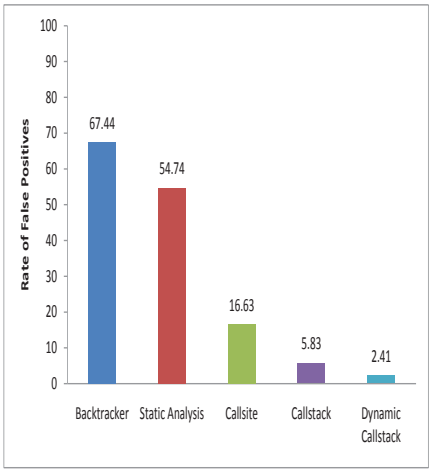
(a) gzip



(b) wc

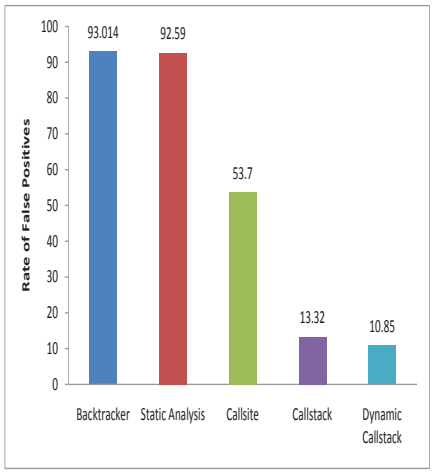


(c) grep

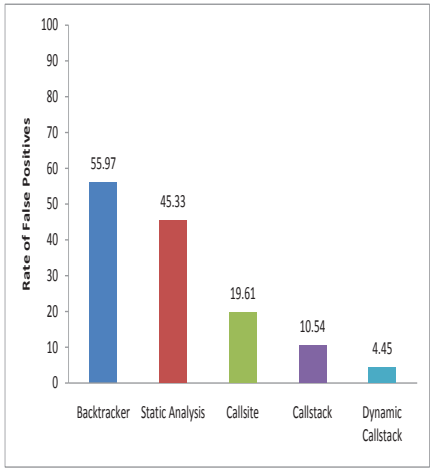


(d) cp

Figure 4.8.: Rate of false positives of causal models using Callsite, Callstack and Dynamic Callstack for `gzip`, `wc`, `grep`, `cp`.



(a) tar



(b) Average

Figure 4.9.: Rate of false positives of causal models using Callsite, Callstack and Dynamic Callstack for `tar` along with the average false positive rate across all the applications.

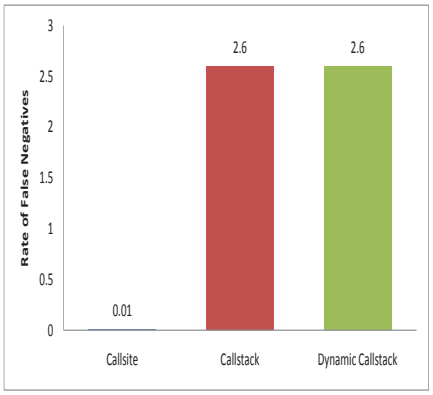
4.6.2 False negatives

False negatives occur when a causal model misses an actual causal relationship. The rate of false negatives is calculated using the following formula:

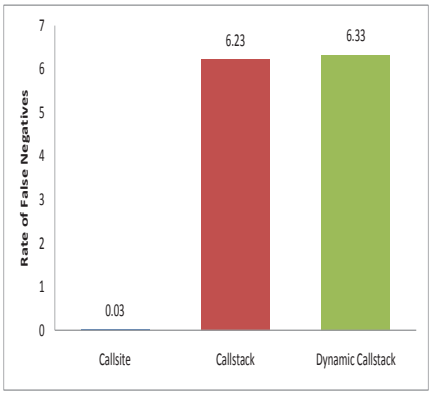
$$\text{Rate of false negatives} = \frac{\text{Number of missed predictions}}{\text{Total number of causal relationships}} * 100$$

False negatives arise when a particular technique fails to infer a causal relationship that is inferred by dynamic slicing. We list the false negatives of the causal models in Figures 4.10, 4.11 and 4.12. All our causal models suffer from false-negatives. The callsite model has the lowest average rate of false-negatives with 0.85%. The callstack and the dynamic callstack models have higher rates of false-negatives at 4.65% and 4.93% respectively. The rate of false negatives increases as the complexity of the observed control-flow feature rises: $Callsite < Callstack < Dynamic\,Callstack$.

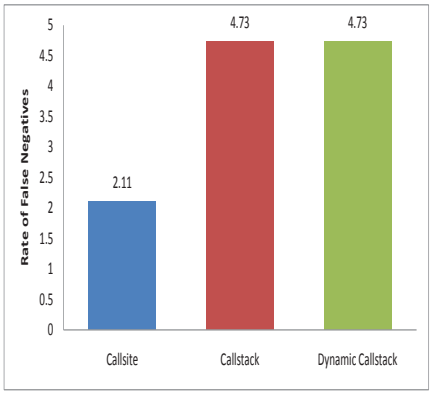
Our causal models miss actual causal relationships between system calls when: (1) System calls that were not present in the training data are encountered in the test data. (2) System calls are associated with previously unobserved control-flow properties. Both can be attributed to the incompleteness in the training data. In general, the more complete the training data, the lower the false negatives are. Refer to Section 4.6.4 for a detailed discussion on the impact of training data on false negatives.



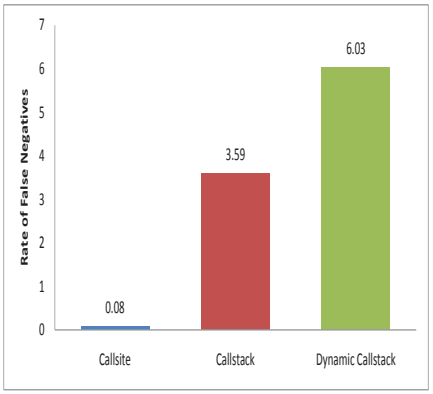
(a) GnuPG



(b) wget

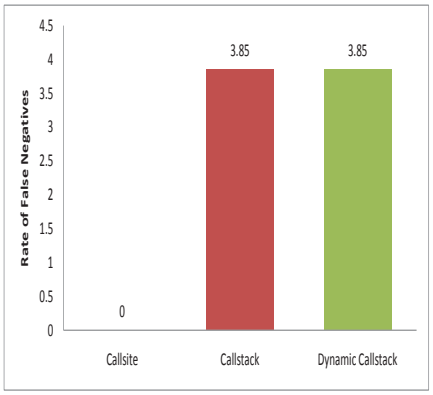


(c) ls

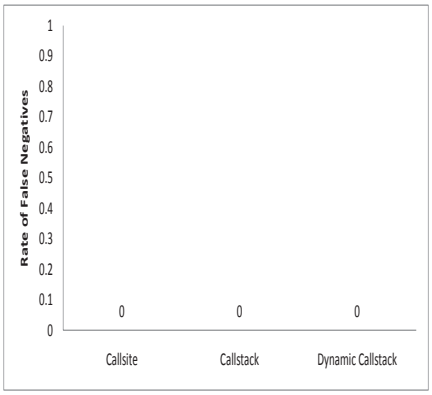


(d) find

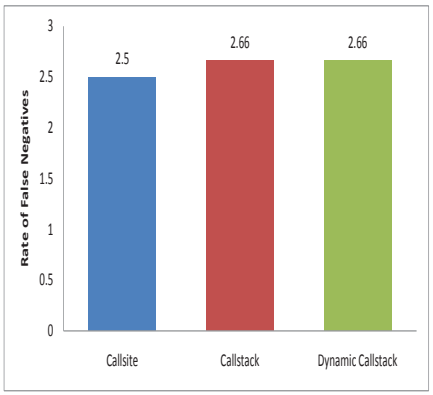
Figure 4.10.: Rate of false negatives of the Callsite, Callstack and Dynamic Callstack models for `GnuPG`, `wget`, `ls`, `find`.



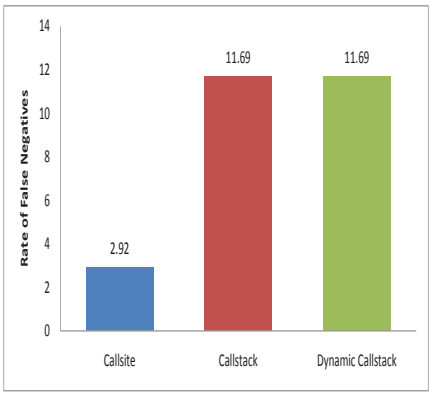
(a) gzip



(b) wc

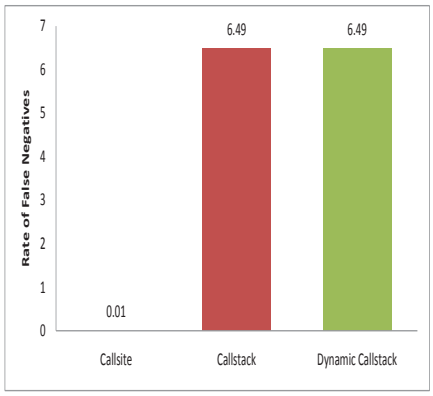


(c) grep

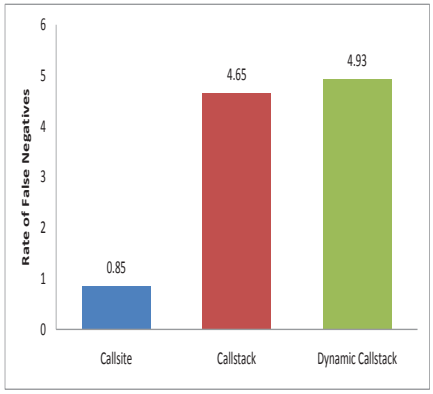


(d) cp

Figure 4.11.: Rate of false negatives of the Callsite, Callstack and Dynamic Callstack models for `gzip, wc, grep, cp`.



(a) tar



(b) Average

Figure 4.12.: Rate of false negatives of the Callsite, Callstack and Dynamic Callstack causal models for `tar` and the average false negative rate across all the applications.

4.6.3 F-measure

A straightforward comparison between our causal models and previous approaches based solely on either false positives or false negatives is not possible: Our approach to causality detection is neither sound nor complete, i.e., it suffers from both false positives *and* false negatives whereas the previous approaches for causality determination (BackTracker, Static slicing) are all complete, i.e., they do not suffer from false-negatives [1].

The *F-measure* is a metric used to combine the false positive rate and the false negative rate in the field of information retrieval [91] and is defined as the weighted harmonic mean of *precision* and *recall*:

$$
\begin{aligned}
F &= 2.\frac{\text{Precision . Recall}}{\text{Precision + Recall}} \\
&= 2.\frac{(1-\text{False Positive Rate})(1-\text{False Negative Rate})}{1-\text{False Positive Rate}+1-\text{False Negative Rate}}
\end{aligned}
$$

The value of F varies between 1 to 0, with 1 being the best score and 0 being the worst. We employ the *F-measure* as a metric to effect a straightforward comparison between our approaches and previous approaches. We list the *F-measure* for our causal models along that of Backtracker and static slicing in Table 4.5. For all applications in the benchmark suite, our causal models have better F values than both Backtracker and Static slicing.

[1] Dynamic taint analysis is both sound and complete. However it is not a *practical* technique for causality determination due to its runtime overhead. Hence we ignore dynamic taint analysis in this discussion. Similarly we do not consider the virtual machine introspection technique in this discussion.

Table 4.5: The *F-measure* of Backtracker, Static slicing, Callsite, Callstack and Dynamic callstack models. We were unable to obtain the results for `GnuPG` in the case of static slicing owing to limitations of `codesurfer`.

Application	Backtracker	Static slicing	Callsite	Callstack	Dynamic callstack
GnuPG	0.08		0.75	0.97	0.97
wget	0.81	0.52	0.91	0.92	0.95
ls	0.66	0.56	0.66	0.71	0.86
find	0.58	0.64	0.95	0.97	0.96
gzip	0.79	0.82	0.95	0.93	0.98
wc	0.78	0.77	0.9	0.95	1
grep	0.68	0.92	0.98	0.98	0.99
cp	0.49	0.62	0.9	0.91	0.93
tar	0.13	0.14	0.63	0.9	0.91
Average	0.64	0.66	0.87	0.92	0.95

4.6.4 Impact of training data

The choice of training data plays a significant role in determining the accuracy of dynamic analysis techniques such as ours. While we have reused existing regression suites where applicable, we had to build our own testcases for many applications. As our approach involves building causal models using the control-flow properties observed in the training data, the quality of the training data is of paramount importance. In this section, we attempt to answer the questions such as:

1. Is our training data sufficient?
2. If not, does adding additional testcases improve the effectiveness of the causal models?

We plot the metrics for model effectiveness (false positives and false negatives) for training data of different sizes in order to understand the relationship between our model accuracy and the completeness of the training data. For each training set size, we choose five random subsets from the original training set of the same size and average the results across them. Figures 4.13, 4.14 and 4.15 list the results for the dynamic callstack model. We omit the results from the causal models as the trends were similar in nature. Some observations:

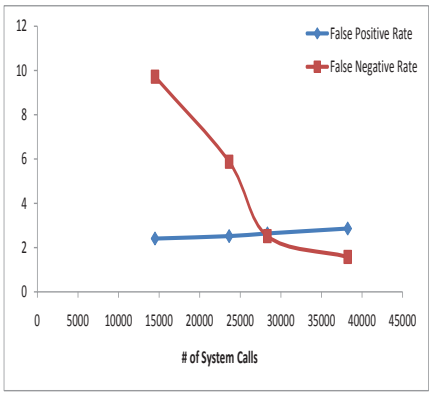
1. As the size of the training data increases, the false negatives decrease sharply. However the false positives slightly increase with training data size (though at a slower rate than the drop in false negatives). While the rise in false positives is surprising at first blush, upon reflection, this is to be expected:

   During the testing phase, if a system call is a "miss" in the causal model, then all the causal relationships involving the system call are flagged as false negatives. These causal relationships will not contribute towards the false positives. With a small training set the "misses" are higher, resulting in higher false negatives *and* lower false positives. As the training set increases, the number of misses drops

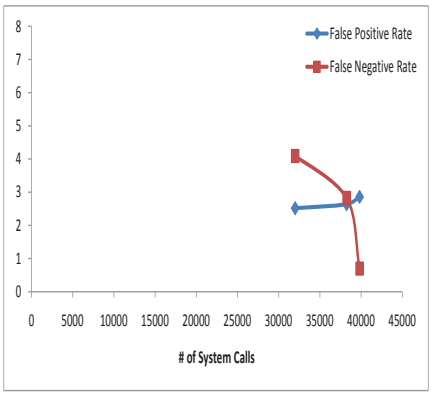
resulting in lower false negatives. But the number of false positives increases simultaneously as some of the new "hits" will result in false positives.

2. The impact of the training set size is not *monotonic* on both false positives and false negatives. Occasionally we see that when the training set is increased, the false negatives also increase (e.g., consider the graph for `grep` in Figure 4.14). Upon investigation, we realized that this is an artifact of our experimental methodology of choosing five random training sets and averaging the results across them. Some testcases have a higher marginal impact on model effectiveness (decrease false negatives more) than others. If a training set TS_i includes more high-impact testcases than a training set TS_j, then the false negatives of the models built using TS_i will be lower than those built using TS_j even if $|TS_i| < |TS_j|$. This effect is more pronounced in applications whose training data has a low number of system calls (e.g., `grep`).

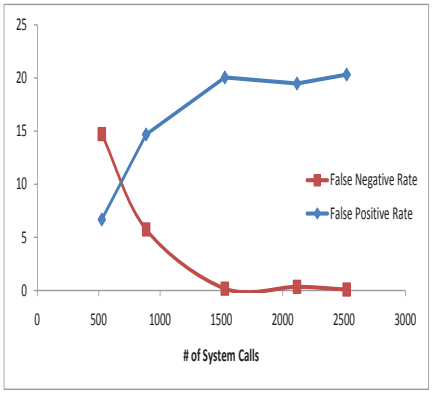
3. For many applications (e.g., `wc, ls`, the false negatives begin to level off when the size of the training set reaches a limit. However for other applications (e.g., `wget, cp` and `tar`), the false negatives continue to drop even as the maximum size of the training set is reached. This implies that, for those applications the coverage of the training data could be improved by adding additional test cases. In fact, some of the applications that suffer from the highest rate of false negatives in our causal models (`cp` and `tar` with 11.69% and 6.49% respectively) could potentially benefit from additional testcases to the training data.



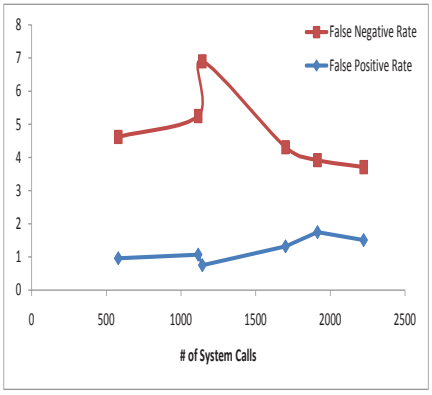
(a) GnuPG



(b) wget

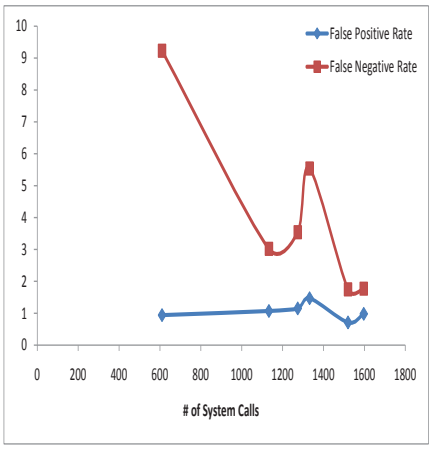


(c) ls

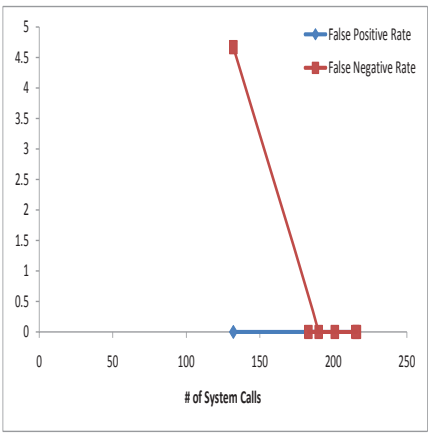


(d) find

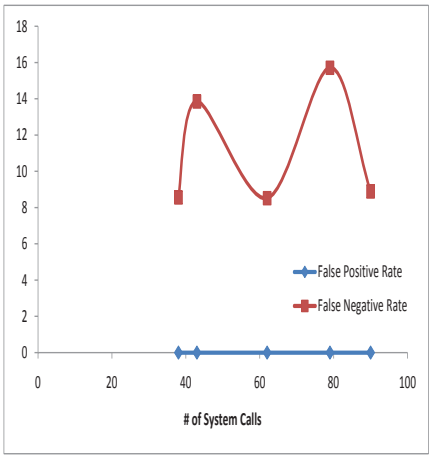
Figure 4.13.: Impact of the size of training data on the effectiveness of causal models. We list the results for the dynamic callstack model (the callsite and callstack models display the same trends) for the applications `GnuPG, wget, ls` and `find`. The X-axis refers to the size of training data in terms of the number of system calls.



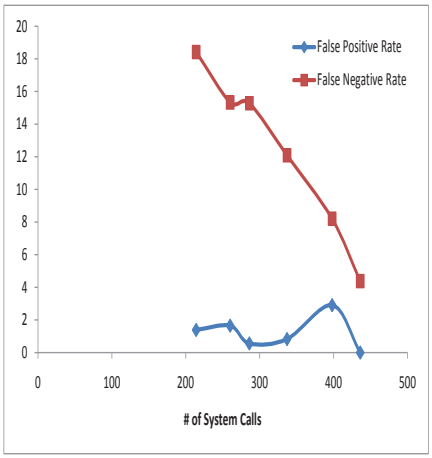
(a) gzip



(b) wc

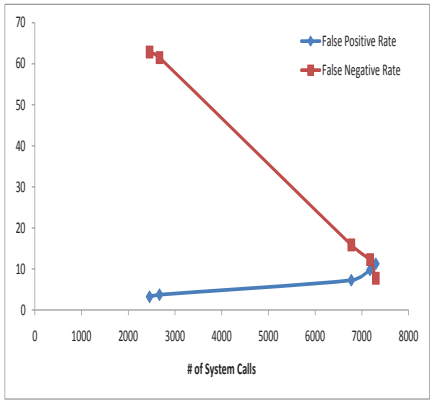


(c) grep



(d) cp

Figure 4.14.: Impact of the size of training data on the effectiveness of causal models. We list the results for the dynamic callstack model (the callsite and callstack models display the same trends) for the applications `gzip`, `wc`, `grep` and `cp`. The X-axis refers to the size of training data in terms of the number of system calls.



(a) tar

Figure 4.15.: Impact of the size of training data on the effectiveness of causal models. We list the results for the dynamic callstack model (the callsite and callstack models display the same trends) for the `tar`. The X-axis refers to the size of training data in terms of the number of system calls.

4.6.5 Runtime overhead

In this section we discuss the performance overhead of the different stages of our approach to causality determination:

1. Model building phase.
2. Online monitoring and audit log generation.
3. Analyzing the audit logs to determine causal relationships.

Model building

The model building phase can be broken down into two sub-phases: causal trace generation and the actual building of the models using the causal traces. Causal trace generation has exactly the same performance characteristics as that of generalized *DTA* as described in Section 3.5. Here we discuss the performance of the actual model building phase.

We present the runtime CPU overhead of model building in Table 4.6. The model building algorithms are implemented as `perl` scripts and the control-flow conditions are internally stored as text strings. We measure the CPU overhead using the `real` time spent by the model building programs for building each of our causal models. The `real` time is the total time taken for the process to complete (includes time spent by the processes in the `user` mode, in the `system` mode and time waiting for I/O completion).

We present the memory overhead of the model building program in Table 4.7. We measure the peak memory used by our model building `perl` scripts. We breakdown the memory usage into the baseline usage introduced by the perl interpreter for merely running our programs and the overhead introduced by model building. The measurements presented in Tables 4.6 and 4.7 represent averages taken across five different

Table 4.6: Runtime CPU overhead of building the callsite, callstack and dynamic callstack causal models from causal traces. The overhead is presented in terms of the `real` time in seconds consumed by the model building process for each of the models. The size of the causal trace is given in terms of number of system calls.

Application	Causal trace size	Callsite	Callstack	Dynamic Callstack
GnuPG	24996	11.92	13.73	955.43
wget	30163	96.36	97	7894
ls	967	0.5	0.53	7.79
find	1647	1.14	1.98	101.6
gzip	1071	0.86	0.91	23.15
wc	203	0.04	0.06	0.1
grep	56	0.1	0.19	0.11
cp	262	0.13	0.13	0.13
tar	818	0.51	0.53	3.15

runs of model building. All the runs were on a lightly loaded 2.8 GHz Pentium 4 Linux workstation with 512 MB RAM and 1 GB swap space.

In general, bigger the causal trace, the longer the model building takes. The callsite and the callstack models take roughly the same time to build for all the applications in our benchmark suite. The dynamic callstack model is the most expensive model to build, sometimes taking 80x more time than the callstack model (e.g., `wget`). The additional overhead is primarily due to our naive implementation of adding new clauses to the logical formulae in the causal models. Unsurprisingly the memory overhead of the causal models depends on the complexity of the respectively models – the more complex the model, the greater the overhead. Our model building implementation uses an inefficient text representation for storing the causal models. Potential for substantial memory savings exists if the causal models are stored using a compact binary representation.

Table 4.7: Memory overhead of building the callsite, callstack and dynamic callstack causal models from causal traces. The overhead is presented in units of KiB consumed by the model building process. The size of the causal trace is given in terms of number of system calls. The table contains the peak memory used up by the process during its lifetime, broken down into: (a) baseline memory usage when the program is loaded by the perl interpreter (b) memory usage due to model building and storage.

Application	Causal trace size	Baseline	Callsite	Callstack	Dynamic Callstack
GnuPG	24996	3596	7712	14120	28948
wget	30163	3596	8952	12708	14736
ls	967	3596	596	752	1040
find	1647	3592	1076	1692	2964
gzip	1071	3592	596	884	1176
wc	203	3596	272	380	392
grep	56	3596	848	1164	1480
cp	262	3592	444	572	852
tar	818	3592	1992	3020	4308

Online monitoring

We use Pin, the binary instrumentation tool [89] to implement the audit log generation mechanism. As Pin has a baseline overhead by itself and because the online monitoring can be implemented without Pin (using a combination of the operating system kernel instrumentation and custom binary instrumentation), we discard the overhead introduced by Pin in our results. In order to discard the overhead introduced by Pin, we estimate the overhead using a Pin extension that does nothing ("nullpin").

Table 4.8 lists the runtime overhead of generating the augmented audit logs that contain additional control-flow information. The callsite and callstack models increase the runtime of the applications being monitored by a modest average of 2.64% and 3.06% respectively. However the dynamic callstack model incurs a relatively high monitoring overhead of 47%.

In order to obtain the dynamic callstack, the instance information for each of the return addresses in the runtime stack at the time of system call execution has to be maintained. To achieve this, we instrument all the function call statements in a binary and track their instance information. This naive tracking of all callsites leads to the high overhead of the dynamic callstack model monitoring.

However, it is sufficient to track the instance information of only those callsites that are present in the dynamic callstacks of system calls. Tracking other callsites is unnecessary as those callsites are not present in the dynamic callstacks observed at system call execution time. Furthermore, it is sufficient to track those callsites that were present in the dynamic callstacks observing during the training phase. If a callsite is newly observed in a dynamic callstack during online monitoring, such a callstack will result in a "miss" during offline analysis and hence need not be tracked.

We obtain the smaller set of callsites from the causal traces used for model building and manually instrument them for instance tracking in the application binaries. For this version of the dynamic callstack monitor (henceforth referred to as "dynamic

callstack monitor (optimized)"), the runtime overhead drops to a respectable 4.66%. However as the optimization required custom instrumentation of the binaries of the applications, we report both the optimized and unoptimized results.

Finally, the memory overhead associated with all of our models was negligible when we discount the memory overhead associated with Pin. The low memory overhead is due to the fact that we do not store our causal models in-memory – our online monitor merely generates the control-flow properties and it is the offline analyzer that uses the causal models.

Table 4.8: Runtime overhead of online monitoring and audit log generation for the callsite, callstack and dynamic callstack causal models. The overhead is measured in terms of additional % of elapsed time taken by each application when the online monitoring and audit log generation is added.

Application	Callsite	Callstack	Dynamic Callstack	Dynamic Callstack (optimized)
GnuPG	8.3	10.4	87.6	11.2
wget	1	1	46.1	2.9
ls	3.2	3.5	26.2	4.5
find	1.5	1.7	43.5	3.5
gzip	1.4	1.8	37.3	3.3
wc	1	1.6	41.3	3.3
grep	1.5	1.6	50.4	3.7
cp	4.9	4.9	47.5	6.7
tar	1	1	43.1	2.81
Average	2.64	3.06	47	4.66

Audit log analysis

We implemented the causality determination algorithm described in Section 4.5 as a perl script that takes the audit logs and the causal models as input and outputs the list of causes for each system call in the audit log for a specified causal model. The analysis was performed in a lightly loaded 2.8 GHz Pentium 4 Linux workstation with 512 MB RAM and 1 GB swap space.

We present the CPU overhead of the log analysis program in Table 4.9. We use the `real` time consumed by the analysis process as the metric to measure CPU overhead. The analysis time is influenced both by the number of system calls in the audit logs and the size of the logical formulae expressing the control-flow conditions in a model: We observe that typically the bigger the audit log, the longer it takes to analyze the log. Similarly if the causal model contains more control-flow conditions (as is typically the case for the dynamic callstack model as it is more specific than the other two models), the analysis time is longer. Log analysis for `wget` takes the longest, 9180 seconds for analyzing 19076 system calls.

We present the memory overhead of the log analysis program in Table 4.10. The memory overhead is determined by the size of the causal model. The callsite model is the smallest of the three causal models, followed by the callstack model and the dynamic callstack model in that order.

Table 4.9: Runtime CPU overhead of analyzing audit logs and determining causality using the callsite, callstack and dynamic callstack causal models from causal traces. The overhead is presented in terms of the `real` time in seconds consumed by the causality determination programs. The size of the audit logs is given in terms of number of system calls

Application	Audit log size	Callsite	Callstack	Dynamic Callstack
GnuPG	20766	51.47	67.66	674.86
wget	19076	192.84	211.44	9180.78
ls	1969	5.84	5.97	63.67
find	955	1.36	1.75	18.85
gzip	704	0.93	1.04	6.81
wc	63	0.06	0.07	0.13
grep	45	0.11	0.1	0.21
cp	202	0.19	0.21	0.55
tar	6841	330.66	370.39	490.62

Table 4.10: Memory overhead of determining causality using the callsite, callstack and dynamic callstack causal models in audit logs. The overhead is presented in units of KiB consumed by the causality detection program. The table contains the peak memory used up by the program during its lifetime, broken down into: (a) baseline memory usage when the program is loaded by the perl interpreter (b) memory usage due to model storage and causality detection.

Application	Audit log size	Baseline	Callsite	Callstack	Dynamic Callstack
GnuPG	20766	3596	11440	22268	41484
wget	19076	4000	12468	17268	20576
ls	1969	3592	1124	1604	2140
find	955	3592	1444	2428	4068
gzip	704	3592	756	1332	1784
wc	63	3596	544	700	740
grep	45	3596	1164	1796	2228
cp	202	3592	576	704	1212
tar	904	3596	6972	9908	14168

4.7 Discussion

4.7.1 Signals

Unix processes have the ability to register routines to handle signals generated by the kernel [92]. Signal handling routines introduce asynchronous control flow that is typically not observed during program execution. Our causal models do not model the signal handling mechanism. Hence the presence of signal handling routines can increase both the false positives and false negatives generated by our causal models:

- When a signal is observed during the training phase, the system calls that are executed in the context of the signal handler result in the "loosening" of the control flow conditions due to the addition of disjunctional clauses to the logical formulae. This makes the causal model more general and permissive thereby increasing the false positives.
- When a signal is observed for the first time during the testing phase, the control-flow conditions associated with any system call executed within the signal handler will not be recognized by the causal model resulting in false negatives. One way to mitigate the false negatives is to take special care to include test cases that trigger signals and the signal handlers to be executed. Currently our test cases are not designed to generate and handle signals as part of their normal operation.

4.7.2 Multi-threaded applications

The way our causal models are currently implemented, they are agnostic to the presence of different threads of execution within a program. This could potentially result in false positives. For example, consider a causal relationship $open \xrightarrow{Cause} read$ that exists only if both the `open()` and `read()` calls are executed in the context of the

same thread. If during the testing phase `open()` and `read()` are executed in two different threads of control, but with the rest of the control-flow conditions the same as that of the training phase, our causal models will falsely implicate `open()` as a cause of `read()`. Our models have to become thread-aware to overcome this limitation.

4.7.3 Address space layout randomization

Randomizing the address-space layout of a software program, known as ASLR (Address Space Layout Randomization) [93] is a popular technique used to prevent attackers from using the same exploit code effectively against all instantiations of the program containing the same flaw. The attacker must either craft a specific exploit for each instance of a randomized program or perform brute force attacks to guess the address-space layout. A popular implementation of ASLR for Linux is PaX [94] which randomizes the base address of the stack, heap, code, and `mmap()`ed segments of ELF executables and dynamic libraries at load and link time.

As a result of the address space randomization, the addresses used in the `text` segment of a program will vary from one instantiation of the program to another. This could make the causal models useless if the control-flow properties (such the callsite and return address values) found in the training phase are used as is. Hence we propose the following: During the training phase, the random value that is added to the base address of memory segments is also logged and is used to derive the "relative" addresses of the control-flow properties. Similarly the online monitor would log the random value for each program instantiation in the audit log which can be then used to derive the relative addresses of the control-flow properties in the audit log.

4.7.4 Dynamically linked libraries

For a statically linked application, addresses of the `libc` wrappers around system calls is fixed at compile/link time. However for dynamically linked applications, the

program counter values of system calls (and hence the callsite values of the system calls) will vary from one system to the other. This makes it difficult to use the program counter values of system call invocations during the training phase directly in our causal models. To overcome this challenge, we use the program counter value of the `call` instruction into the `.plt` (procedure linkage table) section [95] of the binary as a proxy for the program counter value of the actual system call. This makes our causal models independent of dynamically linked libraries.

4.7.5 Control-flow modification and code-injection attacks

Our approach works very well when the control-flow of the program is consistent between the training phase and the deployment. However when a program is subject to attacks that modify its control-flow (e.g., buffer overflows, format string attacks) it will exhibit control-flow that is not captured in the causal model. This results in false-negatives. One way to mitigate this limitation is to fall back to a "happened-before" model of causality – we conservatively assume that all previously executed system calls were causes. Alternatively, we could fall-back to a Backtracker [35] or the static slicing model.

4.7.6 Unknown applications

Our approach relies on the availability of the application binary a-priori. False-negatives result when faced with a binary for which a causal model is not available. This situation could arise when an attacker downloads malware after gaining privileged access to a system. Similar to control-flow modification attacks, we propose to fall-back to either a "happened-before" model of causality or a Backtracker or static slicing model. An interesting avenue of research is to develop causal models that are resilient to obfuscation and slight variation in the control-flow. Obfuscation resilient

models could be built based on known variants of malware and those models could be used to detect causality in any future variation of the malware.

4.7.7 Causality through data-flow

In our experimental evaluation, we noticed that most of the false-positives of our causal models arose while predicting causality that is at least partially enabled through the data-flow of a program. The example of `ls` discussion in Section 3.5 illustrates this issue. This is an inherent limitation to any approach that considers only the control-flow of a program. Extending our causal models to selectively track data flow to obtain clues about causality is another interesting opportunity for future research.

4.7.8 Improving false negative rate

False-negatives occur in our approach when a control-flow property is encountered during deployment that was not encountered during the training phase. The rate of false-negatives depends on the completeness of our training test cases. One way to improve the coverage of the test cases is to leverage the testcases used for testing an application during its development lifecycle. Software vendors could release their test cases (or better the causal models themselves) along with the software. Another way to improve the coverage of the testcases is to continuously sample a stream of real world inputs similar to the cooperative bug isolation technique proposed by Liblit [96].

4.7.9 Causality modeling

All the existing approaches for causality determination take a "binary" approach to causality. Either a system call is a cause or not. There is no notion of how "strong" the impact of a cause is on an effect. This limitation arises from using program dependence as a proxy for causality. Using program dependences does not capture

all the dimensions of causality. While it captures the notion of a "necessary" cause, it ignores the aspect of the "sufficiency" of a cause [3, 4].

Sufficiency. How sufficient is a cause for the production of an effect? It is a measure of the ability of a cause to produce an effect in situations where the effect is actually absent. The measure of sufficiency is important especially in cases where there are multiple events that are equally necessary to produce an effect. Consider an example where an intruder exploited the *crackaddr* vulnerability [2] present in `sendmail` resulting in a root shell being spawned. The *crackaddr* vulnerability can be successfully exploited only in a few operating systems e.g., `Slackware 8.0` [2]. Traditional causation would identify many causes: the attacker actually launching the attack, the presence of crackaddr vulnerability and the presence of `Slackware 8.0`. It does not discriminate or rank the causes. In some cases, it is reasonable that the spawning of the root shell is more attributable to the actions of the attacker than say the presence of Slackware 8.0. Sufficient causation helps capture precisely this notion. It helps in ranking the necessary causes, if more than one were responsible for a particular effect. Other researches have explored using alternate notions of causality such as *channel capacity* to quantitatively measure causality. Studying ways to quantify causality and developing new techniques to practically quantify the same is an important area of future research.

4.8 Conclusion

Past approaches for determining causality have either had high fidelity or low overhead, but seldom both. We propose a practical approach for tracking causality that has both the properties. Our approach has a very low rate of false-positives (4.45%) and false-negatives (4.93%). And it is suitable for practical deployment as it has a very low CPU overhead (4.66%). We believe that our approach is an attractive addition to the toolkit of causality determination mechanisms employed by intrustion analysis and forensics systems.

5 CONCLUSIONS AND FUTURE WORK

This dissertation builds evidence to support the thesis that it is possible to practically and automatically determine causal relationships between system calls in software execution traces. This chapter summarizes our conclusions, contributions and provides directions for future work.

5.1 Conclusions and contributions

This dissertation makes the following contributions:

- We empirically study the effectiveness of existing approaches for causality determination in event reconstruction systems. As part of this study:
  - We develop a systematic approach for evaluating the effectiveness of causality determination techniques.
  - We develop a suite of real world applications and testcases for benchmarking the effectiveness of causality determination. The suite allows us to identify the source of inaccuracy and performance overhead of the various causality determination techniques that we study.
  - Using our approach, we provide experimental data quantifying the accuracy and the overhead (time, space, memory) of each technique.
  - We conclude that generalized DTA, while being the most accurate technique to determine causal relationships, suffers from a high CPU overhead and is impractical to be deployed widely.

  - We conclude that the rate of false positives is very high for all the techniques (BackTracker and static slicing) that we evaluate, sometimes as high as 96%. This could have legal ramifications (Trojan Horse Defense) and highlights the need for more accurate techniques.
  - We analyze the experimental data and shed light on the conditions that lead to the inaccuracies and the overhead of the techniques we evaluate. For example, we found that BackTracker and the static slicing techniques do not work well in applications that exhibit recursive and iterative workflow characteristics.

- Based on the insights that we gain from our empirical study, we describe a new approach to causality determination:
  - Our approach involves developing a "causality prediction model" to determine causal relationships based on observations of control-flow of a program. Our approaches provides efficient and accurate causality determination when the following conditions are met: (1) The program was not subject to control-flow modification or code-injection attacks and (2) The executable code of the program is available apriori.
  - Experimental evaluation of our new approach shows that the causality determination through control-flow monitoring has a low false-positive rate (4.5%), a low false-negative rate (4.93%) and a low runtime overhead (4.66%).

- Finally, we analyze the experimental data from our evaluation and provide insights on improving the accuracy of causality determination even more. Specifically we note how recursive workloads of programs limits the accuracy of purely control-flow based causal models.

5.2 Future work

There are several interesting dimensions in which our work can be extended:

5.2.1 Increasing coverage

Additional intrusion analysis systems

An empirical study of causality determination techniques has to periodically updated as newer techniques are proposed and deployed. For example, virtual machine introspection is a promising new technique for logging event information (Krishnan et al. [46]). While we can qualitatively argue that this approach is more precise than BackTracker but less precise than dynamic slicing, it would be helpful to have the actual metrics and performance overhead.

Additional operating systems

One of the goals of our empirical study of causality determination schemes and our causal models is to provide reliability metrics for the intrusion analysis and digital forensics community. Our study was done in Linux with a benchmark consisting of applications used in Unix-like operating systems. However, intrusion analysts and digital investigators have to deal with a heterogeneous set of operating systems in addition to Unix-like systems (e.g., Windows and its many flavors, OS X). With the advent of smart phones, the landscape of operating systems has become even more diverse (iOS, WebOS, Android etc.,). Extending our empirical study to provide reliability metrics on causality determination techniques in other operating systems is an important and interesting challenge.

The first challenge is to survey the set of event reconstruction systems used in those operating environments and the causality determination techniques employed by those

systems. Second, we need to customize our benchmarking suite for each operating system by adding applications typically used in that operating system. Finally, we need to evaluate the effectiveness of the existing causality determination techniques to obtain the reliability metrics.

5.2.2 Improving causality determination accuracy

As we discussed in chapters 3 and 4, both existing and our newly proposed causality determination techniques struggle in the face of "recursive" workloads. Our current models do not sufficiently model the program semantics in the face of recursive workloads.

One potential way to improve the accuracy of our models is to observe additional control-flow. Consider the source code listed in Figure 4.5 from Chapter 4. The `if` condition in line 18 provides a clue as to if the `readdir()` call in line 16 is involved in a recursive causal relationship. If the `if` branch is taken, then `readdir()` will be involved in a recursive relationship, otherwise not. Identifying additional control-flow properties that can be observed to increase accuracy, but without impacting runtime performance is an interesting avenue for research.

Another potential way to improve model accuracy is to observe data-flow of the program. Observing data-flow increases both the runtime overhead and the size of the audit logs. Care must to taken to identify the subset of data-flow that increases model accuracy but does not impact runtime overhead and audit log size. Automatically identifying such a subset is an interesting research problem.

5.2.3 Alternate notions of causality

As previously mentioned in Chapter 4, program dependences are not an exact measure of causality. While program dependences capture the aspect of *necessary* causes, they fail to inform how *sufficient* the causes are. A richer notion of causality is needed

to capture the sufficiency of a cause. *Channel capacity* has been proposed as an interesting alternative to quantify the influence a cause has over an effect in program execution traces [90]. Exploring such alternate measures of causality and developing practical techniques for their measurement is a future opportunity for research.

LIST OF REFERENCES

LIST OF REFERENCES

[1] 2010 E-Crime watch survey. Available at: `http://www.cert.org/archive/pdf/ecrimesummary10.pdf`

[2] SANS critical vulnerability analysis Vol. 2. No. 9. March 10, 2003. Available at: `http://www.sans.org/newsletters/cva/vol2_9.php`

[3] Judea Pearl. Reasoning with cause and effect. In *Proceedings of the International Joint Conference on Artificial Intelligence,* San Francisco, Morgan Kaufman, pages 1437–1449. 1999.

[4] Judea Pearl. *Causality: Models, reasoning, and inference.* Cambridge: Cambridge University Press. 2000.

[5] D. Hume. *An enquiry concerning human understanding.* 1748.

[6] J.S. Mill. *System of logic.* Volume 1. John W. Parker, London, 1843.

[7] D. Lewis. *Counterfactuals.* Oxford: Blackwell. 1973.

[8] D. Lewis. *Philosophical papers: Volume II.* Oxford: Oxford University Press. 1986.

[9] H. Reichenbach. *The direction of time.* University of California Press, Berkeley and Los Angeles, 1956.

[10] P. Suppes. *A probabilistic theory of causality.* Amsterdam: North-Holland Publishing Company. 1970.

[11] P. Sprites, C. Glymour, and R. Scheines. *Causation, prediction and search.* Second edition. Cambridge, MA: M.I.T. Press. 2000.

[12] J. P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, April 1980.

[13] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for UNIX processes. In *Proceedings of 1996 IEEE Symposium on Security and Privacy,* 1996.

[14] C. Ko. *Execution monitoring of security-critical programs in distributed systems: A specification-based approach.* PhD thesis, U.C. Davis, September 1996.

[15] T. Lane and C. E. Brodley. Temporal sequence learning and data reduction for anomaly detection. *ACM Transactions on Information and System Security,* 2(3):295 331, 1999.

[16] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. *11th Annual Network and Distributed Systems Security Symposium,* 2004.

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

[18] M. Bernaschi, E. Gabrielli, and L. V. Mancini. Operating system enhancements to prevent the misue of system calls. In *Proceedings of the Seventh ACM Conference on Computer and Communications Security,* pages 174–183, Athens, Greece.

[19] J. Zimmermann, L. Mi and C. Bidan. Experimenting with a policy-based HIDS based on an information flow control model. In *Proceedings of the 19th Annual Computer Security Applications Conference,* 2003.

[20] J. Zimmermann, L. Mi and C. Bidan. An improved reference flow control model for policy-based intrusion detection. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS),* 2003.

[21] C. Ko and T. Redmond. Non-interference and intrusion detection. *IEEE Symposium on Security and Privacy,* May 2002.

[22] J. Zimmermann, L. Mi and C. Bidan. Introducing reference flow control for detecting intrusion at the os level. In *Proceedings of the Fifth International Symposium on Recent Advances in Intrusion Detection,* pages 292–306, October 2002.

[23] D. Wagner and D. Dean. Intrusion detection via static analysis. *IEEE Symposium on Security & Privacy,* 2001.

[24] A. Hdtdld, C. Sdrs, R. Addams-Moring and T. Virtanen. Event data exchange and intrusion alert correlation in heterogeneous networks. In *Proceedings of the Eigth Colloquium for Information Systems Security Education,* West Point, NY, June 2004

[25] P. Ning, Y. Cui and D. S. Reeves. Analysing intensive intrusion alerts via correlation. In *Proceedings of Recent Advances in Intrusion Detection 2002,* Lecture Notes in Computer Science 2516, pages 74–94; Springer-Verlag; 2002.

[26] A. Valdes and K. Skinner. Probabilistic alert correlation. In *Proceedings of Recent Advances in Intrusion Detection 2001,* Lecture Notes in Computer Science 2212, pages 54–68, Springer-Verlag; 2001.

[27] F. Cuppens and A. Miige. Alert correlation in a cooperative intrusion detection framework. *IEEE Symposium on Security and Privacy,* May 2002.

[28] X. Qin and W. Lee. Statistical causality of INFOSEC alert data. In *Proceedings of Recent Advances in Intrusion Detection 2003,* Lecture Notes in Computer Science 2820, pages 73–94; Springer-Verlag; 2003.

[29] P. Ning, Y. Cui and D. S. Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *Proceedings of the Ninth ACM Conference on Computer and Communications Security,* 2002.

[30] The sleuth kit. `http://www.sleuthkit.org`

[31] The coroners toolkit. `http://www.porcupine.org/forensics/tct.html`

[32] Buchholz Florian, Falk Courtney. Design and implementation of Zeitline: a forensic timeline editor. *Digital Forensics Research Workshop* (2005).

[33] Wireshark. `http://www.wireshark.org/`

[34] Guidance EnCase. `http://www.guidancesoftware.com/`

[35] King, S. T and Chen, P. M. Backtracking intrusions. In *Proceedings of the 2003 Symposium on Operating Systems (SOSP)* (October 2003).

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

[37] G.W. Dunlap, S.T. King, S. Cinar, M.A. Basrai, and P.M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.

[38] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer*, pages 34-45, June 1974.

[39] Ashvin Goel, Wu-chang Feng, David Maier, Wu-chi Feng, Jonathan Walpole. Forensix: a robust, high-performance reconstruction system. In *Distributed Computing Systems Workshops* (2005).

[40] Purdie L, Cora G. SNARE system iNtrusion analysis & reporting environment. `http://www.intersectalliance.com/projects/Snare/.`

[41] Sitaraman S, Venkatesan S. Forensic analysis of file system intrusions using improved Backtracking. In *Third IEEE international workshop on information assurance (IWIA05)* (2005).

[42] Buchholz F, Shields C. Providing process origin information to aid in computer forensic investigations. Technical report, CERIAS TR 2004-48, 2004.

[43] Xuxian Jiang, AAron Walters, Florian Buchholz, Dongyan Xu, Yi-Min Wang, Eugene H. Spafford. Provenance-aware tracing of worm break-in and contaminations: A process coloring approach. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS 2006)* (July 2006).

[44] Florian Buchholz. Pervasive binding of labels to system processes. PhD Dissertation. Purdue University (2005).

[45] Sarmoria Christian G, Chapin Steve J. Monitoring access to shared memory-mapped files. *Digital Forensics Research Workshop*. 2005.

[46] Srinivas Krishnan, Kevin Snow, Fabian Monrose. Trail of bytes: efficient support for forensic analysis. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)* (2010).

[47] Brenner Susan, Carrier Brian, Henninger Jef. The Trojan defense in cybercrime cases. *Santa Clara Computer and High Technology Law Journal 2004;21(1).*

[48] B. D. Carrier and E. H. Spafford. Defining event reconstruction of a digital crime scene. *Journal of Forensic Sciences,* 49(6), 2004.

[49] B. D. Carrier and E. H. Spafford. An event-based digital forensic investigation framework. In *Proceedings of the 2004 Digital Forensic Research Workshop,* 2004.

[50] F. Buchholz and E. H. Spafford. On the role of file system metadata in digital forensics. Technical Report, CERIAS TR 2004-56, 2004.

[51] F. Buchholz and C. Shields. Providing process origin information to aid in computer forensic investigations. Technical Report, CERIAS TR 2004-48, 2004.

[52] Brian D. Carrier. A hypothesis-based approach to digital forensic investigations. PhD Dissertation. Purdue University (2006).

[53] Palmer Gary. A road map for digital forensic research. *Digital Forensics Research Workshop,* 2001.

[54] M. J. Ranum. Experiences Benchmarking Intrusion Detection Systems. NFR Security White Paper, December, 2001.

[55] CodeSurfer, Available at: `http://www.grammatech.com/products/codesurfer/index.html`.

[56] Sundararaman Jeyaraman, Mikhail J. Atallah. An empirical study of automatic event reconstruction systems. In *Digital Forensics Research Workshop,* (2005).

[57] Edward J. Schwartz, Thanassis Avgerinos, David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy* (2010).

[58] M. Weiser. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. PhD thesis, University of Michigan, Ann Arbor, 1979.

[59] Zhang X, Gupta R, Zhang Y. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering* (2003).

[60] Binkley David, Harman Mark. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *Proceedings of the 19th IEEE International Conference on Software Maintenance* (2003).

[61] D. Bell and L. LaPadula. Secure computer systems: mathematical foundations and model. *MITRE Report MTR 2547 v2*, 1973.

[62] K. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE Corporation, Bedford, MA, 1977.

[63] D. Brewer and M. Nash. The chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*. May 1989.

[64] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications,* 21(1), 2003.

[65] C. Bodei, P. Degano, H. Riis Nielson, and F. Nielson. Security analysis using flow logics. In *Current Trends in Theoretical Computer Science,* G. Paun, G. Rozenberg, and A. Salomaa, Eds., pages 525–542. World Scientific, 2000.

[66] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM,* vol. 20, no. 7, pages 504–513, July 1977.

[67] D. E. Denning. *Cryptography and Data Security.* Addison-Wesley, 1982.

[68] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)* December, 2004.

[69] G. E. Suh, J. W. Lee, D. Zhang and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems,* pages 85–96, 2004.

[70] Y. Beres and C. I. Dalton, Dynamic label binding at run-time. In *Proceedings of the 2003 Workshop on New Security Paradigms,* pages 39–46, 2003.

[71] Sean Peiset, Matt Bishop and Keith Marzullo. Computer forensics in forensis. *ACM Operating System Review 42* (2008).

[72] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium,* 2004.

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

[74] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis* (2007).

[75] Walter Chang and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the ACM Conference on Computer and Communications Security, pages 39-50* (2008)

[76] Feng Qin, Cheng Wang, Zhenmin Li, Ho-Seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM Symposium on Microarchitecture* (2006).

[77] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium, pages 121-136* (2006).

[78] L. C. Lam and T.-C. Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference, pages 463-472* (2006).

[79] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture, pages 221-232* (2004).

[80] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th International Symposium on Computer Architecture, pages 482-493* (2007).

[81] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 85-96* (2004).

[82] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security, 6(3):151180* (1998).

[83] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy* (2001).

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

[85] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy* (2003).

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

[87] Rajeev Gopalakrishna, Eugene Spafford, and Jan Vitek. Efficient intrusion detection using automaton inlining. In *Proceedings of the IEEE Symposium on Security and Privacy* (2005).

[88] Department of Defense. Orange book summary (TCSEC). *Trusted Computer System Evaluation Criteria, DOD 5200.28 STD* (1985).

[89] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005), pages 190-200* (2005).

[90] Newsome, J., Mccamant, S., and Song, D. Measuring channel capacity to distinguish undue influence. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* (2009).

[91] C. J. Van Rijsbergen. Information retrieval. Butterworth-Heinemann, Newton, MA, 1979.

[92] M. Bach. The design of the UNIX operating system. Prentice Hall, ISBN 0-13-201799-7.

[93] Hovav Shacham, Mattew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address space randomization. In *ACM Computer and Communication Security Symposium* 2004.

[94] T. Durden. Bypassing PaX ASLR protection. *Phrack Magazine*, 59(9), June 2002.

[95] System V application binary interface. Edition 4.1 (1997-03-18).

[96] Ben Liblit. Cooperative bug isolation. PhD Dissertation. University of California, Berkeley. 2005.

VITA

VITA

Sundararaman Jeyaraman obtained his Bachelors in Engineering (B.E.) degree from the College of Engineering Guindy, Anna University, Chennai, India in 2000. He received the M.S. and Ph.D degrees in Computer Sciences from Purdue University in 2007 and 2011 respectively. His interests are information assurance, systems security, network security and WAN optimization. He is currently working at Cisco Systems, San Jose, California, where he is learning how to engineer complex systems.