

Practical Automatic Determination of Causal Relationships in Software Execution Traces

Sundar Jeyaraman
jsr@cs.purdue.edu

May 3, 2005

1 Introduction

The goal of this thesis is to propose, develop, investigate and evaluate techniques that enable practical determination of causal relationships between system events. From the forensics expert who needs to analyse the origins of a trojan (“who was responsible for the creation of this file?”) to the system administrator that needs to analyse an intrusion (“how did the intruder break in?”), security experts have to frequently answer questions about the cause-effect relationships between the various events that occur in a computer system. Unfortunately, causal relationships between events are poorly defined and understood in today’s computer systems. Even simple questions about causality are often surprisingly difficult to answer. In this work, I will:

- Shed more light on the nature of causal relationships between system events. Most of the current difficulties stem from a poor understanding of the meaning of causality. I provide a comprehensive definition that captures most of the aspects of causality.
- Investigate the challenges in determining causal relationships. Specifically, I point out how the treatment of processes as black boxes has forced system administrators to make very conservative assumptions about causal relationships, leading to highly imprecise answers to causal queries.
- Propose and develop *practical* techniques to determine causal relationships. I take a two-step approach:
 1. Program-dependences are syntactic approximations of causal relationships. I propose a profile-driven approach for practical determination of program-dependence relationships between system events from program execution traces. The accuracy of the profile-driven approach is tunable and the approach lends itself naturally for studying the tradeoffs between precision and the overhead involved.
 2. Next, using the intuition and experience gained from detecting dependence relationships, I will develop practical techniques to capture the causal relationships between system events.

1.1 Organisation of the paper

I explain the background concepts needed in section 2, and formally state my thesis in section 3. In section 4, I discuss how the ability to practically determine causal relationships significantly helps in furthering the state-of-art in computer security. Then I point out the difficulties in determining causal relationships and explain the inadequacies of related work. In section 5, I propose my approach and discuss the issues and challenges that we are likely to face in achieving my goal. In section 6, I present a timeline for the work to be done towards realising my goal.

2 Background

In this section, I discuss the background concepts that need to be defined so that my thesis statement can be made clearly. Purely for expository purposes, I restrict the discussion to computer systems that run Unix-like operating systems. This is not a fundamental limitation. The concepts and the approaches described in this paper should be applicable to other systems with only minor modifications.

2.1 System Event

For the purpose of this paper, I consider an event to be an *action* that is performed by a process on behalf of an user. In this work, I observe system events at the granularity of *system calls* since from the point of view of an attacker, most of the “interesting” actions that happen in a host can be composed in terms of system calls. Henceforth, the terms *system event*, *computer event*, *event* and *system call* are used interchangeably.

2.2 Causal Relationships between events

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., Counterfactual theories, Regular theories [25, 33, 34, 46, 48, 50]. None of those theories has been found completely satisfactory and the exact semantics of causation has eluded the researchers despite an enormous body of related research conducted in the fields of philosophy, econometrics, epidemiology, statistics and other fields. From previous work in the study of causality, I adopt precisely those theories that would help in understanding and explaining causal relationships between computer events. Specifically, I adopt the following measures of causality from Pearl et. al. [42, 43]:

1. Necessity

This is the most intuitive aspect of causality and is commonly expressed in terms of the following counterfactual query: would E (effect) have occurred if it were not for C (cause)? Intuitively, necessary causation tries to capture the *influence* a *cause* has over an *effect*. In other words, how much *dependent* is E on C ?. An example counterfactual query using system events: would the *root kit be still installed* (effect), if it were not for the *email received by the email-server*(cause)?¹ However, necessary causation does not capture all the dimensions of causality. In particular, it ignores the aspect of “sufficiency” of a cause.

¹The 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 a system call that received the corresponding network packets.

2. 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 the example from the previous paragraph. Assume that the intruder in question exploited the *crackaddr* vulnerability present in *sendmail* resulting in a root shell being spawned. *crackaddr* vulnerability can be successfully exploited only in a few operating systems e.g., Slackware 8.0. [61]. Necessary 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.

2.3 Causal Mechanisms

Causal relationships between events are enabled by *function mechanisms*. For example, a user *Alice* deletes a file *foo*, using the file-system related system call *unlink*. Here, the inner workings of *unlink* enables *alice* to delete *foo*. Broadly, causal relationships between system events are enabled by two types of functional mechanisms:

1. Operating System (OS):

Cause-effect relationships between system events could be enabled through various subsystems of the operating system e.g., the file system, the IPC system. Consider the following example where process-1 and process-2 execute a sequence of system calls in the specified order: The

```
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: OS relationship example

read system call of process 2 is causally dependent on the *write* system call of process 1, since the data that is “used” by *read* is dependent on the data “produced” by *write*. In this case, the causal relationship is enabled by the file-system component of the OS. I refer to such relationships as **OS-enabled** causal relationships.

2. Process address space (PAS):

For two events executed by the same process, the process address space (both code and data) could enable a cause-effect relationship between them. For example, the *write* system call is dependent on the *read* system call in the piece of code in figure 2:

The causal relationship between the *read* and the *write* is enabled by the *strncpy* library call. I refer to such relationships as **PAS-enabled** or simply *PAS* causal relationships.

```
fd_r = open("foo", O_RDONLY);
fd_w = open("bar", O_WRONLY);
read(fd_r, buffer, 10);
strncpy(dest, buffer, 10);
write(fd_w, dest, 5);
```

Figure 2: PAS relationship example

3 Thesis statement

For my thesis, I will work towards determining the feasibility of the following statement:

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

4 Motivation

In this section, I first discuss the significance of causal relationships in general. Then, I discuss the difficulty in estimating causal relationships.

4.1 Why study causal relationships?

The question of how the various events that occur in a computer system are causally related to each other arises frequently in a variety of contexts in information security. In this section, we discuss a series of fields that have been considered as vital for securing the computing infrastructure. We describe how some of the fundamental and significant questions that arise in those fields are causal in nature.

Intrusion Analysis, Forensic analysis. The number of security incidents and intrusions have been rapidly on the rise over the past few years. Given that it is difficult to completely secure computing infrastructure, and the heavy financial loss inflicted by intrusions [62], 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 analyse the events related to the intrusions and answer the following questions in a timely and efficient manner:

- How did the intruder gain access to the system? e.g., the vulnerability that was exploited.
- What was the damage inflicted by the intruder? For example:
 - Did the intruder install a root kit?
 - Did the attacker steal any proprietary information?
 - Was the compromised system used as a stepping-stone in attacking other systems in the organisation?

Questions of similar nature also arise when a digital forensic expert examines digital evidence [8, 11]. Collectively, answering such questions has been referred to as the process of “event reconstruction” [10]. Current attempts at reconstructing the chain of events related to an intrusion are still in their infancy [16, 26]. Event reconstruction is simplified with the knowledge of causal relationships between events: knowledge of causal relationships would allow accurate “back track”ing from the effect to the cause (or causes) and “forward track”ing from a cause to all its effects.

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) misuse-detection systems (b) anomaly-detection systems. Misuse-detection systems require a rule-base, based on which they determine if the ongoing activities in a system constitute an attack or not [5]. Anomaly-detection systems build a model of normal behaviour of the system and flag any anomalous behaviour as a possible attack [2, 17, 19, 20, 28, 31]. Both types of systems have some fundamental limitations that limit their usefulness and practicability. Misuse 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 very high false-alarm rate.

Recently, “policy-based” intrusion detection systems have been proposed as a promising alternative to both anomaly and misuse detection systems [29, 58–60]. 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 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.

Intrusion Alert Correlation. A security conscious organisation 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 misuse-based). Therefore, the system administrators of an organisation are typically overwhelmed with a profusion of intrusion-alerts emanating from diverse detection systems. Correlating alerts [40, 45, 53] from such heterogenous 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 recognising 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 [40, 41, 45]. 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 afore-mentioned functions. In fact, work by King et. al. [27] have found that enriching intrusion alerts with some contextual information based on even a very simple notion of causality is quite valuable.

4.2 Practical estimation of PAS-relationships is difficult

1. Unobserved functional mechanisms:

Unlike OS-enabled relationships, it is difficult to determine if causal relationships have been enabled by the process address space. For example, the *write* system call is dependent on the *read* system call in the piece of code in figure 2. The causal relationship between the *read* and the *write* is enabled by the *strncpy* library call. Such relationships are difficult to determine as the functional mechanism (the instructions executed by the process) goes unobserved by the audit logging mechanisms usually because of the huge space and time overhead involved. Because of this difficulty, current intrusion analysis and forensics systems [9, 26] have had to make very conservative assumptions that make them extremely imprecise.

2. Inability to reason about Causal Semantics:

Even if we assume that it is feasible to observe all the instructions executed by the process, it is not clear how to reason about causal semantics in the execution trace. As far as we know, there has been no work done in developing a logic or calculus for causal reasoning in program execution traces.

4.3 Previous and related work

The field that is most closely related to our work is that of information flow control. Information flow control policies regulate the way information flows through a computer system. For example, a confidentiality policy might specify the set of users that are allowed to access a particular information. Most of the research in this field has focused on developing language-based approaches for detecting information flow policy violations [47]. Given the source code of a program, these approaches try to determine if it satisfies a given information flow policy. However, such language-based information-flow control approaches might not be suitable in determining causal relationships owing to the following reasons:

1. Encoding causality using information flow:

Qualitative information flow policies like non-interference capture some limited aspects of causality. For example, if two events are non-interfering [21], then they are not causally related. Quantitative information flow policies [35, 37, 44] *could* capture the necessity aspect of causation. However, it is unclear, if in general, it is possible to encode all aspects of causality using information flow semantics.

2. Lack of precision:

Even if it were possible to encode some limited aspects of causality in terms of information flow policies, the inherently conservative nature of the language-based approaches might render them too imprecise to be of any practical use. We are not aware of any study that measures the precision of the various type-system based information flow control systems. However, it is known that classical data and control-flow analysis techniques are well suited for detecting information flow in programs [14, 15] and they could be potentially more precise than the type-system based approaches [7]. Further, the imprecision of classical static-analysis is well documented (see discussion on *static slicing* in section 5.1.3). Hence it is reasonable to assume that type-system based approaches too suffer from a similar lack of precision.

3. Lack of support for current applications.

Current language-based approaches mandate that the applications be written in special programming languages. It is doubtful if the type-system based techniques can be applied to applications developed in low-level languages like C or binary code in a straight forward fashion.

Dynamic information flow control mechanisms [4, 49, 52] 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 [49, 52]. 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 to develop have a very low overhead without significantly sacrificing precision.

5 Proposed Research

Instead of directly attempting to solve the problem of practical determination of PAS causal relationships, I take a two-step approach. First I try to solve a related and potentially easier problem

– practical determination of program dependences [1]. The intuition behind this decision is:

1. Program dependences (data and control 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. Moreover, they have been extensively used as approximations in other related fields like *Information Flow* for a considerable period of time [7, 14, 15]. It would be interesting to see how effective program dependences are as surrogates of PAS causal relationships.
2. Program Dependence relationships have been studied extensively by researchers in the Programming languages and the software engineering communities. As a result, a vast array of techniques and tools are available to reason about dependence relationships. In comparison, there is very little previous work done on causal relationships.

By first developing practical techniques to detect dependence relationships, I will develop more intuition and insight about the issues that I am likely to face in determining causal relationships.

5.1 Practical Determination of Program Dependences

5.1.1 Background

For the purpose of my work, I consider two types of dependence relationships between the statements in a program [18, 24]:

1. **Data Dependence:** A statement S_1 of program P is said to be data dependent on another statement S_2 if S_2 “modifies” the value of some variable x that is “used” by S_1 and there exists a control flow path from S_2 to S_1 where x is not “modified” again.
2. **Control Dependence:** A statement S_1 is said to be control dependent on another statement S_2 if S_2 determines if S_1 is executed or not.

Program Slicing [24, 51, 56] is the traditional programming language technique used to identify dependence relationships between various program statements.

Program slice: A *program slice* consists of the parts of a program that (potentially) affect/influence the values computed at some point of interest, referred to as a *slicing criterion*. Typically, a slicing criterion consists of a pair (line-number, variable). The parts of a program which *might* have a direct or indirect effect on the values computed at a slicing criterion C are called the *program slice with respect to criterion C*. The task of computing program slices is called *program slicing* [56]. A statically computed program slice is also known as a *static slice*.

Dynamic slice: A *dynamic slice* consists only of those parts of a program that affect the values computed at the *slicing criterion* for a *particular execution* of the program. The task of computing dynamic slices is called *dynamic slicing* [1, 30].

5.1.2 Inadequacies of existing work

The slicing techniques that have been developed so far were developed with applications in program understanding, maintenance, debugging, optimization etc., in mind. The techniques are either too imprecise, or are not efficient enough to detect dependences at run-time with minimal overhead:

1. Static slicing is known to produce very imprecise results because of the highly conservative nature of data-flow analysis and alias analysis techniques [6]. It means that when applied to detecting dependency relationships between system events, static slicing is likely to indicate that any given system event is dependent on most other system events with high probability.
2. Dynamic slicing on the other hand could provide precise answers to dependence queries. Dynamic slices are known to be much smaller in size when compared to the static slices [54, 57]. But, computing precise slices involves storing trace information about every instruction executed by the program. This imposes both a memory and time overhead that makes dynamic slicing unsuitable for answering queries about dependence queries at run-time.
3. Other slicing techniques have tried to combine the benefits of static and dynamic slicing e.g., hybrid slicing [22]. Research on these techniques is relatively still in its infancy. Moreover, there has been no comprehensive study that sheds light on the accuracy, efficiency and scalability aspects of those algorithms.

So, the state-of-art in slicing techniques has the following to offer: On one extreme is the highly scalable but imprecise static slicing techniques and on the other extreme is the precise but overhead-heavy dynamic slicing algorithms. There are no readily useable alternatives in the middle of the spectrum.

I propose a profile driven approach that does not involve the run-time overhead of dynamic-slicing, and could be potentially more precise than static-slicing. My approach avoids the runtime overhead of dynamic-slicing by doing most of the work *apriori*. The precision of my approach is tuneable. The approach is very suitable for studying the tradeoffs between overhead and precision in dynamically detecting dependence relationships between system events. In the next section 5.1.3, I discuss my approach in detail. I also discuss potential ways of improving the performance of my approach along various dimensions: accuracy, scalability and efficiency.

5.1.3 Profile Driven Dependence Detection

The basic approach consists of three phases: First, the binary of a given program is instrumented to the collection of dependence relationships between the system calls executed by the program. Second, the instrumented binary is executed over a set of inputs and the *dependence traces* are collected. Third, a *dependence model* is built based on the traces collected so far. The dependence model can be later queried at runtime by a “monitor” that observes the execution of the program, about dependence relationships between the executed system calls.

1. Instrumentation phase

For every memory location (main memory, registers etc.,) potentially accessed by the instructions executed by the binary, an alternate memory called *shadow memory*² is maintained.

²This is very similar to the shadow memory used by Taintbochs [12]. But, the information stored in the shadow memory is different.

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 binary 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 will be propagated to the shadow memory locations corresponding to all the locations that are modified by this system call. Additionally, code that dumps the labels stored in the shadow memory locations corresponding to the read operands of the system call is also added. For instructions that are not system calls, the binary is instrumented to propagate the label set information from the read operands of the instruction to the write operands. e.g., If an instruction adds two registers and writes the result into a third register, the instrumented code would perform a union of the label sets corresponding to the read registers and propagate them to the shadow memory of the write register.

2. Trace Collection phase

The instrumented binary is executed over a set of inputs. For each execution, the *dependence trace* consists of the sequence of label sets dumped by the instrumented binary for each executed system call.

3. Model Building phase

A *dependence model* is built from the collection of dependence traces. The dependence model is essentially a weighted directed graph whose nodes are the system calls present in the program binary. There is an edge from system call A to system call B, if in at least one of the traces, the label set associated with B contained A. The weight is some measure of the strength of the dependence relationship.

5.1.4 Issues and Challenges

The basic approach which was described in the previous section, can be improved along a series of dimensions such as scalability, accuracy and efficiency. The following issues are of particular interest:

- **The precision of the model:**

In the basic approach, the “weight” metric used is just a cumulative measure based on the frequency of the dependence relationship in the traces. It is insensitive to the execution context, the data flow and the control-flow of the program. The accuracy of the model could be increased by:

1. Adding Context-sensitivity:

- (a) The system call sequences found in the *dependence traces* provide us with information about the control flow of the program and consequently some context for each system call. [20, 55] use the system call sequence for providing context-sensitivity in their abstract stack model.
- (b) During the trace collection phase, the binaries can be instrumented to spit out additional control-flow information such as the function call sequence, which will provide additional context-sensitivity to the model.

2. Adding Data-flow and Path sensitivity:

Some dependence relationships are sensitive to data flow e.g., particular values of pointer variables [38] and also to the intraprocedural control-flow.

But the additional precision comes at a price. The more precise the model, greater are the details to be observed by the runtime “monitor” (e.g., function call sequences) to leverage the added precision. This increases the overhead of the monitor [3]. In general, I would like to explore the tradeoffs between the additional precision provided by the increased data/control-flow sensitivity and the overhead incurred.

- **A good representative set of testcases:**

An important factor that could significantly affect the accuracy of the results is the *coverage* provided by the set of inputs used to generate profiles. This issue is not unique to my approach. Every dynamic analysis technique has to overcome this issue. One promising approach is to continuously refine the model based on inputs observed in practice after the program has been deployed in multiple locations.

- **Scalability:**

The current implementation of the basic approach uses the PIN binary instrumentation tool³ developed by Intel for instrumenting x86 binaries on Linux [36]. Currently, I am able to successfully instrument binaries of medium-sized C files (a few thousand LOC) with low memory requirements.

One of the potential roadblocks to scaling the current implementation to larger programs or programs that are more memory hungry is the size of the *shadow memory*. Currently, the shadow memory is naively implemented as an array of label sets. In the worst case, the size of the shadow memory is proportional to the product of the memory requirements of the program and the number of system calls executed. We would like to develop more efficient data structures to make clever use of some of the redundancy that might be present across the different label sets, while keeping the resultant increase in time overhead within manageable limits.

Another potential optimization is *selective label propagation*. By leveraging the knowledge of program semantics, the instrumented binary could selectively “turn-off” label propagation, when the program enters a previously visited state (where no new information about the dependence relationships can be collected) saving valuable execution time.

- **Effectiveness of program dependences in answering causal queries:**

One of the motivations for building a tool to practically capture dependence relationships is that it would allow us to answer queries regarding causal relationships. I will evaluate the effectiveness of dependence relationships, by building an intrusion analysis system based on the dependence relationships and studying its effectiveness. I will also explore the possibility of developing heuristics based on dependence relationships that adequately capture both the *necessity* and *sufficiency* aspects of causality.

³It provides functionality similar to EEL [32] which provide instrumentation for SPARC binaries on solaris machines.

5.2 Practical determination of PAS causal relationships

In the previous section, I described my proposed approach to practically determine dependence relationships. After completing the work on dependence relationships, I will develop various techniques to reason about causal semantics in program execution traces. Specifically, I will explore how notions of “sufficiency” and “necessity” can be measured for system events. Traditionally, studies aimed at determining causal relationships have been conducted in fields like epidemiology, econometrics and AI. I believe that there are many differences (some encouraging and some discouraging) between those fields and our context:

- Knowledge of the functional mechanism:

For example, a epidemiologist that studies the causal relation between smoking and cancer is constrained by an incomplete knowledge of the actual process that enables smoking to become a cause for cancer. He is forced to rely on just statistical data to estimate the causal relationships between the two. Fortunately, the functional mechanisms that enable the PAS causal relationships between system events, viz. the executable code of the program, is completely known in my case.

- Controlled experimentation:

More often than not, the traditional fields were constrained by the inability to perform controlled experimentation and had to merely rely on observational statistics for determining causal relationships. For example, an agricultural engineer studying the effect of soil fumigants on oat crop yields could not perform randomized controlled experimentation since the farmers insisted on deciding which plots are to be fumigated. On the other hand, the executable code of a program is eminently suitable for controlled experimentation.

- The number of variables:

Traditional studies on causal relationships, have had to deal with only a few random variables at a time. Pearl’s IC algorithm [43] to infer causal relationships from statistical data has been found to scale well to just *tens* of random variables. But a software application could have *hundreds* or even *thousands* of system call sites. I believe that the knowledge of the executable code coupled with the ability to run randomized controlled experiments would help in successfully meeting the challenge posed by the large number of variables.

6 Expected Timeline

In working towards my dissertation, I hope to follow the following timeline:

- Implementation of basic approach to determine program dependences - June 2005
- Adding context, data-flow and path sensitivity - Aug 2005 (NDSS 2006)
- Scaling to larger, memory hungry applications - Nov 2005 (Oakland 2006)
- Implement an Intrusion Analysis system and evaluate - Jan 2006 (Usenix 2006)
- Determination of causal relationships - May 2006 (CCS 2006)
- Write Dissertation and defend - Aug 2006

References

- [1] H. Agrawal and J. Horgan. “Dynamic Program Slicing.” *In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 246-256, 1990.
- [2] J. P. Anderson, “Computer security threat monitoring and surveillance.” Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, April 1980.
- [3] T. Ball and J. R. Larus. “Efficient Path Profiling.” *International Symposium on Microarchitecture*, pp. 46-57, 1996.
- [4] Y. Beres and C. I. Dalton, “Dynamic label binding at run-time.” *In Proceedings of the 2003 workshop on New security paradigms*, pp. 39 - 46, 2003.
- [5] M. Bernaschi, E. Gabrielli, and L. V. Mancini. “Operating system enhancements to prevent the misuse of system calls.” *In Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 174-183, Athens, Greece.
- [6] D. Binkley, M. Harman, “A Large-Scale Empirical Study of Forward and Backward Static Slice Size and Context Sensitivity.” *In International Conference on Software Maintenance*, 2003.
- [7] 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., pp. 525-542. World Scientific, 2000.
- [8] F. Buchholz and E. H. Spafford. “On the role of file system metadata in digital forensics.” Technical Report, CERIAS TR 2004-56, 2004.
- [9] F. Buchholz and C. Shields. “Providing Process Origin Information to Aid in Computer Forensic Investigations.” Technical Report, CERIAS TR 2004-48, 2004.
- [10] B. D. Carrier and E. H. Spafford. “Defining Event Reconstruction of a Digital Crime Scene.” *Journal of Forensic Sciences*, 49(6), 2004.
- [11] B. D. Carrier and E. H. Spafford. “An Event-based Digital Forensic Investigation framework.” *In Proceedings of the 2004 Digital Forensic Research Workshop*, 2004.
- [12] 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.
- [13] F. Cuppens and A. Miige. “Alert correlation in a cooperative intrusion detection framework.” *In IEEE Symposium on Security and Privacy*, May 2002.
- [14] D. E. Denning and P. J. Denning. “Certification of programs for secure information flow.” *Comm. of the ACM*, vol. 20, no. 7, pp. 504-513, July 1977.
- [15] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.
- [16] 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 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

- [17] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. “Anomaly detection using call stack information.” *In IEEE Symposium on Security and Privacy*, May 2003.
- [18] J. Ferrante, K. Ottenstein, and J. Warren. “The program dependence graph and its use in optimization.” *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3(July) 1987, pp. 319-349.
- [19] 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.
- [20] J. Giffin, S. Jha, and B. Miller. “Efficient context-sensitive intrusion detection.” *In 11th Annual Network and Distributed Systems Security Symposium*, 2004.
- [21] J. A. Goguen and J. Meseguer. “Security policies and security models.” *In Proc. IEEE Symp. on Security and Privacy*, Apr. 1982, pp. 11 20.
- [22] R. Gupta and M. L. Souffa, “Hybrid slicing: an approach for refining static slices using dynamic information.” *In Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, 1995.
- [23] A. Hdtld, C. Sdrs, R. Addams-Moring and T. Virtanen, “Event Data Exchange and Intrusion Alert Correlation in Heterogeneous Networks.” *In Proceedings of the 8th Colloquium for Information Systems Security Education*, West Point, NY, June 2004
- [24] S. Horwitz, T. Reps, D. Binkley, “Interprocedural slicing using dependence graphs.” *ACM Transactions on Programming Languages and Systems* 12, 1, pp.26-61, 1990.
- [25] D. Hume, *An Enquiry Concerning Human Understanding*. 1748.
- [26] S. T. King and P. M. Chen. “Backtracking Intrusions.” *In Proceedings of the 2003 Symposium on Operating Systems (SOSP)*, Oct. 2003.
- [27] S. T. King, Z. M. Mao, D. G. Lucchetti, P. M. Chen, “Enriching Intrusion Alerts Through Multi-Host Causality.” *In Proceedings of the 2005 Network and Distributed System Security Symposium (NDSS)*, February 2005.
- [28] C. Ko. *Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach*. PhD thesis, U.C. Davis, September 1996.
- [29] C. Ko and T. Redmond. “Non-interference and intrusion detection.” *In IEEE Symposium on Security and Privacy*, May 2002.
- [30] B. Korel and J. Laski, “Dynamic Program Slicing.” *Information Processing Letters (IPL)*, Vol. 29, No. 3, pages 155-163, 1988.
- [31] 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.
- [32] J. R. Larus and E. Schnorr, “EEL: Machine-Independent Executable Editing.” *In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1995.

- [33] D. Lewis. *Counterfactuals*. Oxford: Blackwell. 1973.
- [34] D. Lewis. *Philosophical Papers: Volume II*. Oxford: Oxford University Press. 1986.
- [35] G. Lowe. “Quantifying information flow.” *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, Canada, 2002.
- [36] 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 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.
- [37] J. Millen. “Covert channel capacity.” *In Proceedings of the IEEE Symposium on Security and Privacy*, 1987.
- [38] M. Mock, D. Atkinson, C. Chambers, S. Eggers, “Improving program slicing with dynamic points-to data.” *In Proceedings of the 10th ACM International Symposium on the Foundations of Software Engineering*, 2002.
- [39] A. C. Myers. “JFlow: Practical mostly-static information flow control.” *In Proc. ACM Symp. on Principles of Programming Languages*, Jan 1999, pp. 228 241.
- [40] P. Ning, Y. Cui and D. S. Reeves. “Analysing Intensive Intrusion Alerts via Correlation.” *In Proceedings of Recent Advances in Intrusion Detection 2002*, LNCS 2516, p. 74-94; Springer-Verlag; 2002.
- [41] P. Ning, Y. Cui and D. S. Reeves. “Constructing attack scenarios through correlation of intrusion alerts.” *In Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.
- [42] Judea Pearl, “Reasoning with Cause and Effect,” *in Proceedings of the International Joint Conference on Artificial Intelligence*, San Francisco, Morgan Kaufman, pp. 1437 - 1449. 1999.
- [43] Judea Pearl, *Causality: Models, Reasoning, and Inference*, Cambridge: Cambridge University Press.
- [44] A. D. Pierro, C. Hankin, and H. Wiklicky. “Approximate non-interference.” *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, Canada, 2002.
- [45] X. Qin and W. Lee. “Statistical Causality of INFOSEC Alert Data,” *In Proceedings of Recent Advances in Intrusion Detection 2003*, LNCS 2820, p. 73- 94; Springer-Verlag; 2003.
- [46] H. Reichenbach, *The Direction of Time*. University of California Press, Berkeley and Los Angeles, 1956.
- [47] A. Sabelfeld and A. Myers. “Language-Based Information-Flow Security.” *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.
- [48] P. Sprites, C. Glymour, and R. Scheines, *Causation, Prediction and Search*, Second edition. Cambridge, MA: M.I.T. Press. 2000.

- [49] 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*, pp. 85 - 96, 2004.
- [50] P. Suppes. *A Probabilistic Theory of Causality*. Amsterdam: North-Holland Publishing Company. 1970.
- [51] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages* 3, 3, pp. 121-189, September 1995.
- [52] 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.
- [53] A. Valdes and K. Skinner. "Probabilistic Alert Correlation." *In Proceedings of Recent Advances in Intrusion Detection 2001*, LNCS 2212, p. 54-68, Springer-Verlag; 2001.
- [54] G. Venkatesh, "Experimental results from dynamic slicing of C programs." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 17, No. 2, pages 197-216, 1995.
- [55] D. Wagner and D. Dean. "Intrusion detection via static analysis." *IEEE Symposium on Security & Privacy*, 2001.
- [56] M. Weiser, *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.
- [57] X. Zhang, R. Gupta and Y. Zhang. "Precise dynamic slicing algorithms." *In Proceedings of the 25th International Conference on Software Engineering*, 2003.
- [58] 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.
- [59] 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.
- [60] 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.
- [61] SANS Critical Vulnerability Analysis Vol. 2. No. 9. March 10, 2003. Available at: http://www.sans.org/newsletters/cva/vol2_9.php
- [62] "2004 E-Crime watch Survey." Available at: http://www.csoonline.com/releases/052004129_release.html