CERIAS Tech Report 2006-20

AN EMPIRICAL STUDY OF AUTOMATIC EVENT RECONSTRUCTION SYSTEMS

by Sundararaman Jeyaraman, Mike Atallah

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

An empirical study of Automatic Event Reconstruction Systems

Sundararaman Jeyaraman, Mike Atallah
jsr,mja@cs.purdue.edu

Abstract

Reconstructing the sequence of computer events that led to a particular event is an essential part of the digital investigation process. The ability to quantify the accuracy of automatic event reconstruction systems is an essential step in standardizing the digital investigation process thereby making it resilient to tactics such as the Trojan Horse defense. In this paper, we present findings from an empirical study to measure and compare the accuracy and effectiveness of a suite of such event reconstruction techniques. We quantify (as applicable) the rates of false positives, false negatives, and scalability both in terms of computational burden and memory-usage. Some of our findings are quite surprising in the sense of not matching a priori expectations, and whereas other findings qualitatively match the a priori expectations they were never before quantitatively put to the test to determine the boundaries of their applicability. For example, our results show that automatic event reconstruction systems proposed in literature have very high false-positive rates (up to %).

1 Introduction

After every security incident, the universally asked questions are "what happened, and how did it happen?". 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., mis-using the system to send spam, send confidential material to outsiders, etc);
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.

Collectively, the process of identifying the underlying conditions and reconstructing the sequence of events that led to a security incident, is referred to as *event reconstruction* [1, 2]. Typically, event reconstruction involves forensic investigators manually sifting through evidence and proposing various hypotheses about the possible sequences of events that could have led to the security incident. The degree of difficulty and the accuracy of the results of the reconstruction process differ from case to case. For instance, for the second kind of incident, investigators are often hindered by the lack of sufficient evidence to reliably reconstruct the event sequence. Most often the image of the hard disk is the only evidence available to the investigators. In the presence of such limited information, event reconstruction often becomes a tedious and highly inaccurate effort [3].

However, the reconstruction process in cases of incidents typified by examples one and three (commonly referred to as operational forensic analysis or intrusion analysis [3] as opposed to prosecutorial forensic analysis in the case of example two) can often leverage the presence of additional evidence in the form of audit logs. In the past few years, many researchers have developed automated reconstruction systems that rely on a-priori audit logging to help the reconstruction effort. Examples include BackTracker [3], Forensix [6], Improved BackTracker [5] and the Process-Labels scheme proposed in [7]. The key idea is that, with more information logged about the events during the normal operation of a system, reconstruction becomes easier and can be automated [1].

Despite the growing body of literature regarding such automated reconstruction systems, there is hardly any work that quantifies their effectiveness. A rigorous study that quantifies their effectiveness is essential for the following reasons:

- The importance of reliability and accuracy metrics for forensic analysis techniques has been well documented [8].
  - 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 [9]. A forensic expert providing testimony in a court of law could buttress his/her conclusions by citing studies that evaluate the effectiveness of the reconstruction system.
  - Event reconstruction systems often provide multiple hypotheses regarding the events that could have led to 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.
- System administrators and forensic investigators need guidance as to which reconstruction system to deploy for their particular framework and circumstances (as we shall see, reconstruction systems that are suitable for certain situations mis-behave in others). A study that sheds light on the relative advantages and disadvantages of these systems will be useful in such investigations.
- Researchers can use such a study as a guide towards identifying the challenges that need to be tackled in order to build better reconstruction systems.

In this paper, we present an experimental study that evaluates the effectiveness of existing automated event reconstruction systems. Our contributions are the following:

- We develop a systematic approach for evaluating the effectiveness of the reconstruction techniques
  - We propose using the ability of reconstruction systems to infer causal relationships between system events as an indicator of their overall effectiveness. Identification of causal relationships between objects and events that occurred in a host and between the events themselves, is a critical step in every reconstruction effort [1, 2]. The higher the accuracy in identifying causal relationships, the more precise the resulting reconstruction.
  - Furthermore, we argue that the effectiveness of the event reconstruction systems we evaluate in our study is predicated on their ability to infer causal relationships enabled by *program dependences* [10]. Consequently, we propose using dynamic slicing [11] as a metric to benchmark the accuracy with which different systems identify causal relationships enabled by program dependences.
  - We develop a suite of real world applications and testcases for benchmarking the ability of reconstruction systems to identify causal relationships. The suite allows us to identify the source of inaccuracy and performance overhead of reconstruction techniques.

[1]Storing those logs in a tamper-resistant manner is a problem that lies in the critical path of the success of these systems. Cryptographic solutions [23] and Virtual Machine Introspection [24] are among the proposed approaches to make audit logs tamper resistant.

- Using our approach, we provide experimental data quantifying the effectiveness of the reconstruction systems 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 %. The legal ramifications of this result are substantial and it highlights the urgent need for more accurate event reconstruction systems.
  - The huge space and time overhead of dynamic slicing has been well documented [11, 12]. Our initial impression was that dynamic slicing was useful only for serving as a benchmark against which other techniques can be measured. However, we found out that our implementation of dynamic slicing runs with relatively moderate cost for I/O intensive applications which typically execute less number of instructions when compared to CPU intensive applications. Given the fact that many applications that might be involved in a security incident (HTTP, FTP, mail, file servers etc.,) are I/O intensive, the ability to use dynamic slicing to accurately identify causal relationships will be a great aid in forensic investigations.
- We discuss the experimental data and shed light on the conditions that lead to the inaccuracies and the overhead of the event reconstruction systems we evaluate.
  - For forensic practitioners and system administrators, this could be a valuable guide in choosing the appropriate reconstruction technique to deploy. For example, we found that the static slicing technique used by Improved BackTracker does not work well in applications that exhibit "recursive" and "iterative" work-flow characteristics (more on this in Section 6).
  - For academic and industrial researchers, this will aid in the development of more effective event reconstruction systems.

The rest of the paper is organised as follows. In Section 2, we provide a brief explanation of the background concepts needed in the rest of the paper. In Section 3 we survey the existing automatic event reconstruction systems. We explain our evaluation strategy in Section 4 and provide the results from our experiments in Section 5. We then briefly analyse the results in Section 6. Finally, we discuss future work in Section **??** and conclude in Section 7.

2 Background

In this section, we discuss the background concepts that are necessary for the discussion of the main results of this paper. 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.

System Events For the purpose of this paper, we consider an event to be an action that is performed by a process on behalf of a user. We define events at the granularity of system calls since most of the "interesting" actions (both from the point of view of an investigator and the perpetrator) that happen in a host consist of system calls. Henceforth, the terms computer event, event and system call are used interchangeably.

Event Causality Causality is commonly expressed using the following counter-factual query: would E (effect) have occurred if it were not for C (cause)? Intuitively, causality tries to capture the *influence* a *cause* has over an *effect*. In other words, how much dependent is E on C. An example counter-factual query using events: Would the *root kit be still installed* (effect), if it were not for the *email received by the email-server*(cause)? [2]

[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 a system call that received the corresponding network packets.

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 $< e_0, e_1, \ldots, e_k >$ 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, 2] (henceforth referred to as simply *reconstruction*).

Program Slicing Intuitively, a program slice [10, 11] 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 [11, 12].

3 Event Reconstruction Systems

There are many tools that can be used by forensic investigators and system administrators for event reconstruction purposes. In this section, we provide an overview of the available tools.

3.1 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 [21], SleuthKit [22], Guidance Softwareś Encase, Access Data's Forensic toolkit, ASR Data's SMART, Internal Revenue Services' ILook and Zeitline [15] help the investigators in collecting and analyzing the evidence from hard disk images. Tools such as Ethreal can interpret the network traffic at the application level. Since these tools are dependent on evidence available *ex post facto*, they are severely limited in their ability to reconstruct events and reason about what happened *ex ante facto.*

3.2 Ex ante logging

Unlike certain types of digital crimes, where the investigation is constrained by the absence of 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 and Forensix use this approach of combining pre-facto logging with post-facto intrusion analysis for event reconstruction.

BackTracker BackTracker [3] is an automatic event reconstruction tool that identifies chains of events that could have influenced a security incident. At run-time, BackTracker records system events that *induce* dependence relationships between operating system objects. A dependency relationship induced by an event consists of a *source object* (the cause), the *sink object* (the effect) and the *time* interval during which the event took place [4]. Once a security incident is detected, BackTracker constructs a dependency relationship graph using the dependency relationships inferred from the recorded events. The nodes of the graphs are operating system objects such as files, processes and file-names. The edges represent dependency 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 dependency graph backwards from the detection points using the dependency edges.

Evaluated	BackTracker, Forensix, Process Labels, Improved BackTracker (program slicing)
Not Evaluated	Improved BackTracker (file offsets), Memory Mapped Files

Table 1: Event reconstruction systems considered in this study

Forensix Forensix [6] is a forensics and intrusion analysis tool similar to BackTracker. It uses the SNARE framework [16] (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. The dependency relationships captured during the analysis phase are similar to those captured by BackTracker. Unlike BackTracker, 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.

Process-Labels Though not originally intended for event reconstruction purposes, the `Process-Labels` scheme proposed by Buchholz et al. [7] possesses the same capabilities as BackTracker. Buchholz et al. 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.

Improved BackTracker Sitaraman et al. [5] propose the following improvements to Backtracker:

Offset Intervals BackTracker (and Forensix), treats files as atomic objects – If a process modifies a file, it influences all future reads of the file regardless of which portion of a file is modified. This might lead to false-dependences. To overcome this, Improved-BackTracker recodes 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.

Program slicing Another major source of spurious (i.e., false) dependences in BackTracker (and Forensix) is the treatment of processes as Black boxes [4, 5]. Similar to files, processes are considered atomic (Black boxes) by BackTracker. To overcome this limitation, Improved-BackTracker uses program slicing techniques to track the program dependences.

Tracking memory mapped files Another source of false-dependences in BackTracker is its simplistic treatment of memory mapped files. Sarmoria et al. [4] propose a runtime kernel memory management monitor to observe the reads and writes to memory mapped files, facilitating finer-grained observation.

Table 3.2 lists the tools considered in this paper among those described in this section.

4 Evaluation Strategy

In this section, we explain our approach for measuring the effectiveness of event reconstruction systems.

4.1 Metrics for Event Reconstruction

The first challenge in measuring the effectiveness of reconstruction techniques is to decide upon a set of metrics. The key idea is that the effectiveness of the reconstruction process is directly dependent on the accuracy with which causal relationships between events are inferred. The Higher the accuracy of causality inference, the more effective the resulting reconstruction effort. We propose to use the rate of false-positives and false-negatives as a metric 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 a reconstruction system has a high rate of false-positives, the forensic investigator has to waste time investigating and eliminating the spurious hypotheses. In the worst case, the existence of spurious hyphotheses 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 the security incident. Hence, we use the rate of false-positives and the rate of false-negatives to evaluate the effectiveness of the reconstruction systems under consideration.

4.2 Measurement Methodology

The next challenge in evaluating the effectiveness of event reconstruction systems is to develop a suite of benchmarks to measure the metrics defined in section 4.1. Initially, we considered using a suite of "scenarios" – a collection of security incidents along with corresponding audit logs, disk and memory images. The reconstruction systems would then be used to reconstruct event chains for each scenario and the resulting false-positives and false-negatives could be measured. In fact, previous work on the systems described in section 3 adopted a combination of qualitative reasoning and scenarios to evaluate their systems. 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 [17]. Event reconstruction systems are similar to IDSes in the sense that they are both complex and their effectiveness is predominantly dependent on the operating environment.

Fortunately, the following observation allows us to develop a benchmark suite for reconstruction systems that is less biased and is more scientific than a suite of scenarios.

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

Causal relationships between events are enabled by *causal mechanisms* [18]. 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 are of the following two types:

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, where process-1 and process-2 execute a sequence of system calls in the specified order. The `write` system call of process-2 is a *cause* of the `read` system call of process-1, because the result of the `read` system call is *dependent* on the `write` system call. In other words, the data that is "used" by `read` is 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 [3].

```
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

Program Dependences Causal relationship between two events could be enabled by the address space of a process if both events are executed by the same process. For example, in the piece of code listed in Figure 2, the causal relationship between the `read` and the `write` is enabled by a chain of program dependences between the two events. The `write` event 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` event. We refer to such causal relationships are *Program Dependence enabled* (or simply *PD*) *relationships.*

```
fd_r = open(foo, O_RDONLY);
fd_w = open(bar, O_WRONLY);
read(fd_r, buffer, 10);
if( buffer[0] == 0 )
strncpy(dest, buffer, 10);
write(fd_w, dest, 5);
```

Figure 2: Program Dependences example

Because the semantics of system calls are well defined (the effect of each system call on system objects is well understood), all reconstruction systems under consideration are able to make precise deductions about OS-enabled relationships. As a result, there is no difference in their ability to infer OS-enabled relationships.

On the other hand, the reconstruction systems vary in their ability to infer PD relationships. For example, BackTracker, Forensix and the Process-labels scheme treat the processes as blackboxes. On the other hand, Improved-BackTracker and Memory-Mapped Files both have the ability to observe the process address space at a finer granularity [4]. Intuitively, this ability should make them more accurate than those systems that treat processes as mere blackboxes.

Hence to make an assessment of the effectiveness of reconstruction systems, it is sufficient to measure the effectiveness of the techniques employed the systems for inferring PD relationships:

BackTracker BackTracker, Forensix and Process-Labels treat PD relationships similarly. They simply consider processes as blackboxes. Their detection policy is simple: Any input event is a cause for future output events. Henceforth, we refer to this technique as simply the BackTracker technique.

Static Slicing 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 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. Hence, we use it as a baseline for measuring the effectiveness of the other techniques. False-positives arise when a particular technique infers a PD 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.

5 Experimental Evaluation

5.1 The benchmarks

Our benchmark suite consists of a collection of open source applications and a suite of testcases for each application. The table 5.1 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 `2939` LOC. `GnuPG` is the largest application with `68081` 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`. For each application in our suite, we develop a set of testcases (test suite), designed to maximise the coverage of the functionality of the respective application. Some of the applications, have publicly available regression test

Name	Description	Lines of Code (LOC)
gnuPG 1.4.2	GNUs replacement for PGP	68081
gnu wget 1.10	Program for retrieving files through HTTP(S), FTP	22268
find (findutils 4.2.25)	Search for files in a directory hierarchy	19217
locate (findutils 4.2.25)	List files in a database that matches pattern	11864
ls (coreutils 4.5.3)	List directory contents	2939
cp (coreutils 4.5.3)	Copy files	3321
wc (coreutils(4.5.3)	Print the number of bytes, words and lines in a file	3226
tar 1.15.1	Archiving software	8425
gzip 1.3.3	A popular data compression program	4296
grep 2.5.1	Search files for a given input pattern	7485

Table 2: Table listing the applications in the benchmark suite

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

Table 3: Table listing the system calls considered in this study

suites, e.g., `GnuPG`. In such cases, we borrow those test suites. If no such suite is publicly available for an application (e.g., `gzip, wget`), we develop our own testsuite. In this study, we consider causal relationships between the system calls listed in table table 5.1. 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, along with the number of instructions executed are presented as part of Table 5.3.

5.2 Implementation

Identifying causal relationships using Backtracker technique is straight forward. We implement this functionality as a simple table lookup. For Dynamic Slicing, we implemented a customised variation that suites our purposes using PIN [19]. PIN is a binary instrumentation tool developed by Intel. We use CodeSurfer [20] a program analysis 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 it could be 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. Owing to space restrictions, additional details of the dynamic slicing implementation using PIN and details regarding our usage of CodeSurfer can be obtained from [**?**].

5.3 Results

For each application in the benchmark suite, we ran the testcases in the application's 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 used BackTracker, static slicing and dynamic slicing to determine if S_a is a cause of S_b as explained in sections 4.2 and 5.2. We calculate the rate of false-positives and false-negatives as explained in Section 4.1. A total of 110882 system calls and 11,328,876,826 (approx. 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 5.3. The false-positive and the false-negative rate for the dynamic slicing technique is 0 by definition. Static slicing technique has a 0 false-negative rate (by definition).

The time and memory overhead associated with dynamic slicing is reported in 5. Both Static Slicing and BackTracker 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 [13]. Owing to space restrictions we provide the actual data regarding the overhead of BackTracker and static slicing in [?]

We note some significant results :

1. The rate of false-positive is extremely high for both techniques. For BackTracker, it varies from % in the case of `locate` to % in the case of `tar`. For Static Slicing, it varies from a low of % in the case of `cp` to a high of % in the case of `tar`.

2. Contrary to previously published expectations [5], in most applications (except `cp` and `ls`, Static Analysis does not provide a significantly better performance than BackTracker.

3. Dynamic slicing has a lower overhead for I/O intensive applications such as `wget` (4933x) when compared to CPU intensive applications such as `gzip` (32894x). The overhead in table 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 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 drops to 4.74x.

   Though the absolute overhead of dynamic slicing is large (53x - 43298x), it scales very well with an increase of system calls. For example, applications such as `wget`, have a relatively lower overhead (4933x) especially when amortized over a large number of system calls. The reason is that, `wget` is an I/O intensive application, and hence executes less number of instructions (the cost of dynamic slicing is dominated by the number of instructions executed). This result is significant given the fact that many applications involved in security incidents are likely to be I/O intensive (web servers, file servers etc.,). Investigators can directly use dynamic slicing instead of resorting to less accurate alternatives.

			BackTracker				Static Slicing	
			False-positives		False-negatives		False-positives	
Application	System Calls	Instructions	Avg	Std	Avg	Std	Avg	Std
gpg	45762	9735745189	95.60	12.88	1.88	9.28		
wget	49239	168432151	31.78	28.99	3.55	3.00	64.94	12.44
find	2602	11640606	59.34	27.27	8.96	8.90	52.99	25.13
locate	78	15674625	81.44	33.40	0	0	0	0
tar	7659	109540215	93.01	20.16	0.20	2.02	92.59	22.92
gzip	1775	824574115	35.32	30.43	0	0	30.21	38.41
wc	266	441862104	36.61	43.35	0	0	36.75	43.51
ls	2936	17563651	85.01	20.79	0	0	83.23	25.92
cp	464	3511599	67.44	31.21	0.04	0.72	54.74	41.08
grep	101	332571	48.30	41.58	0	0	14.13	28.76

Table 4: The rate of false-positives and false-negatives for BackTracker and Static Slicing. We were unable to obtain the results for `gpg` in the case of static slicing owing to limitations of `codesurfer`

6 Discussion and Future Work

- It is important to analyse the root causes of the inaccuracy of both BackTracker and Static Slicing so that better event reconstruction techniques can be developed. A complete analysis is out of the scope of this work. However, we did some preliminary analysis on Static Slicing. We found that, the main reasons for the ineffectiveness of static slicing are “iterative” and “recursive” program structures. To illustrate the case of the “iterative” program structure, consider the application ’ls’. A high-level over-view of ’ls’ looks like this:

Application	Time					Memory (MB)			
	Avg	std	Min	Max	Overhead	Avg	Std	Min	Max
gpg	787.489	585.39	49.96	4953.9	8458	450.25	78.34	275.23	788.87
wget	162.808	65.55	31.32	427.72	4933	431.56	84.73	274.50	638.01
find	49.97	5.82	40.45	74.26	648.96	313.99	22.78	275.03	378.9
locate	43.29	.26	42.67	43.72	43298	104.58	142.35	4.11	339.94
tar	38.40	30.95	15.06	263.1	12802	308.95	27.86	276.29	385.97
gzip	180.91	478.55	28.02	2530.32	32894	431.98	96.91	3.61	502.62
wc	178.06	303.92	36.68	1132.69	28719	345.46	21.86	274.21	357.22
ls	38.32	18.73	17.47	78.60	22153	310.56	47.57	3.71	393.20
cp	15.54	4.32	10.44	32.35	10502	23.62	.87	22.36	25.29
grep	28.15	9.85	16.26	53.76	53.31	159.30	156.10	4.14	384.85

Table 5: Time (in seconds) and Memory (in MB) overhead associated with Dynamic Slicing. Time overhead is a ratio of the dynamic slicing time to the normal application execution time.

```
while (pending_dirs)
{
    extract_files_from_dir(pending_dirs);
    print_files();
}
```

When the 'ls' command is executed, it iterates over a list of directories. For each directory, it extracts the files residing in the directory and prints the files. The extracting a single file involves a `readdir` system call and printing information about a file involves a `write` call. Now, consider the 'ls' command being invoked with arguments: 'ls dir1 dir2' (assume 'dir1' and 'dir2' each have only one file). In this case, Static Analysis declares the `readdir` call associated with 'dir1' to be a cause of the both the `write` call associated with 'dir1' and the one associated with 'dir2'.

- The suite 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. We were severely constrained by the fact that our applications should be compatible to both PIN and Codesurfer. We found that Codesurfer was the bottleneck due to its limitations on handling applications of large size ($>$ 100K LOC). In future work, we would like to work on expanding the benchmark to a more comprehensive one.
- Program Dependences are not the only means through with 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 $i$mplicit dependences which are not captured using program dependences alone [14]. Exploring the impact of implicit dependences and the relation between information-flow and causal relationship is another promising area that we would like to work on.

7 Conclusion

In this study we measured the effectiveness of a suite of event reconstruction systems. We found that reconstruction systems that treat processes as blackboxes (`BackTracker, Forensix, Process-Lables`) and systems that use Static Slicing, have a very high rate of false-positives. We also found that contrary to previously published work, Static Slicing does not provide a significantly better alternative to BackTracker. Additionally, we also document the time and memory overhead of dynamic slicing. We found that dynamic slicing could be a practical alternative while investigating I/O intensive applications.

References

[1] B. D. Carrier and E. H. Spafford. “Defining Event Reconstruction of a Digital Crime Scene.” *Journal of Forensic Sciences,* 49(6), 2004.

[2] B. D. Carrier and E. H. Spafford. “An Event-based Digital Forensic Investigation framework.” *In Proceedings of the 2004 Digital Forensic Research Workshop,* 2004.

[3] S. T. King and P. M. Chen. “Backtracking Intrusions.” *In Proceedings of the 2003 Symposium on Operating Systems (SOSP),* Oct. 2003.

[4] Christian G. Sarmoria, Steve J. Chapin. “Monitoring Access to Shared Memory-Mapped Files” *DFRWS* 2005.

[5] S. Sitaraman and S. Venkatesan. “Forensic Analysis of File System Intrusions Using Improved Backtracking.” *In Third IEEE International Workshop on Information Assurance (IWIA05),* College Park, Maryland, USA, 2005. IEEE Association.

[6] A. Goel, M. Shea, S. Ahuja, W.-C. Feng, W.-C. Feng, D. Maier, and J. Walpole. “Forensix: A Robust, High- Performance Reconstruction System.” *Distributed Computing Systems Workshops, ,* 2005.

[7] F. Buchholz and C. Shields. “Providing Process Origin Information to Aid in Computer Forensic Investigations.” Technical Report, CERIAS TR 2004-48, 2004.

[8] Gary. Palmer, “A Road Map for Digital Forensic Research.”, *DFRWS* 2001.

[9] Brenner, Susan, Brian Carrier, and Jef Henninger. “The Trojan Defense in Cybercrime Cases,” *Santa Clara Computer and High Technology Law Journal,* 21(1), (2004).

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

[11] X. Zhang, R. Gupta and Y. Zhang. “Precise dynamic slicing algorithms.” *In Proceedings of the 25th International Conference on Software Engineering,* 2003.

[12] X. Zhang and R. Gupta. “Cost Effective Dynamic Program Slicing,” *ACM SIGPLAN Conference on Programming Language Design and Implementation,* pages 94-106, Washington D.C., June 2004.

[13] David Binkley and Mark Harman. “A Large-Scale Empirical Study of Forward and Backward Static Slice Size and Context Sensitivity ” *19th IEEE International Conference on Software Maintenance (ICSM’03),* p. 44, 2003.

[14] 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.

[15] Florian Buchholz and Courtney Falk, “Design and Implementation of Zeitline: a Forensic Timeline Editor”, *DFRWS*, 2005.

[16] L. Purdie and G. Cora. SNARE - System iNtrusion Analysis & Reporting Environment. http://www.intersectalliance.com/projects/Snare/. Viewed in Jan 2003.

[17] Marcus. J. Ranum, “Coverage in Intrusion Detection Systems ”, http://www.snort.org/docs/Benchmarking-IDS-NFR.pdf

[18] 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.

[19] 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.

[20] CodeSurfer. Available at www.grammatech.com/products/codesurfer/index.html

[21] The Coronerś toolkit. Available at: http://www.porcupine.org/forensics/tct.html

[22] The Sleuth Kit. Available at: http://www.sleuthkit.org/

[23] B Schneier, J Kelsey. “ Cryptographic Support for Secure Logs on Untrusted Machines.” *In Proceedings of the 7th USENIX Security Symposium,* 1998.

[24] T Garfinkel, M Rosenblum. “A Virtual Machine Introspection Based Architecture for Intrusion Detection.” *In Proceedings of the 2003 Network and Distributed System Security Symposium.*