An Empirical Study of Automatic Event Reconstruction Systems

By

Sundararaman Jeyaraman, Mikhail Atallah

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2006 USA

Lafayette, IN (Aug 14th - 16th)

available at www.sciencedirect.com

SCIENCE DIRECT®

journal homepage: www.elsevier.com/locate/diin

Digital Investigation

An empirical study of automatic event reconstruction systems☆

*Sundararaman Jeyaraman**, *Mikhail J. Atallah*

Department of Computer Sciences, Purdue University, 250 North University Street, West Lafayette, IN 47907-2066, USA

Keywords:
Intrusion analysis
Digital forensics
Event reconstruction
Incident response

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 and false negatives, and scalability in terms of both 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 96%).

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., 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.

☆ Portions of this work were supported by Grants IIS-0325345, IIS-0219560, IIS-0312357, and IIS-0242421 from the National Science Foundation, and by sponsors of the Center for Education and Research in Information Assurance and Security.

* *Corresponding author.*
E-mail addresses: jsr@cs.purdue.edu (S. Jeyaraman), mja@cs.purdue.edu (M.J. Atallah).

1742-2876/$ – see front matter © 2006 DFRWS. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2006.06.013

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* (Carrier and Spafford, 2004a,b). 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 (King and Chen, 2003).

However, the reconstruction process in cases of incidents typified by examples one and three (commonly referred to as operational forensic analysis or intrusion analysis (King and Chen, 2003) 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 (King and Chen, 2003), Forensix (Goel et al., 2005), Improved BackTracker (Sitaraman and Venkatesan, 2005) and the Process Labels scheme proposed in Buchholz and Shields (2004). 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 (Palmer, 2001).
  - 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 (Brenner et al., 2004). 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 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.
- 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 misbehave 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, based on their ability to infer causal relationships between system events that are enabled by *program dependences* (Weiser, 1979). The higher the accuracy in identifying causal relationships, the more precise the resulting reconstruction.
- 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.
- 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 96%. The legal ramifications of this result are substantial and this highlights the urgent need for more accurate event reconstruction systems.
- We analyze 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 example, we found that BackTracker and the static slicing technique used by Improved BackTracker do not work well in applications that exhibit ''recursive'' and ''iterative'' workflow characteristics (more on this in Section 5.3).

The rest of the paper is organized 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 and analyze the results in Section 5. Finally, we discuss future work in Section 6 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.

[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 (Schneier and Kelsey, 1998) and Virtual Machine Introspection (Garfinkel and Rosenblum) are among the proposed approaches to make audit logs tamper resistant.

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 from the point of view of 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*. The following is 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]

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 $\langle e_0, e_1, \ldots, e_k \rangle$ 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* (Carrier and Spafford, 2004a,b) (henceforth referred to as simply *reconstruction*).

Program slicing. Intuitively, a program slice (Weiser, 1979; Zhang et al., 2003) 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 (Zhang et al., 2003; Zhang and Gupta, 2004).

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, Sleuth Kit, Guidance Software's Encase, Access Data's Forensic toolkit, ASR Data's SMART, Internal Revenue Services' ILook and Zeitline (Buchholz and Falk, 2005) 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*

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 *ex ante* logging with *ex post* intrusion analysis for event reconstruction.

BackTracker. This is an automatic event reconstruction tool that identifies chains of events that could have influenced a security incident (King and Chen, 2003). At runtime, 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 (Sarmoria and Chapin, 2005). 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.

Forensix. This is a forensics and intrusion analysis tool similar to BackTracker (Goel et al., 2005). It uses the SNARE framework (Purdie and Cora, 2003) (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 is 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 and Shields (2004) 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

[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.

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 and Venkatesan (2005) 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 the 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 (Sitaraman and Venkatesan, 2005; Sarmoria and Chapin, 2005). 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 and Chapin, 2005 propose a runtime kernel memory management monitor to observe the reads and writes to memory mapped files, facilitating finer-grained observation.

Table 1 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 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 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 hypotheses 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.

Table 1 – Event reconstruction systems considered in this study

Evaluated	BackTracker, Forensix, Process Labels, Improved BackTracker (program slicing)
Not evaluated	Improved BackTracker (file offsets), memory mapped files

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 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 (Ranum). 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* (Pearl, 1999). 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 Fig. 1, where process-1 and process-2 execute a sequence of system calls in the specified order. The `write` system

```
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);
```

Fig. 1 – OS relationship example.

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 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 (King and Chen, 2003).

Program dependences. Causal relationships 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 Fig. 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 as *program dependence enabled* (or simply PD) *relationships*.

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 black boxes. On the other hand, Improved BackTracker and memory mapped files both have the ability to observe the process address space at a finer granularity (Sarmoria and Chapin, 2005). Intuitively, this ability should make them more accurate than those systems that treat processes as mere black boxes.

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

BackTracker. BackTracker, Forensix and Process Labels treat PD relationships similarly. They simply consider processes as black boxes. Their detection 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.

```
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);
```

Fig. 2 – Program dependences example.

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. Table 2 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`. For each application in our suite, we develop a set of testcases (test suite), designed to

Table 2 – List of the applications in the benchmark suite

Name	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
locate (findutils 4.2.25)	List files in a database that matches pattern	11,864
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

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 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. 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 4.

5.2. *Implementation*

Identifying causal relationships using the BackTracker technique is straightforward. We implement this functionality as a simple table lookup. For dynamic slicing, we implement a customized variation that suites our purposes using PIN (Luk et al., 2005), a binary instrumentation tool developed by Intel. We use CodeSurfer, 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. Owing to space restrictions, additional details of the dynamic slicing implementation using PIN and details regarding our usage of CodeSurfer are omitted and can be obtained from our technical report (Jeyaraman and Atallah, 2006).

5.3. *Results*

For each application in the benchmark suite, we run 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 use 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 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 4. 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 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 (Binkley and Harman, 2003). Owing to space restrictions we leave the actual data regarding the overhead of BackTracker and static slicing to the technical report (Jeyaraman and Atallah, 2006).

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 gpg – 95.6%. For static slicing, it is 92.59% in the case of tar.
2. Contrary to plausible expectations (Sitaraman and Venkatesan, 2005), in most applications (except grep and locate), static analysis does not provide a significantly better precision than BackTracker. In some cases such as wget it is actually much worse than BackTracker. 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) (Binkley and Harman, 2003). 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% (gpg). This suggests that the nature of an application plays a crucial rule in determining its amenability to causality inference. A comprehensive analysis of this effect is beyond the scope of the current study. However, we did some preliminary analysis and found that the "iterative" and "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

Table 3 – List of the system calls considered in this study

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

Table 4 – The rate of false positives for BackTracker and static slicing

Application	System calls	Instructions	Back Tracker		Static slicing	
			False positives		False positives	
			Avg	Std	Avg	Std
gpg	45,762	9,735,745,189	95.60	12.88		
wget	49,239	168,432,151	31.78	28.99	64.94	12.44
find	2602	11,640,606	59.34	27.27	52.99	25.13
locate	78	15,674,625	81.44	33.40	0	0
tar	7659	109,540,215	93.01	20.16	92.59	22.92
gzip	1775	824,574,115	35.32	30.43	30.21	38.41
wc	266	441,862,104	36.61	43.35	36.75	43.51
ls	2936	17,563,651	85.01	20.79	83.23	25.92
cp	464	3,511,599	67.44	31.21	54.74	41.08
grep	101	332,571	48.30	41.58	14.13	28.76

We were unable to obtain the results for gpg in the case of static slicing owing to limitations of codesurfer. "Avg" and "Std" stand for average and standard deviation, respectively.

Table 5 – Time (s) and memory (MB) overhead associated with dynamic slicing

Application	Time					Memory			
	Avg	Std	Minimum	Maximum	Overhead	Avg	Std	Minimum	Maximum
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	43,298	104.58	142.35	4.11	339.94
tar	38.40	30.95	15.06	263.1	12,802	308.95	27.86	276.29	385.97
gzip	180.91	478.55	28.02	2530.32	32,894	431.98	96.91	3.61	502.62
wc	178.06	303.92	36.68	1132.69	28,719	345.46	21.86	274.21	357.22
ls	38.32	18.73	17.47	78.60	22,153	310.56	47.57	3.71	393.20
cp	15.54	4.32	10.44	32.35	10,502	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

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.

prints the files. This is an example of iterative workflow. The extraction of information 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 Fig. 3. In this case, both BackTracker and static analysis 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).

Similarly, consider the case when `ls` is executed with the arguments: "-R dir1 dir2 dir3". In this case, `ls` recursively lists all the files under "dir1", "dir2" and "dir3". We found that BackTracker and static slicing spuriously label the `readdir` calls associated with subdirectory "d1" of "dir1" as causes of system calls associated with "dir2" and "dir3".

4. Dynamic slicing 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 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 (Jeyaraman and Atallah, 2006).

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

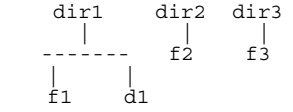

Fig. 3 – "dir1", "dir2" and "dir3" are directories. "dir1" consists of file "f1" and directory "d1". "dir2" consists of file "f2" and directory "dir3" contains file "f3".

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 (Jeyaraman and Atallah, 2006). 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.

6. Limitations and future work

- 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. In future work, we would like to expand the benchmark to a more comprehensive one.
- We were also severely 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 (>100K LOC). For instance, the version of CodeSurfer we used could not handle the slicing queries for `gpg`. Also, as indicated in Section 5.3, 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 reconstruction technique.
- The testcases developed for each application were designed to exercise maximum coverage of each application's 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 (Vachharajani et al., 2004). 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.
- A more comprehensive analysis is needed for uncovering the root causes for the imprecision of static slicing and BackTracker in order to develop reconstruction techniques that are more precise. We are currently working on developing more precise reconstruction techniques.

7. Conclusion

In this study, we propose an approach to evaluate the effectiveness of automatic event reconstruction techniques, based on their ability to detect causal relationships enabled through program dependences. We use our approach to evaluate a suite of reconstruction systems and conclude that BackTracker, Forensix, Process Labels and systems that use static slicing, 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 dynamic slicing. We found that dynamic slicing could be a practical alternative while investigating I/O-intensive applications.

REFERENCES

Binkley David, Harman Mark. A large-scale empirical study of forward and backward static slice size and context sensitivity. In: 19th IEEE international conference on software maintenance (ICSM'03); 2003. p. 44.

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

Buchholz Florian, Falk Courtney. Design and implementation of Zeitline: a forensic timeline editor. Digital Forensics Research Workshop; 2005.

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

Carrier BD, Spafford EH. Defining event reconstruction of a digital crime scene. Journal of Forensic Sciences 2004a;49(6).

Carrier BD, Spafford EH. An event-based digital forensic investigation framework. In: Proceedings of the 2004 digital forensic research workshop; 2004b.

CodeSurfer, <www.grammatech.com/products/codesurfer/index.html>.

Garfinkel T, Rosenblum M. A virtual machine introspection based architecture for intrusion detection. In: Proceedings of the 2003 network and distributed system security symposium; 2003.

Goel A, Shea M, Ahuja S, Feng W-C, Maier D, et al. Forensix: a robust, high-performance reconstruction system. In: Distributed computing systems workshops; 2005.

Jeyaraman S, Atallah M. An empirical study of event reconstruction systems. CERIAS technical report. CERIAS TR 2006-20.

King ST, Chen PM. Backtracking intrusions. In: Proceedings of the 2003 symposium on operating systems (SOSP); October 2003.

Luk C, Cohn R, Muth R, Patil H, Klauser A, Lowney G, et al. Pin: building customized program analysis tools with dynamic instrumentation. In: ACM SIGPLAN conference on programming language design and implementation (PLDI); 2005.

Palmer Gary. A road map for digital forensic research. Digital Forensics Research Workshop; 2001.

Pearl Judea. Reasoning with cause and effect. In: Proceedings of the international joint conference on artificial intelligence. San Francisco: Morgan Kaufman; 1999. p. 1437–49.

Purdie L, Cora G. SNARE – system iNtrusion analysis & reporting environment, <http://www.intersectalliance.com/projects/Snare/> [accessed January 2003].

Ranum Marcus J. Coverage in intrusion detection systems, <http://www.snort.org/docs/Benchmarking-IDS-NFR.pdf>; 2001.

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

Schneier B, Kelsey J. Cryptographic support for secure logs on untrusted machines. In: Proceedings of the 7th USENIX security symposium; 1998.

Sitaraman S, Venkatesan S. Forensic analysis of file system intrusions using improved Backtracking. In: Third IEEE international workshop on information assurance (IWIA'05). College Park, Maryland, USA: IEEE Association; 2005.

The Sleuth Kit, <http://www.sleuthkit.org/>.

The Coroner's toolkit, <http://www.porcupine.org/forensics/tct.html>.

Vachharajani N, Bridges MJ, Chang J, Rangan R, Ottoni G, Blome JA, et al. RIFLE: an architectural framework for user-centric information-flow security. In: Proceedings of the 37th international symposium on microarchitecture (MICRO); December 2004.

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

Zhang X, Gupta R. Cost effective dynamic program slicing. In: ACM SIGPLAN conference on programming language design and implementation. Washington, D.C.; June 2004. p. 94–106.

Zhang X, Gupta R, Zhang Y. Precise dynamic slicing algorithms. In: Proceedings of the 25th international conference on software engineering; 2003.

Sundararaman Jeyaraman is a graduate student in the Department of Computer Sciences at Purdue University where he is affiliated with the Center for Education and Research in Information Assurance and Security (CERIAS). His research interests span the areas of Intrusion Analysis, Digital Forensics, Dynamic Program Analysis and Usable Security.

Mikhail ("Mike") Atallah is a graduate student in the Department of Computer Sciences at Purdue University where he is obtained the PhD in 1982 from Johns Hopkins and joined Purdue, where he currently holds the rank of Distinguished Professor. He served on the editorial boards of top journals (SICOMP, JPDC, IEETC, etc), and on the Program Committees of top conferences (PODS, SODA, CG, WWW, etc). He was Keynote/Invited Speaker at top conferences, and speaker in the Distinguished Colloquium Series of six top CS Departments. He was selected in 1999 as one of the best teachers in the history of Purdue and included in a permanent wall display of Purdue's best teachers past and present.