CERIAS Tech Report 2006-06

A HYPOTHESIS-BASED APPROACH TO DIGITAL FORENSIC INVESTIGATIONS

by Brian D. Carrier

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

A HYPOTHESIS-BASED APPROACH TO DIGITAL FORENSIC INVESTIGATIONS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Brian D. Carrier

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2006

Purdue University

West Lafayette, Indiana

To my parents,

Gerry and Suzanne.

ACKNOWLEDGMENTS

I would first like to thank my advisor, Eugene Spafford (spaf). His advice and guidance helped to shape and direct this work. I would also like to thank my committee for their helpful ideas and comments: Cristina Nita-Rotaru, Sunil Prabhakar, Marc Rogers, and Sujeet Shenoi (University of Tulsa). The Center for Education and Research in Information Assurance and Security (CERIAS) provided me with a great environment and the resources needed to complete this work and my appreciation goes to the faculty and staff.

Thanks to the many people in the digital forensics community that have assisted me over the years. Special thanks to Dan Kalil, Chet Maciag, Gary Palmer, and others at the Air Force Research Labs for the creation of the annual Digital Forensic Research Workshop (DFRWS). This work is based on concepts from the initial DFRWS Research Roadmap and the framework discussions at DFRWS 2004 helped to direct this work. Simson Garfinkel provided many helpful comments during his review of this document.

Lastly, thanks to my family. My parents have always supported my endeavors (and gently guided me towards a degree in engineering instead of one in music). Enormous appreciation goes to my wife Jenny for being my best friend, moving back to West Lafayette from Boston, and being extra understanding during the final months of this process.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

SYMBOLS

The symbols listed here are in a Times font, but they are used in the document in two different fonts. If the name is in sans serif then the set or function is part of the ideal history model. If the name is in *italics* then the set or function is part of the inferred history model.

$\Sigma_{\mathrm{prim(t)}}$	Set of possible primitive event symbols at time t.
$\Sigma_{\mathrm{X(t)}}$	Set of possible complex event symbols at time t and layer $\mathrm{X} \in \mathrm{L}$.
$\delta_{\mathrm{prim}}(\mathrm{t})$	State change function for primitive events at time t.
$\delta_{\mathrm{X}}(\mathrm{t})$	State change function for complex events at time t and layer $\mathrm{X} \in \mathrm{L}$.
$\mathrm{ABS_{cs}}$	Set of complex storage abstraction transformation functions.
$\mathrm{ABS_{ce}}$	Set of complex event abstraction transformation functions.
$\mathrm{ADO_{cs}}$	Set of possible values for each complex storage address.
$\mathrm{ado_{cs}}$	Function to map a complex storage address to its set of possible values.
$\mathrm{ADO_{ps}}$	Set of possible values for each primitive storage address.
$\mathrm{ado_{ps}}$	Function to map a primitive storage address to its set of possible values.
$\mathrm{AST_{cs}}$	Set of possible states for each complex storage address.
$\mathrm{ast_{cs}}$	Function to map a complex storage address to its possible states.
$\mathrm{AST_{ps}}$	Set of possible states for each primitive storage address.
$\mathrm{ast_{ps}}$	Function to map a primitive storage address to its possible states.
$\mathrm{C_{ce}}$	Set of all combinations of program names (power set of $\mathrm{D_{ce}}$).
$\mathrm{c_{ce}}$	Function to map a time to the program configuration.

C_{cs}	Set of all combinations of complex storage types (power set of D_{cs}).
c_{cs-X}	Function to map a time to the complex storage type configuration at layer $X \in L$.
C_{pe}	Set of all combinations of primitive event device names (power set of D_{pe}).
c_{pe}	Function to map a time to the primitive event device configuration.
C_{ps}	Set of all combinations of primitive storage device names (power set of D_{ps}).
c_{ps}	Function to map a time to the primitive storage device configuration.
CCG_{ce-X}	Set of state change functions for each program configuration at layer $X \in L$.
ccg_{ce-X}	Function to map a program configuration to its state change function at layer $X \in L$.
CCG_{pe}	Set of state change functions for each primitive event device configuration.
ccg_{pe}	Function to map a primitive event device configuration to its state change function.
CST_{cs}	Set of possible states for each complex storage configuration.
cst_{cs-X}	Function to map a complex storage configuration to its possible states at layer $X \in L$.
CST_{ps}	Set of possible states for each primitive storage device configuration.
cst_{ps}	Function to map a primitive storage device configuration to its possible states.
CSY_{ce-X}	Set of possible event symbols for each program configuration at layer $X \in L$.

csy_{ce-X}	Function to map a program configuration to its possible event symbols at layer $X \in L$.
CSY_{pe}	Set of possible event symbols for each primitive event device configuration.
csy_{pe}	Function to map a primitive event device configuration to its possible event symbols.
D_{ce}	Set of program names (complex event).
D_{cs}	Set of complex storage types.
D_{pe}	Set of primitive event device names.
D_{ps}	Set of primitive storage device names.
DAD_{cs}	Set of addresses for each complex storage type.
dad_{cs}	Function to map a complex storage type to its set of addresses.
DAD_{ps}	Set of addresses for each primitive storage device.
dad_{ps}	Function to map a primitive storage device to its set of addresses.
DAT_{cs}	Set of attributes for each complex storage type.
dat_{cs}	Function to map a complex storage type to its set of attributes.
DCG_{ce-X}	Set of possible state change functions for each program at layer $X \in L$.
dcg_{ce-X}	Function to map a program to its state change function at layer $X \in L$.
DCG_{pe}	Set of possible state change functions for each primitive event device.
dcg_{pe}	Function to map a primitive event device to its state change function.
DEF_{ce}	Set of primitive and complex storage locations that are defined by complex events.
def_{ce}	Function to map a complex event symbol to the locations that it writes to.

DEF_{pe}	Set of primitive storage locations that are defined by primitive events.
def_{pe}	Function to map a primitive event symbol to the locations that it writes to.
DST_{cs}	Set of possible states for each complex storage type.
dst_{cs}	Function to map a complex storage type to its possible states.
DST_{ps}	Set of possible states for each primitive storage device.
dst_{ps}	Function to map a primitive storage device to its possible states.
DSY_{ce-X}	Set of possible event symbols for each program at layer $X \in L$.
dsy_{ce-X}	Function to map a program to its possible event symbols at layer $X \in L$.
DSY_{pe}	Set of possible event symbols for each primitive event device.
dsy_{pe}	Function to map a primitive event device to its possible event symbols.
H	Set of inferred history identifiers.
h_{ce}	Function to map a time to the complex event that occurred at that time.
h_{cs}	Function to map a time to the complex state that existed at that time.
h_{pe}	Function to map a time to the primitive event that occurred at that time.
h_{ps}	Function to map a time to the primitive state that existed at that time.
L	Set of layers of event abstraction that exist on a system.
MAT_{cs}	Set of complex storage materialization transformation functions.
MAT_{ce}	Set of complex event materialization transformation functions.
$Q_{prim}(t)$	Set of possible primitive states at time t.
$Q_X(t)$	Set of possible complex states at time t and layer $X \in L$.
SAD_{cs}	Set of system wide complex storage addresses.

SAD_{ps}	Set of system wide primitive storage addresses.
SL_{ce}	Set of ordered lists of event symbols from each abstraction category and primitive-level events.
SSY_{ce-X}	Set of system wide complex event symbols.
T	Set of times that the system has a history.
USE_{ce}	Set of primitive and complex storage locations that are used by complex events.
use_{ce}	Function to map a complex event symbol to the locations that it reads from.
USE_{pe}	Set of primitive storage locations that are used by primitive events.
use_{pe}	Function to map a primitive event symbol to the locations that it reads from.

ABSTRACT

Carrier, Brian D. Ph.D., Purdue University, May, 2006. A Hypothesis-Based Approach to Digital Forensic Investigations. Major Professor: Eugene H. Spafford.

This work formally defines a digital forensic investigation and categories of analysis techniques. The definitions are based on an extended finite state machine (FSM) model that was designed to include support for removable devices and complex states and events. The model is used to define the concept of a computer's history, which contains the primitive and complex states and events that existed and occurred. The goal of a digital investigation is to make valid inferences about a computer's history.

Unlike the physical world, where an investigator can directly observe objects, the digital world involves many indirect observations. The investigator cannot directly observe the state of a hard disk sector or bytes in memory. He can only directly observe the state of output devices. Therefore, all statements about digital states and events are hypotheses that must be tested to some degree.

Using the dynamic FSM model, seven categories and 31 unique classes of digital investigation analysis techniques are defined. The techniques in each category can be used to test and formulate different types of hypotheses and completeness is shown. The classes are defined based on the model design and current practice.

Using the categories of analysis techniques and the history model, the process models that investigators use are formally compared. Until now, it was not clear how the phases in the models were different. The model is also used to identify where assumptions are made during an investigation and to show differences between the concepts of digital forensics and the more traditional forensic disciplines.

1 INTRODUCTION

This chapter provides the motivation and general background material for this work. Section 1.1 describes the current state of digital investigations and why additional theory is needed. Section 1.2 describes the background work related to computer models, definitions, and investigation procedures. Section 1.3 describes the approach taken by this work to define a digital investigation, which is based on a series of hypotheses about the system's capabilities and states. Section 1.4 provides the thesis statement and an outline of the remainder of this document.

1.1 Motivation

Digital investigations, or digital forensics, are conducted by law enforcement and corporate investigation teams on a regular basis. Yet, no formal theory exists for the process. A practitioner in the field can describe how he recognizes evidence for a specific type of incident, but the recognition process cannot typically be described in a general way.

Most forensic science disciplines have theories that are published, generally accepted, and testable, but digital forensics does not [Pal01]. To date, the digital investigation process has been directed by the technology being investigated and the available tools. While the applied focus helps to solve today's crimes, it is too limiting when the longer-term needs of the field are considered.

For example, when an investigation is conducted to answer questions related to an incident that will be brought to a court, some guidelines may need to be met. Some states in the U.S. use the Daubert guidelines [Uni93] when technical or scientific evidence is entered [SB03]. These guidelines exist to prevent "junk science" from being used in the courtroom. The four guidelines are as follows:

- Has the procedure been published (preferably in a journal)?
- Is the published procedure accepted by the relevant professional community?
- Can the procedure be tested?
- What is the error rate?

These are basic principles of science, yet it is debatable if they are met by current digital investigations [Car02] [MR05]. Most of the procedures used by currently available tools are not published or publicly tested. The National Institute of Standards and Technology (NIST) has a Computer Forensic Tools Testing group [Uni05b] that tests tools, but the focus thus far as been on tools that copy hard disks and prevent evidence from being overwritten. The searching and analysis features of tools have not been tested, although they may be in the future. This work does not try to solve all of the problems associated with Daubert guidelines and digital evidence, but it provides a general theory that could be applied in the future.

While some may claim that theory is not needed in an applied field that is constantly changing, recent challenges in physical investigations show that scientific rigor is useful when considering investigation techniques. The U.S. Federal Bureau of Investigations (FBI) recently discontinued the use of lead analysis in bullets [Fed05]. Between the the 1960s and 2005, the FBI lab would analyze the chemical elements in two bullets to determine if they may have been manufactured at roughly the same time and therefore could be from the same box or carton. The use of this analysis technique was stopped because scientists and bullet manufacturers were not able to determine the significance of the analysis results [Com04].

Recently, fingerprint analysis has been under increased scrutiny. In 2004, a man in Boston was freed after serving 6.5 years in prison after the assistant district attorney admitted that the original latent fingerprint match was a mistake [SD04]. Also in 2004, Brandon Mayfield was arrested in response to the train bombings in Madrid, Spain based on digital photographs of latent fingerprints, which were later determined to not be his [Fed04]. The Massachusetts Supreme court in 2005

considered the value of latent fingerprints in a case where a match was made based on accumulating points from prints of four different fingers, instead of only one finger [Lav05]. Fingerprint identification has occurred since the 19th century, but there is not an agreed upon standard [IR01] and this makes legal attacks easier to launch.

The goal of this work is to define the basic theory of a digital investigation and therefore direct future research and identify where assumptions are being made in practice. The theory is based on the design and theory of computers, but most computer models are used to evaluate computability and complexity, which have different requirements than are needed for an investigation of previous computations.

This work has two major contributions. First, it defines a model that can describe the previous events and states of a computer at the primitive and abstract levels. Second, it uses the model to define 31 unique classes of analysis techniques, which are organized into seven categories. Completeness for the categories can be shown and these classes can be used to more easily define requirements.

The following goals were used to define the model:

- The model must be based on the theoretical foundations of computing so that existing and future work in computer science can be used.
- The model must be general with respect to the technology being investigated so that the theory will apply to future as well as current technologies.
- The model must be capable of supporting events and storage locations at arbitrary levels of abstraction so that complex systems can be represented.
- The model must be capable of supporting systems with removable storage and event devices.
- The model must be capable of describing previous events and states so that all evidence can be represented.

Using this model, categories of analysis techniques are defined. The following goals were used for the categories:

- The categories must be general with respect to the investigation technology so that new techniques can be identified and supported.
- The categories must be general with respect to the types of investigations and apply to law enforcement, industry, and military so that common terminology can be used.
- The categories must be specific so that general requirements can be defined to direct testing and development efforts.

1.2 Related Work

While there is little existing work on the formal theory of digital investigations, this work is based on the theory of computation and the practice of digital investigations. This section provides an overview of related work on computation models, investigation-related definitions, and digital investigation process models. Additional related work will be described in the following chapters when the relevant topic is discussed.

1.2.1 Computation Models

Many models of computation have been proposed for the purposes of answering questions about which problems can be computed or which problems can be efficiently computed [Mor98]. In their basic form, they do not satisfy the requirements previously outlined, but a brief overview of the common models is given in this section.

The classic example of a model of computation is the Turing Machine [Tur36], which has a tape of infinite length that is divided into squares. Each square can contain one symbol from a finite alphabet. The machine has a read and write head that scans a square of the tape and moves it to the left or right. The machine also has a finite state control system, which is the logic of the system and it uses the symbol

under the head to determine if the tape should be moved to the left or right. This model mimics a person solving a problem using a notebook of infinite size [Mor98].

The Turing Machine does not resemble modern computers because it uses an infinite tape, but the Random Access Machine (RAM) models, also called Register Machines, have a central processor and an infinite number of registers of infinite size. These more closely resemble the computers that exist in practice. A RAM and Turing machine can both simulate each other [Mor98].

A general computational model is a Finite State Machine (FSM). A FSM is defined by a quintuple $M = (Q, \Sigma, \delta, s0, F)$ where Q is a finite set of machine states and Σ is a finite alphabet of event symbols. The transition function $\delta : Q \times \Sigma \rightarrow Q$ is the event mapping between states in Q for each event symbol in Σ. The machine state changes only as a result of a new input symbol. The starting state of the machine is $s0 \in Q$ and the final states are $F \subseteq Q$ [Sud97]. The concept of a FSM is used extensively in this work.

1.2.2 Digital Investigation Definitions

To define the theory of an investigation, an investigation must be first defined and understood. The definitions of a digital investigation, which is more commonly called computer or digital forensics, are considered first. This work uses the term digital investigation because the process being considered is related more to a crime scene investigation than the traditional forensic sciences [CS03].

A commonly referenced definition of computer forensics is that it "involves the preservation, identification, extraction, documentation, and interpretation of computer data [KH01]." The Scientific Working Group on Digital Evidence (SWGDE), which is a group that was formed by the directors of federal labs, defines computer forensics as involving "the scientific examination, analysis, and/or evaluation of digital evidence in legal matters [Sci05]." Both of these definitions are general in that

they are not technology specific, but they describe a taxonomy of techniques and it is not clear if the taxonomy is sufficient or necessary.

The definition of digital evidence is considered next. Some examples include:

- "information stored or transmitted in binary form that may be relied upon in court" [Int02]
- "information of probative value that is stored or transmitted in binary form" [Sci05]
- "information and data of investigative value that is stored on or transmitted by a computer" [Ass05]
- "any data stored or transmitted using a computer that support or refute a theory of how an offense occurred or that address critical elements of the offense such as intent or alibi" [Cas04]

Some of these definitions consider only data that can be entered into a legal system while others consider all data that may be useful during an investigation, even if they are not court admissible. In general, legal requirements, which are locale specific, limit the evidence that can be used [Twi85] and therefore the set of objects with legal value is a subset of the objects with investigative value. For this work, a general theory of evidence is used, which can be restricted to satisfy the rules of evidence in a specific legal system.

1.2.3 Digital Investigation Process Models

This section describes digital investigation process models, which are typically used for training and education purposes to help show what types of procedures and techniques are used. Because one of the goals of this work is to define classes of analysis techniques, the phases in the process models were considered to determine if their phases have unique technical requirements and therefore form a class of techniques. After reviewing the models, a unique set of classes could not be defined.

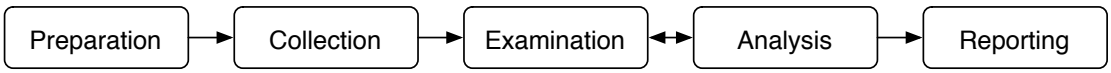

Figure 1.1. The NIJ Electronic Crime Scene Guide has a process with five phases in it for a digital investigation.

This is because phases in a process model are created based on the experiences of one or more investigators and may not be complete for all types of investigations. An overview of some of the process models is given in the following four subsections.

NIJ Electronic Crime Scene Model

The U.S. National Institute of Justice (NIJ) published a process model in the Electronic Crime Scene Investigation Guide [Uni01]. The guide is a first responder's reference to different types of electronic evidence and includes procedures to safely handle the evidence. The guide is oriented towards those who respond to the physical crime scene, so emphasis is placed on the collection process. The five phases are shown in Figure 1.1 and are listed here:

- Preparation: Prepare the equipment and tools to perform the tasks required during an investigation.
- Collection: Search for, document, and collect or make copies of the physical objects that contain electronic evidence.
- Examination: Make the electronic evidence "visible" and document contents of the system. Data reduction is performed to identify the evidence.
- Analysis: Analyze the evidence from the Examination phase to determine the "significance and probative value."
- Reporting: Create examination notes after each case.

The target of the paper was the Collection phase, so few details of the Examination and Analysis phases are given beyond lists of data types that may contain evidence for a certain type of crime. Without additional research, it was not clear if the requirements for the Examination and Analysis phases were different. It could be argued that the techniques used to reduce data in the Examination phase are simply a more general form of the techniques in the later Analysis phase, which performs data reduction by identifying the evidence that is significant. If this is the case, then the model has only a single phase that contains analysis techniques.

There are several other models that have phases similar to this model, such as the framework from the Digital Forensic Research Workshop Research Roadmap [Pal01] and the *Abstract Process Model* [RCG02]. Because they are similar, they are not covered here.

Integrated Digital Investigation Process Model

A digital investigation process model based on the investigation process of a physical crime scene was also proposed [CS03]. The model has high-level phases for the analysis of both the physical crime scene where the computer was located and the digital data. The *digital crime scene* was defined as "the virtual environment created by software and hardware where digital evidence of a crime or incident exists."

The high-level Digital Crime Scene Investigation phase is organized into the same five phases that can be found in a physical crime scene investigation, such as those described in [JN03] [LPM01] [Saf00]. The phases can be seen in Figure 1.2 and are described here:

- Preservation: Preserve the state of the digital crime scene.
- Survey: Search for obvious evidence that is relevant to the investigation.
- Documentation: Document the digital crime scene.
- Search: Conduct a more thorough search for all evidence not found in the Survey phase.

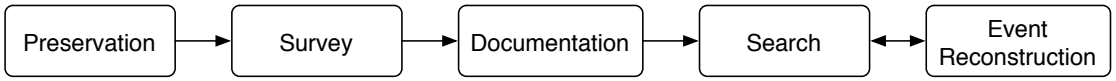

Figure 1.2. The Integrated Digital Investigation Process Model has five phases in it the Digital Crime Scene Investigation that are based on a physical crime scene process.

- Event Reconstruction: Reconstruct the digital events that occurred at the crime scene.

When these phases are considered for technical requirements, it becomes clear that the Survey phase is a basic form of the Search phase. More specifically, the Survey phase searches for evidence that is characteristic of the type of incident, for example looking for a rootkit in a computer intrusion case.

By recognizing that the Survey phase is not unique, the model was simplified to the Preservation, Search, and Reconstruction phases to define the *Event-Based Digital Forensic Investigation Framework* [CS04b]. The goals of each of these phases are unique and requirements can be defined. Because completeness is not shown for the three phases, it is not clear if this framework is sufficient.

Hierarchical, Objectives-Based Framework

The Hierarchical, Objectives-Based Framework has two tiers, similar to the crime scene-based model, but it has different phases at both levels [BC04]. The phases of the first tier are as follows:

- Preparation: Prepare equipment and people for investigation.
- Incident Response: Detect, validate, and assess the incident and determine a response strategy.
- Data Collection: Collect digital evidence in support of the response strategy.

- Data Analysis: Survey, extract, and reconstruct the collected data for evidence.
- Presentation: Communicate findings to the appropriate audience.
- Incident Closure: Review the investigation process and make changes.

The Data Analysis phase is further organized into the following second-tier phases:

- Survey: Describe the digital "landscape" by mapping the file system and special locations.
- Extract: Extract data based on objectives using techniques such as keyword searching and filtering.
- Examine: Examine the extracted data for reconstruction and confirmation or refutation of the goals.

The Extract and Examine phases in this model appear to be similar to the Analysis and Examine phases of the NIJ model and it again is not clear if the filtering process required to extract data is simply more general than the filtering that occurs during examination. The Survey phase in this model has a clear goal of analyzing the data abstractions and their characteristics.

End-to-End Digital Investigation Process

The *End-to-End Digital Investigation Process* has nine phases that focus mainly on the analysis process [Ste03] [Ste04b]. The model has phases to collect evidence, analyze the events, and then correlate and normalize the events. Much of the focus is on merging events from multiple locations.

This model focuses more on the analysis process than other models do, but is not consistent with where focus is given. For example, the process of searching for and finding evidence is a complex and time consuming process and must occur before events can be analyzed. However, it is grouped into a single phase with the collection and preservation of the data. As with the previous process models, the completeness of the phases in this model is not clear.

1.3 A Hypothesis-Based Approach

The process models described in the previous section do not satisfy our requirements. This section describes the approach taken by this work.

An investigation commences to answer questions about previous states or events. To answer the questions, the investigator makes observations and formulates hypotheses about what existed or occurred. Investigators cannot, in general, directly observe digital data and instead they can only observe the data displayed on a monitor or other output device, which is driven by various types of hardware and software. Because the observation of the data is indirect, a hypothesis must be formulated that the actual data is equal to the observed data. Testing this hypothesis requires that the hardware and software being used are accurate and reliable. Hypotheses also need to be formulated about the data abstractions that exist and the previous states and events that occurred. Chapter 4 shows that classes of analysis techniques can be defined based on the types of hypotheses that are formulated and tested during an investigation.

This work defines a *digital investigation* as a process that formulates and tests hypotheses to answer questions about digital events or the state of digital data. This is based on the concept of using the scientific method during an investigation [IR01] [LPM01] and a more general form of Casey's definition of digital evidence [Cas04]. Therefore, instead of having a definition that focuses on a specific set of phases in the process, this work focuses on the general problem of making inferences about a system's previous states and events. Section 5.1 will show that the difference between the process models previously described in Section 1.2.3 is based on what types of hypotheses are formulated in each phase.

To test a hypothesis, the investigator searches for data that supports or refutes it. This work defines *digital evidence* as digital data that supports or refutes a hypothesis about digital events or the state of digital data. An object is evidence because it contains information about events that occurred before, during, or after

the incident being investigated. Rynearson observes that "everything is evidence of some event. The key is to identify and then capture evidence relative to the incident in question [Ryn02]." Each piece of data can be used to support or refute some hypothesis and an investigation is interested in the data related to the questions being asked.

Note that this definition includes evidence that may not be capable of being entered into a court of law, but may have investigative value. This is not a shortcoming because many corporations and intelligence agencies conduct investigations and collect evidence that will not be entered into a court of law. When evidence is collected using procedures that will be court admissible, the investigation is called a *digital forensic investigation*. The addition of the word "forensic" when the evidence is legally admissible is based on the definition of forensic as an adjective and "relating to the use of science or technology in the investigation and establishment of facts or evidence in a court of law [Hou00]."

The investigation process relies on the formulation of hypotheses to explain what happened. At the lowest levels of an investigation, hypotheses will be used to reconstruct events and to abstract data into files and complex storage types. At higher levels of an investigation, hypotheses will be used to explain user actions and sequences of events.

It could be argued that the digital investigation process itself requires more types of hypotheses than a physical world investigation because there are fewer occurrences in the digital world that are relatively constant. The scientific community has, through observation and experimentation over many years, developed laws that, for example, allow an investigator or analyst to predict the forces of gravity and he does not need to measure it at each crime scene. However, at a digital crime scene the forces that control how events occur are defined by software and hardware and they can change with each new version or be changed by a user or attacker. Therefore, there are no general laws at digital crime scenes and hypotheses must be formulated about these variables each time. In many cases, the only constants that can be

taken as "axioms" are the machine instructions of hardware processors because it is relatively difficult, but not impossible, for an attacker to modify them and still have a functioning system.

Further, in the physical world an investigator can directly observe many states and events using his senses. In the digital world, hardware and software are needed to observe digital events and states and therefore a digital investigator relies more on indirect observations and must test and trust his observation tools.

1.3.1 Scientific Hypotheses

The term "digital forensic science" has been frequently used (for example [Pal01]), but it has not been clear which of the techniques use scientific principles and which are engineering. This work proposes that all techniques can be based on science if they are based on scientific hypotheses.

The scientific process is based on formulating hypotheses and testing them through experiments [Den05]. A hypothesis tries "to explain some otherwise unexplained happenings by inventing a plausible story [QU98]." If a system is behaving differently then two hypotheses could be that there is a system malfunction or that it has been compromised by an attacker.

Not all hypotheses may be considered scientific. Sir Karl Popper, a philosopher of science, has said that the "criterion of the scientific status of a theory is its falsifiability, or refutability, or testability [Pop98]." More specifically, he came to the following conclusions with respect to confirming and refuting hypotheses (emphasis in original text):

- "It is easy to obtain confirmations, or verifications, for nearly every theory – if we look for confirmations."
- "Confirmations should count only if they are the result of *risky predictions*; that is to say, if, unenlightened by the theory in question, we should have

expected an event which was incompatible with the theory – an event which should have refuted the theory."

- "Every 'good' scientific theory is a prohibition: it forbids certain things to happen. The more a theory forbids, the better it is."
- "A theory which is not refutable by any conceivable event is nonscientific. Irrefutability is not a virtue of theory (as people often think) but a vice."
- "Every genuine *test* of a theory is an attempt to falsify it, or to refute it. Testability is falsifiability; but there are degrees of testability; some theories are more testable, more exposed to refutation, than others; they take, as it were, greater risks."
- "Confirming evidence should not count *except when it is the result of a genuine test of the theory*; and this means that it can be presented as a serious but unsuccessful attempt to falsify the theory."
- "Some genuinely testable theories, when found to be false, are still upheld by their admirers – for example by introducing ad hoc some auxiliary assumption, or by reinterpreting the theory ad hoc in such a way that it escapes refutation. Such a procedure is always possible, but it rescues the theory from refutation only at the price of destroying, or at lest lowering, its scientific status."

Therefore, the digital investigation process can be considered scientific if it formulates and tests hypotheses that are scientific. The formulation of hypotheses is a creative process and it is as much of an art with digital investigations as it is with physics or chemistry. There are many approaches to hypothesis formulation and problem solving and this paper does not choose any one in particular.

1.4 Thesis Statement and Outline of Dissertation

The thesis statement for this dissertation is as follows:

> It is possible to define a digital investigation framework that is based on how digital data are created and designed, and that shows what steps must occur during the investigation.

Note that this work considers only the discrete state of the system. Typically, a storage location has both a continuous and discrete value. A discrete value is written to the location, but it is stored in a continuous form. For example, hard disks store discrete values as a magnetic field. This work considers only the discrete value of the location, although the continuous value can also be useful. It has been proposed that the previous values of a storage location can be determined from the continuous value of magnetic media [Gut96], although as the density of magnetic media has increased this technique is more difficult [Gut01].

This dissertation is organized as follows. Chapter 2 describes the basic model for representing the history of a system. The history of a system is the sequence of its previous states and events and an investigator wants to make as many valid inferences as possible about a system's history. The process of inferring about previous states and events involves formulating hypotheses and Chapter 3 describes the general process for doing this. Specific types of hypotheses about previous states and events are discussed in Chapter 4 to define categories and classes of analysis techniques. Chapter 5 brings the theory and current practice together using the concepts of the history and classes of analysis techniques. Lastly, Chapter 6 summarizes the findings. Appendix A provides results from reconstructing the states of a simple system.

2 IDEAL AND INFERRED COMPUTER HISTORY MODELS

This chapter describes two computer models that are used to define a theory of digital investigations. The goals for these models were previously defined in Section 1.1. Section 2.1 describes the theoretical concept of a history, which corresponds to the previous states and events of the computer. Sections 2.2, 2.3, and 2.4 define the first model, which is the *ideal computer history* model. Section 2.5 defines the second model, which is the *inferred computer history* model.

2.1 An Object's History

The basic concept that motivates the models defined in this work is that all objects, both physical and digital, have a history since the time they were created. The term *object* can refer to any entity, such as a physical thing or a bounded set of digital storage locations.

For physical objects, the history includes stimulation from the senses: what the object "touched," "heard," or "saw" etc. Objects that are not alive do not interpret what they see, but the concept of sight is used to represent how film records the state of physical objects based on light. When a physical crime is considered, the investigators and forensic scientists analyze objects at the scene to determine their history, for example to determine if someone or something touched it or if it has an odor from a cigarette or cologne. Materials may have transfered between objects when they came into contact and the evidence of the contact may exist when the investigation starts. A goal of an investigation is to learn as much as possible about the history of objects at a crime scene and objects that are suspected of being from the crime scene. The investigator will never know the full history of an object, but he can make inferences about it.

Histories occur at multiple levels. In physical objects, there are histories at the molecular and macroscopic levels. Not all investigations need to determine all levels of an object's history. For example, an object may need to be analyzed using only visual techniques or it may need to be analyzed at lower levels if DNA information is needed.

For a computer, the history includes the states and events that have occurred. A history includes the machine instructions that have been executed as well as complex events, such as a user clicking on a button in a graphical interface. As in the physical world, evidence from previous states and events may exist when a computer is investigated. The investigator must make inferences about the previous states and events based on a final, and possibly intermediate, states of the system.

2.2 General Model Concepts

This section defines basic concepts that apply to both models presented in this chapter. A *digital system* is defined as a connected set of digital storage and event devices. Digital storage devices are physical components that can store one or more values and a digital event device is a physical component that can change the state of a storage location. The *state* of a system is the discrete value of all storage locations and an *event* is an occurrence that changes the state of the system. This can be graphically seen in Figure 2.1, which shows an event $\mathtt{E}_1$ reading the values from storage locations R3 and R6 and writing to locations R3 and R4. The *history* of a digital system describes the sequence of states and events between two times.

The computer history models presented in this chapter make assumptions about the system being investigated. The main assumption is that the system can be represented by a finite state machine (FSM) quintuple $(\mathsf{Q}, \Sigma, \delta, \mathsf{s0}, \mathsf{F})$. Modern computers are FSMs with a large number of states and complex transition functions. It should be stressed that this work does not assume that a computer will be reduced to a

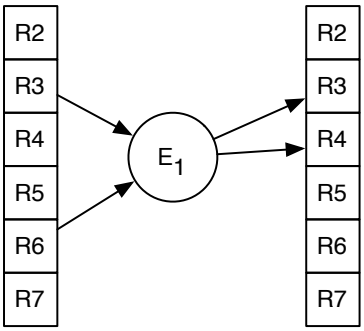

Figure 2.1. Graphical representation of an event $\mathtt{E}_1$ that causes a state change by writing to locations `R3` and `R4`.

FSM during an investigation. A FSM is used as a generic computer to prevent us from being tied to a specific platform.

When considering the history of a modern computer, the FSM model is too simple and static because it does not directly support removable components. The removal or insertion of components, such as external storage devices, coprocessors, and networks, will cause the storage and computing capabilities of a system to change and therefore the Q, Σ, and δ FSM sets and functions must also change. In addition, some systems have programmable devices where the logic for instructions can be changed and therefore the transition function δ can change. In most other contexts where FSMs are used, a new FSM can be defined when the system changes, but in this context the system changes must be included in the model because the possible states at a given time may need to be determined.

To account for the changing system, functions that map a time to the value of a FSM variable are defined:

- $\Sigma(\mathsf{t})$ is the symbol alphabet of the FSM at time $\mathsf{t} \in \mathsf{T}$.
- $\mathsf{Q}(\mathsf{t})$ is the set of all possible states of the FSM at time $\mathsf{t} \in \mathsf{T}$.
- $\delta(\mathsf{t})(\mathsf{s}, \mathsf{e})$ is the transition function of the FSM at time $\mathsf{t} \in \mathsf{T}$ for state $\mathsf{s} \in \mathsf{Q}(\mathsf{t})$ and event $\mathsf{e} \in \Sigma(\mathsf{t})$.

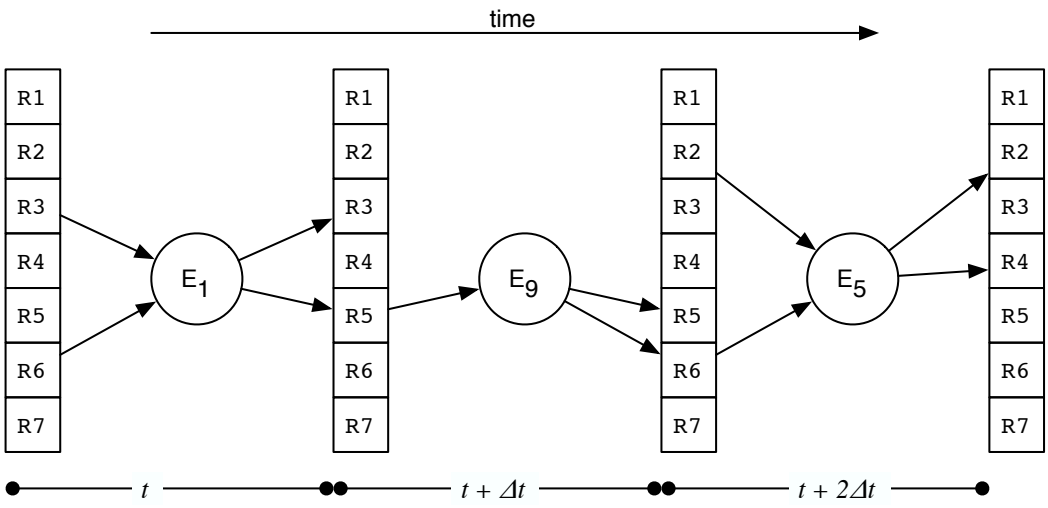

Figure 2.2. Graphical representation of a sequence of events where the history of the system includes the events and state at each time.

The following sections describe these functions in more detail.

When discussing the history of the system, a counter is used as a time reference, where the granularity of each time step (Δt) is less than or equal to the shortest possible time between state changes. This is important because if the time step is too large then multiple state changes could occur during a single time step and the model will not be able to represent all possible state changes. This requirement assumes that a synchronous FSM exists that changes state only at given times.

Consider Figure 2.2, which shows an event sequence for three time steps. The boxes with `Rx` are registers and the circles represent an event that reads one or more registers and writes to one or more registers. For each time t, there is a starting state and an event. The history for this system includes the states and events that existed at each time step.

The names of mathematical sets will have all uppercase characters and the names of functions will have all lowercase characters. The first letter of the name is based

on what the set or function is about. For example, the set "DAD" is about device ("D") addresses ("AD").

Sets and functions are defined for both the primitive and abstract systems and the possible states and events. The first letter of the subscript in a set or function name is a "p" if the set or function is for the primitive system and it is an "c" if the set or function is for the complex system. The second letter of the subscript is a "s" if the set or function is for the system's state and an "e" if the set or function is for the system's events.

In all, two models are defined in this work. The first model is the ideal computer history and it is defined to determine what is theoretically needed to fully represent the history of a system. This model is used to identify the sets and functions that must exist so that the primitive and complex history can be defined. The second model is the inferred computer history and it exists to represent what the investigator defines during the course of an investigation based on his inferences. Both models use the same set and function names, but different fonts are used. The ideal history uses a sans serif font for the names and the inferred history uses an *italic font*.

2.3 The Ideal Primitive History

The *primitive history* of a system includes the complete sequence of primitive states and events that occurred. *Primitive events* are the lowest-level state transitions that a computer program can explicitly cause to occur. For example, in most computers primitive events are caused by programs executing machine instructions.

The *primitive state* of a system is defined by the discrete values of its primitive storage locations, each of which has a unique address. The primitive storage locations are those that primitive events can read from or write to. For example, in many current computers the primitive state includes registers, memory, hard disks, networks, and removable media.

In this work, networks are treated as a short-term storage device. Network devices, such as routers and switches, are computers themselves and each has a FSM definition. The network cables and connections are treated as storage devices between computers and therefore a networked computer will have events to write to the network "storage location."

The primitive history of a system is typically not of interest to an investigator because he does not need to answer questions about machine-level instructions and register contents. It is included in the model though because it represents the actual sequence of events that occurred and future investigations may rely on these details if memory is analyzed in more detail than it currently is.

The primitive history model for a system is formally defined by a tuple with 11 variables:

$$(\mathsf{T}, \mathsf{D}_{\mathsf{ps}}, \mathsf{DAD}_{\mathsf{ps}}, \mathsf{ADO}_{\mathsf{ps}}, \mathsf{c}_{\mathsf{ps}}, \mathsf{h}_{\mathsf{ps}}, \mathsf{D}_{\mathsf{pe}}, \mathsf{DSY}_{\mathsf{pe}}, \mathsf{DCG}_{\mathsf{pe}}, \mathsf{c}_{\mathsf{pe}}, \mathsf{h}_{\mathsf{pe}}) \tag{2.1}$$

T is the set of consecutive time values for which the history is defined and, as previously described, the duration between consecutive time values ($\Delta\mathsf{t}$) is smaller than the fastest state transition. The D_{ps}, $\mathsf{DAD}_{\mathsf{ps}}$, $\mathsf{ADO}_{\mathsf{ps}}$, and c_{ps} sets and functions describe the storage device capabilities and when they were connected to the system. The h_{ps} function is the primitive state history function. The D_{pe}, $\mathsf{DSY}_{\mathsf{pe}}$, $\mathsf{DCG}_{\mathsf{pe}}$, and c_{pe} sets and functions describe the event devices and corresponding state changes and when they were connected to the system. The h_{pe} function is the primitive event history function.

The sets and functions in the primitive history definition are used to define the quintuple FSM definition ($\mathsf{Q}_{\mathsf{prim}}, \Sigma_{\mathsf{prim}}, \delta_{\mathsf{prim}}, \mathsf{s0}_{\mathsf{prim}}, \mathsf{F}_{\mathsf{prim}}$) at each time in T. Figure 2.3 shows the relationship between the sets and functions defined in the primitive model. The bold boxes around the edges are the sets and functions that are defined in the primitive history and that define the FSM. The boxes with dashed lines are sets whose content is defined based on other sets and they exist so that data is in the correct order for mapping functions. The boxes towards the middle with solid

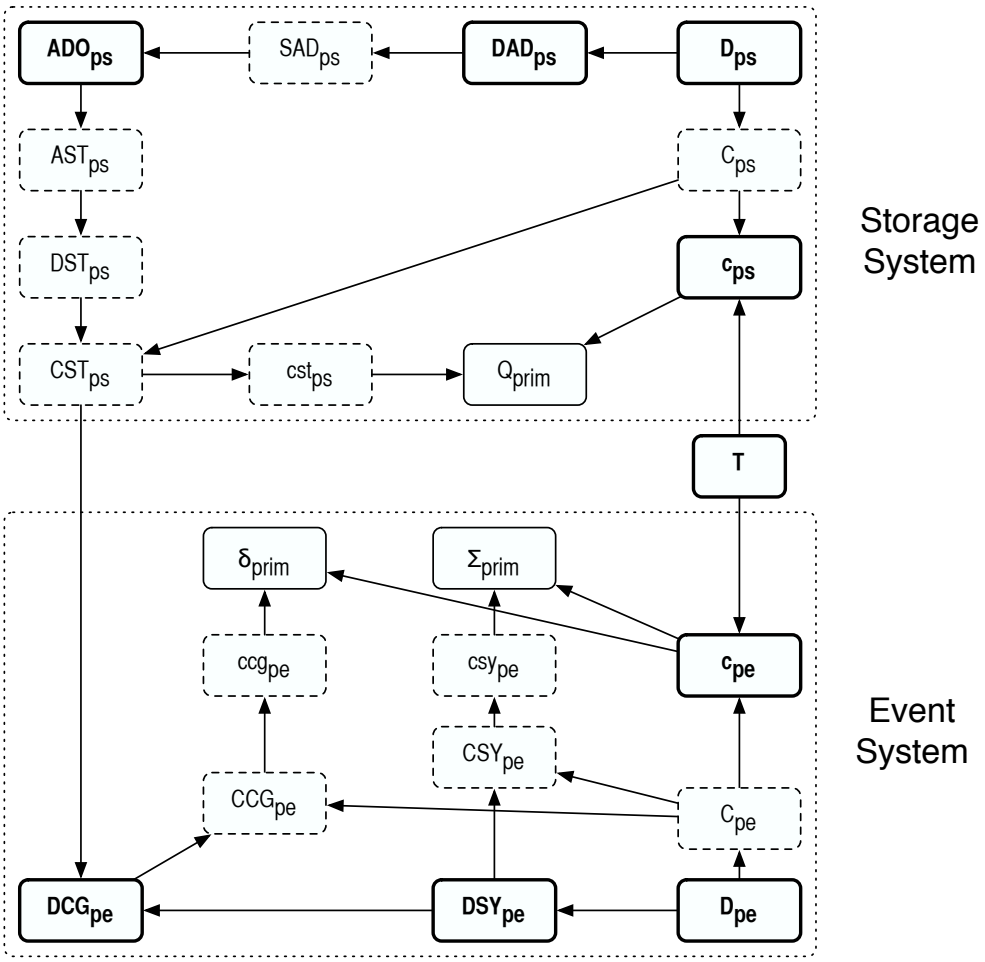

Figure 2.3. Graphical representation of the relationships between the sets and functions that are used to define the primitive finite state machine variables.

lines are the FSM variables. Each of the sets and functions in the history definition will be defined in the following sections.

2.3.1 Primitive State History

The history model must be capable of representing different types of storage devices that are connected at different times. To define the FSM Q_{prim} set, several sets and functions are defined that describe each storage device. The sets and functions

for the primitive storage system have a "ps" subscript. The next three sections describe how the capabilities of each primitive storage system are defined, how the possible states are determined, and how the primitive state history is defined.

Primitive System Definition

The model must define the possible states of a system. One way to do this would be to define a set that contains all possible states for each device, but this removes any concept of addresses. For example, each possible state of a hard disk would be one very large number and there would be no notion of sectors. Because addresses and locations are important in a digital investigation, a more complex approach is taken to defining possible states. This model defines the states of a storage device based on the number of addresses and the domain of each.

The unique names of devices that were ever connected to the system are defined in the D_{ps} set. A storage device is typically a physical unit, so two hard disks and main memory would be considered as three separate storage devices. For example, if a system had only the CPU registers, memory, two hard disks, and a removable USB device then its D_{ps} set would contain:

$$\{\text{CPU}, \text{memory}, \text{harddisk1}, \text{harddisk2}, \text{USB1}\}$$

The range of addresses for each device are defined in the $\mathsf{DAD}_{\mathsf{ps}}$ set. Each entry in $\mathsf{DAD}_{\mathsf{ps}}$ is a set of addresses that are supported by a device.

$$\mathsf{DAD}_{\mathsf{ps}} = \{\{\mathsf{a} : \mathsf{a} \text{ is an address in device } \mathsf{d}\} : \mathsf{d} \in \mathsf{D}_{\mathsf{ps}}\}$$

The function $\mathsf{dad}_{\mathsf{ps}}$ maps a storage device name to its set of addresses.

$$\mathsf{dad}_{\mathsf{ps}} : \mathsf{D}_{\mathsf{ps}} \rightarrow \mathsf{DAD}_{\mathsf{ps}}$$

For example, $\mathsf{dad}_{\mathsf{ps}}$(harddisk1) could map to the values $\{0, 1, \ldots, 2^{31} - 1\}$ for a 2 GB hard disk.

The $\mathsf{SAD}_{\mathsf{ps}}$ set contains the systemwide device and address pairs for all devices that were connected to the system.

$$\mathsf{SAD}_{\mathsf{ps}} = \{(\mathsf{d}, \mathsf{a}) : \mathsf{d} \in \mathsf{D}_{\mathsf{ps}} \text{ and } \mathsf{a} \in \mathsf{dad}_{\mathsf{ps}}(\mathsf{d})\}$$

Using the previous examples, this set would contain device and address pairs for the CPU, memory, harddisk1, harddisk2, and USB1 devices.

Each storage location address can have a different set of values that can be stored, which is called its domain. Each address has an entry in the $\mathsf{ADO}_{\mathsf{ps}}$ set and the entry contains the valid values for the location.

$$\mathsf{ADO}_{\mathsf{ps}} = \{\{\mathsf{v} : \mathsf{v} \in \mathcal{Z} \text{ and can be stored in address } \mathsf{a} \text{ on device } \mathsf{d}\} : (\mathsf{d}, \mathsf{a}) \in \mathsf{SAD}_{\mathsf{ps}}\}$$

The value $\mathsf{undefined}$ can be included in the domain for storage locations that do not have a value, such as when volatile memory does not have power or before the first value is written to a location. The function $\mathsf{ado}_{\mathsf{ps}}$ maps a device and address pair to the domain for the location.

$$\mathsf{ado}_{\mathsf{ps}} : \mathsf{SAD}_{\mathsf{ps}} \rightarrow \mathsf{ADO}_{\mathsf{ps}}$$

For example, $\mathsf{ado}_{\mathsf{ps}}(\text{harddisk1}, 0)$ is for the first 512-byte hard disk sector and it maps to the set $\{0, 1, \ldots, 2^{4096} - 1\}$.

The last function to be formally defined represents the storage devices that were connected at each time, which is called the *primitive storage device configuration*. This function requires the C_{ps} set, which is the power set of D_{ps} and it contains all unique subsets of the storage device names. The c_{ps} function maps a time to the set of devices that were connected at that time.

$$\mathsf{c}_{\mathsf{ps}} : \mathsf{T} \rightarrow \mathsf{C}_{\mathsf{ps}}$$

For example, it could map a time to the following set if the only storage devices at that time were the CPU registers, memory, and a hard disk.

$$\{\text{CPU}, \text{memory}, \text{harddisk1}\}$$

Possible Primitive States

Based on the number of addresses for each device, the domain of each address, and when each device was connected, the possible states can be defined. The possible states will be defined for each location, each device, and each storage configuration.

The possible states of each storage location address are defined in the $\mathsf{AST}_{\mathsf{ps}}$ set. This set has one entry for every address and the entry is a set that contains all possible device, address, and value triplets for that location.

$$\mathsf{AST}_{\mathsf{ps}} = \{\{(\mathsf{d}, \mathsf{a}, \mathsf{v}) : \mathsf{v} \in \mathsf{ado}_{\mathsf{ps}}(\mathsf{d}, \mathsf{a})\} : (\mathsf{d}, \mathsf{a}) \in \mathsf{SAD}_{\mathsf{ps}}\}$$

The $\mathsf{ast}_{\mathsf{ps}}$ function maps an address to its possible states.

$$\mathsf{ast}_{\mathsf{ps}} : \mathsf{SAD}_{\mathsf{ps}} \rightarrow \mathsf{AST}_{\mathsf{ps}}$$

An example of the $\mathsf{AST}_{\mathsf{ps}}$ set is shown here where each line is for a different address on two hard disks. Each triplet on each line is a different possible state for that address. In this case, the first line is for sector 0 of the first hard disk. The second triplet on the first line is the state for sector 0 having a value of 1.

```
{
  {(harddisk1,0,0),(harddisk1,0,1),...},
  {(harddisk1,1,0),(harddisk1,1,1),...},
  {(harddisk1,2,0),(harddisk1,2,1),...},
  ...
  {(harddisk2,0,0),(harddisk2,0,1),...},
  ...
}
```

Using the possible states for each location, the possible states for each device are defined in the $\mathsf{DST}_{\mathsf{ps}}$ set. This set contains an entry for every device and each entry is a set of possible device states. Because the state of a device is based on the state of each location in the device, each possible device state is a set of location states.

To formally define the possible states for each device, the Cartesian product of all possible address states on the device is calculated.

$$\mathsf{DST_{ps}} = \left\{ \left\{ \bigtimes_{\mathsf{a} \in \mathsf{dad_{ps}(d)}} \mathsf{ast_{ps}(d, a)} \right\} : \mathsf{d} \in \mathsf{D_{ps}} \right\}$$

The $\mathsf{dst_{ps}}$ function maps a device to its possible states.

$$\mathsf{dst_{ps}} : \mathsf{D_{ps}} \rightarrow \mathsf{DST_{ps}}$$

An example of the $\mathsf{DST_{ps}}$ set is shown here for two hard disk devices. The first line contains a possible state for the first hard disk and the second line is another possible state, in this case the value of sector 0 changed from 0 to 1 between the two lines.

```
{
  { {(harddisk1,0,0),(harddisk1,1,0),(harddisk1,2,0),...},
    {(harddisk1,0,1),(harddisk1,1,0),(harddisk1,2,0),...},
    ...
  },
  { {(harddisk2,0,0),(harddisk2,1,0),(harddisk2,2,0),...},
    {(harddisk2,0,1),(harddisk2,1,0),(harddisk2,2,0),...},
    ...
  },
  ...
}
```

The last set that needs to be defined contains the possible system states for each storage device configuration. The set $\mathsf{CST_{ps}}$ contains an entry for every storage device configuration and the entry is a set that contains the possible states for the corresponding configuration. This set is defined by calculating the Cartesian product of the possible states for each of the devices in the configuration.

$$\mathsf{CST_{ps}} = \left\{ \left\{ \bigtimes_{\mathsf{d} \in \mathsf{C}} \mathsf{dst_{ps}(d)} \right\} : \mathsf{C} \in \mathsf{C_{ps}} \right\}$$

An example of the $\mathsf{CST}_{\mathsf{ps}}$ set is shown here where two storage device configurations are shown. The first configuration is for only the first hard disk and the second configuration is for both the first and second hard disks. Two states are shown for each configuration. The two states for the first configuration are shown on the first two lines. The two states for the second configuration are shown on the bottom four lines and each state requires two lines. The difference between the two states is that sector 0 of the first hard disk changed from 0 to 1.

```
{
  { {(harddisk1,0,0),(harddisk1,1,0),(harddisk1,2,0),...},
    {(harddisk1,0,1),(harddisk1,1,0),(harddisk1,2,0),...},
    ...
  },
  { {(harddisk1,0,0),(harddisk1,1,0),(harddisk1,2,0),...,
     (harddisk2,0,0),(harddisk2,1,0),(harddisk2,2,0),...},
    {(harddisk1,0,1),(harddisk1,1,0),(harddisk1,2,0),...,
     (harddisk2,0,0),(harddisk2,1,0),(harddisk2,2,0),...},
    ...
  },
  ...
}
```

The function $\mathsf{cst}_{\mathsf{ps}}$ maps a storage device configuration to its possible states.

$$\mathsf{cst}_{\mathsf{ps}} : \mathsf{C}_{\mathsf{ps}} \rightarrow \mathsf{CST}_{\mathsf{ps}}$$

In the previous discussions about FSMs, the function $\mathsf{Q}_{\mathsf{prim}}(\mathsf{t})$ was defined as mapping a time to a set of possible states. Using the sets and functions defined for each storage device, $\mathsf{Q}_{\mathsf{prim}}(\mathsf{t})$ is equivalent to $\mathsf{cst}_{\mathsf{ps}}\left(\mathsf{c}_{\mathsf{ps}}(\mathsf{t})\right)$.

$$\mathsf{Q}_{\mathsf{prim}}(\mathsf{t}) \equiv \mathsf{cst}_{\mathsf{ps}}\left(\mathsf{c}_{\mathsf{ps}}(\mathsf{t})\right)$$

In summary, the definition of $\mathsf{Q}_{\mathsf{prim}}(\mathsf{t})$ is based on the devices that are connected at each time and the number and size of their storage locations.

Section 2.4.2 defines the storage capabilities for an abstract system, which will require a set that contains all sets of primitive storage addresses. The $\mathsf{SEQ}_{\mathsf{ps}}$ set is a power set of $\mathsf{SAD}_{\mathsf{ps}}$ and it contains all subsets of the systemwide storage locations.

Primitive History Definition

The *primitive state history* of the system is defined for each time $\mathsf{t} \in \mathsf{T}$. It is represented by a function h_{ps} that maps a time to a state in $\mathsf{Q}_{\mathsf{prim}}(\mathsf{t})$ if and only if that state existed at time t.

$$\begin{aligned} \mathsf{h}_{\mathsf{ps}} : & \mathsf{T} \rightarrow \mathsf{Q}_{\mathsf{prim}}(\mathsf{t}) \\ & \mathsf{T} \rightarrow \mathsf{cst}_{\mathsf{ps}}\left(\mathsf{c}_{\mathsf{ps}}(\mathsf{t})\right) \end{aligned}$$

2.3.2 Primitive Event History

The primitive state of the system changes as a result of primitive events. The possible primitive events at each time are based on the event devices that exist. The sets and functions for the primitive event system will have a "pe" subscript. The next two sections describe the definition of the primitive event system and the third section describes the primitive event history.

Primitive System Definition

The definition for each event device consists of a unique device name, a set of event symbols, and a state change function. The unique name of each device is stored in the D_{pe} set. Example event devices include the CPU, MMU, and PCI controllers.

Each event device supports a finite number of events, each of which has a unique symbol. The $\mathsf{DSY}_{\mathsf{pe}}$ set contains an entry for each event device and the entry is the set of all supported event symbols for the device.

$$\mathsf{DSY}_{\mathsf{pe}} = \{\{\mathsf{s} : \mathsf{s} \text{ is an event symbol for device } \mathsf{d}\} : \mathsf{d} \in \mathsf{D}_{\mathsf{pe}}\}$$

The function $\mathsf{dsy}_{\mathsf{pe}}$ maps an event device to its possible symbols.

$$\mathsf{dsy}_{\mathsf{pe}} : \mathsf{D}_{\mathsf{pe}} \rightarrow \mathsf{DSY}_{\mathsf{pe}}$$

Each event has a corresponding state change function. The set $\mathsf{DCG}_{\mathsf{pe}}$ contains an entry for every device and the entry is a state change function for every starting state and event symbol supported by the event device.

$$\mathsf{DCG}_{\mathsf{pe}} = \{\{\mathsf{f} : \mathsf{f} \text{ is a state change function for device } \mathsf{d}\} : \mathsf{d} \in \mathsf{D}_{\mathsf{pe}}\}$$

The function $\mathsf{dcg}_{\mathsf{pe}}$ maps an event device to its state change function.

$$\mathsf{dcg}_{\mathsf{pe}} : \mathsf{D}_{\mathsf{pe}} \rightarrow \mathsf{DCG}_{\mathsf{pe}}$$

The *primitive event device configuration* is defined as the set of event devices that are connected to the system at a given time. To define the event device configuration, the C_{pe} set is defined, which is a power set of D_{pe} and contains all subsets of the event device names. The configuration is defined by the c_{pe} function, which maps a time to the set of devices that were connected at that time.

$$\mathsf{c}_{\mathsf{pe}} : \mathsf{T} \rightarrow \mathsf{C}_{\mathsf{pe}}$$

Possible Events and State Changes

Based on information about the event devices and when each was connected, the possible events and state changes can be defined. These will be defined based on event device configurations.

To determine the symbols for an event device configuration, the $\mathsf{CSY}_{\mathsf{pe}}$ set is defined. The set has an entry for every event device configuration and the entry contains the set of events that could occur based on the devices connected.

$$\mathsf{CSY}_{\mathsf{pe}} = \{\{\mathsf{s} : \mathsf{s} \in \mathsf{dsy}_{\mathsf{pe}}(\mathsf{d}) \text{ and } \mathsf{d} \in \mathsf{C}\} : \mathsf{C} \in \mathsf{C}_{\mathsf{pe}}\}$$

The function $\mathsf{csy}_{\mathsf{pe}}$ maps an event device configuration to the set of systemwide events that could occur.

$$\mathsf{csy}_{\mathsf{pe}} : \mathsf{C}_{\mathsf{pe}} \rightarrow \mathsf{CSY}_{\mathsf{pe}}$$

Note that this assumes that the symbols for each device are unique and that there are no collisions. For programmable devices with instructions that take arguments, such as register addresses, each unique argument is considered a unique event if the argument will cause a unique state change. In addition to instructions, hardware interrupts must also be included in the set of events because they cause a state change.

The FSM $\Sigma_{\mathsf{prim}}(\mathsf{t})$ function maps a time to the set of possible event symbols. Using the previous definitions, it is equivalent to $\mathsf{csy}_{\mathsf{pe}}\left(\mathsf{c}_{\mathsf{pe}}(\mathsf{t})\right)$, which maps the event device configuration at time t to a set of symbols.

$$\Sigma_{\mathsf{prim}}(\mathsf{t}) \equiv \mathsf{csy}_{\mathsf{pe}}\left(\mathsf{c}_{\mathsf{pe}}(\mathsf{t})\right)$$

The state change function for each system configuration is determined next. This mapping requires the $\mathsf{CCG}_{\mathsf{pe}}$ set, which contains an entry for every device configuration. Each entry is a state change function for every starting state and event symbol supported by the event device configuration.

$$\mathsf{CCG}_{\mathsf{pe}} = \{\mathsf{f} : \mathsf{f} \text{ is equivalent to all } \mathsf{f}' \in \mathsf{dcg}_{\mathsf{pe}}(\mathsf{d}) \text{ and } \mathsf{d} \in \mathsf{C} \text{ and } \mathsf{C} \in \mathsf{C}_{\mathsf{pe}}\}$$

The function $\mathsf{ccg}_{\mathsf{pe}}$ maps a configuration to the corresponding state change function:

$$\mathsf{ccg}_{\mathsf{pe}} : \mathsf{C}_{\mathsf{pe}} \rightarrow \mathsf{CCG}_{\mathsf{pe}}$$

The FSM definition has a function $\delta_{\mathsf{prim}}(\mathsf{t})$ that maps a time to the state change function. Using the previous definitions, this is equivalent to $\mathsf{ccg}_{\mathsf{pe}}\left(\mathsf{c}_{\mathsf{pe}}(\mathsf{t})\right)$, which the event device configuration at time t to a state change function.

$$\delta_{\mathsf{prim}}(\mathsf{t}) \equiv \mathsf{ccg}_{\mathsf{pe}}\left(\mathsf{c}_{\mathsf{pe}}(\mathsf{t})\right)$$

This work assumes that all state changes are caused by events and not by random hardware faults, which occur when the state of a location changes when it is not defined by an event. To account for random hardware faults, special event symbols could be added that include the unaccounted for state change.

When later discussing analysis techniques, the locations that each event reads from and writes to will need to be referred to. For this, the "use" and "def" terminology from programming languages are used. The $\mathsf{use}_{\mathsf{prim}}$ function maps an alphabet symbol in $\Sigma_{\mathsf{prim}}(\mathsf{t})$ to a set of storage locations that are read from during the event. For this mapping, the set $\mathsf{USE}_{\mathsf{ps}}$ is defined, which contains all sets of the systemwide storage locations in $\mathsf{SAD}_{\mathsf{ps}}$ that are used by events.

$$\mathsf{use}_{\mathsf{prim}} : \Sigma_{\mathsf{prim}} \rightarrow \mathsf{USE}_{\mathsf{ps}}$$

Similarly, the $\mathsf{def}_{\mathsf{prim}}$ function maps an alphabet symbol in $\Sigma_{\mathsf{prim}}(\mathsf{t})$ to an entry in $\mathsf{DEF}_{\mathsf{ps}}$, which contains all sets of systemwide storage locations that are defined by events.

$$\mathsf{def}_{\mathsf{prim}} : \Sigma_{\mathsf{prim}} \rightarrow \mathsf{DEF}_{\mathsf{ps}}$$

In summary, the event system is defined by the event devices that exist at each time and the symbols and state transitions that each support.

Primitive History Definition

The *primitive event history* of a system is represented with the h_{pe} function, which maps the time t to an event symbol.

$$\begin{aligned} \mathsf{h}_{\mathsf{pe}} : \mathsf{T} &\rightarrow \Sigma_{\mathsf{prim}}(\mathsf{t}) \\ \mathsf{T} &\rightarrow \mathsf{csy}_{\mathsf{pe}}(\mathsf{c}_{\mathsf{pe}}(\mathsf{t})) \end{aligned}$$

$\mathsf{h}_{\mathsf{pe}}(\mathsf{t})$ is equal to e if and only if e occurred at or after time t but before $\mathsf{t} + \Delta\mathsf{t}$. The resulting state change is instantaneous. Using the primitive data and event histories, it can observed that the state in the primitive state history at time t is the effect of the event in the primitive event history at time $\mathsf{t} - \Delta\mathsf{t}$ using the state change function mapped to by $\delta_{\mathsf{prim}}(\mathsf{t} - \Delta\mathsf{t})$:

$$\mathsf{h}_{\mathsf{ps}}(\mathsf{t}) = \delta_{\mathsf{prim}}\,(\mathsf{t} - \Delta\mathsf{t})\,(\mathsf{h}_{\mathsf{ps}}(\mathsf{t} - \Delta\mathsf{t}), \mathsf{h}_{\mathsf{pe}}(\mathsf{t} - \Delta\mathsf{t}))$$

2.3.3 Primitive History Model Proofs

This section provides proofs with regard to the necessity and sufficiency of the primitive history model.

Theorem 2.3.1 *If the duration* Δt *between values in* T *is defined to be less than or equal to the smallest possible duration between state changes then either one or zero events can occur between* t *and* $t + \Delta t$.

Proof If two events occur between t and $t + \Delta t$ then the duration between their state changes is less than Δt. By definition of the values in T, this is not possible. ■

Theorem 2.3.2 *If the duration* Δt *between values in* T *is defined to be less than or equal to the smallest possible duration between state changes,* $s = h_{ps}(t)$, *and* $s' = h_{ps}(t + \Delta t)$ *then no state* s'' *could have existed between* t *and* $t + \Delta t$ *where* $s'' \neq s$ *and* $s'' \neq s'$.

Proof If state s'' existed between t and $t + \Delta t$ that was different from the states at t and $t + \Delta t$ then two state changes must have occurred between t and $t + \Delta t$: one change from s to s'' and the other change from s'' to s'. By Theorem 2.3.1 this is not possible. ■

Theorem 2.3.3 *The 11 variable primitive history model defined in Equation 2.1 is necessary to represent the primitive history of a FSM for the times in* T .

Proof The necessity condition requires that a primitive state or event occurred during the times in T only if it is represented in the primitive history model. This is proved by contradiction.

If primitive event e occurred between times $t \in T$ and $t + \Delta t \in T$ and $h_{pe}(t) \neq e$ then either $e \notin \Sigma_{prim}(t)$ or multiple events occurred between t and $t + \Delta t$. If $e \notin \Sigma_{prim}(t)$ than the event could not have occurred. Theorem 2.3.1 proved that

by the definition of the values in T that only one event can occur for each $\mathsf{t} \in \mathsf{T}$ and therefore if $\mathsf{h_{pe}}(\mathsf{t}) \neq e$ then event e did not occur at time t.

If primitive state s existed at time t' then there are three situations. The first is if $\mathsf{t}' \in \mathsf{T}$, $\mathsf{s} \in \mathsf{Q_{prim}}$, and $\mathsf{h_{ps}}(\mathsf{t}') \neq \mathsf{s}$ then a conflict exists about which state existed and this is not possible. If $\mathsf{t} < \mathsf{t}' < \mathsf{t} + \Delta\mathsf{t}$ where $\mathsf{t} \in \mathsf{T}$ and $\mathsf{t} + \Delta\mathsf{t} \in \mathsf{T}$, $\mathsf{h_{ps}}(\mathsf{t}) \neq \mathsf{s}$, $\mathsf{h_{ps}}(\mathsf{t} + \Delta\mathsf{t}) \neq \mathsf{s}$, and $\mathsf{s} \in \mathsf{Q_{prim}}$ then Theorem 2.3.2 shows that this is not possible because this would require two state changes during $\Delta\mathsf{t}$. Finally, if $\mathsf{s} \notin \mathsf{Q_{prim}}$ then the state could not have existed. Therefore if $\mathsf{h_{ps}}(\mathsf{t}) \neq \mathsf{s}$ then state s did not exist at time t. ■

Theorem 2.3.4 *The 11 variable primitive history model defined in Equation 2.1 is sufficient to represent the primitive history of a FSM for the times in* T .

Proof The sufficiency condition requires that a state or event be represented in the model only if it existed or occurred.

Let $\mathsf{e} \in \Sigma_{\mathsf{prim}}$ be a primitive event in the primitive event history where $\mathsf{h_{pe}}(\mathsf{t}) = \mathsf{e}$, but e did not occur before $\mathsf{t} + \Delta\mathsf{t}$ and at or after t. By definition of $\mathsf{h_{pe}}$, this is not possible and e must have occurred if it is in $\mathsf{h_{pe}}$.

Let $\mathsf{s} \in \mathsf{Q_{prim}}$ be a primitive state in the primitive state history where $\mathsf{h_{ps}}(\mathsf{t}) = \mathsf{s}$, but s did not exist at time t. By definition of $\mathsf{h_{ps}}$, this is not possible and s must have existed if it is in $\mathsf{h_{ps}}$. ■

2.4 The Ideal Complex History

2.4.1 Overview

Section 2.3 outlined a model for low-level events and storage locations. While these are the fundamental events and storage methods, they are not useful when considering realistic systems that execute billions of instructions per second and store billions of bytes of data. Real systems are complex and are designed with

several layers of data and event abstraction to hide the lower-level details. Therefore the history model must be able to take system abstractions into account because an investigation is typically concerned with complex storage locations, such as files.

Complex Events

A *complex event* is a state transition that causes one or more lower-level complex or primitive events to occur. Complex events exist to make the development of programs easier because they hide the implementation details. Systems typically have multiple layers of complex events. Complex event e_1 is at a higher-level than another complex or primitive event e_2 if:

- e_1 started at the same time or before the start of e_2
- e_1 ended after or at the same time as e_2
- e_2 would not have occurred if e_1 did not occur

A complex event is an alternative representation for a sequence of lower-level event and it allows us to represent the sequence by a single event. Multiple representations for the same sequence could be possible. One representation for a sequence of events could be a user-level event, which are events that are directly caused by a user when she runs a program or enters input. For example, pressing a button is one event to a user, but many primitive-level events actually occur. The same sequence of events could also be represented by the programming language statements that were used to implement the program. Another representation could use a different programming language that was not even used during the implementation of the program. There could be many equivalent representations for a sequence of events and all are equally valid unless the investigation needs to answer questions about specific complex events. In which case, the investigation should use the representation that contains the events in question or a representation that is equivalent to it.

Complex Storage Locations

An *complex storage location* is a virtual storage location with one or more attributes, which are name and value pairs. In some cases, only a subset of an attribute's possible values may be considered valid. A complex storage location exists only because a program virtually created it. The attribute values are stored in primitive storage locations after undergoing one or more transformations. The possible transformations are dependent on the system's primitive events. Complex storage locations are used in practice because primitive storage locations have a fixed size, which may be smaller or larger than what needs to be stored. They are also used to partition and bind labels to data. The *complex state* is defined as the value of all complex storage locations that exist in the system.

Based on system design, not all layers of complex events may have access to the same complex storage locations. This is because abstraction hides underlying details. Therefore, higher-level complex events may not have direct access to the lower-level storage locations.

Complex Systems

Each major level of complex events can be represented with a FSM. Complex storage locations and events rely on their primitive counterparts and therefore new FSM quintuples will be defined for the complex systems when a new FSM quintuple is defined for the primitive system.

The FSM for each level is defined with a quintuple $(\mathsf{Q_X}, \Sigma_\mathsf{X}, \delta_\mathsf{X}, \mathsf{s0_X}, \mathsf{F_X})$, where X is replaced by the name of the level. Example names include *impl* for implementation level and *user* for user level. Functions are defined that map the time to the relevant FSM set or function. For example, $\mathsf{Q_X(t)}$ maps the time to a set of complex states for level X.

A graphical representation of complex storage locations and events can be seen in Figure 2.4. It shows complex event $E1$ that has lower-level events $E1a$ and $E1b$.

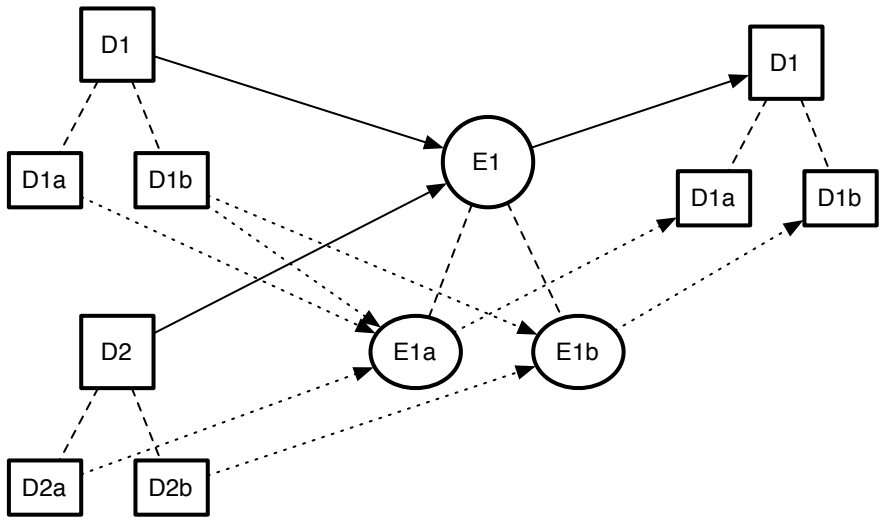

Figure 2.4. Representation of a complex event $E1$ with two complex cause locations and one complex effect location.

The event uses locations $D1$ and $D2$ and defines location $D1$. The lower-level events of $E1$ also use and define lower-level storage locations.

The *complex history* of a system is the complete sequence of complex states and events. The complex history model for a system is defined by a tuple with 17 variables:

$$(\mathsf{T}, \mathsf{L}, \mathsf{D_{cs}}, \mathsf{DAD_{cs}}, \mathsf{DAT_{cs}}, \mathsf{ADO_{cs}}, \mathsf{ABS_{cs}}, \mathsf{MAT_{cs}}, \mathsf{c_{cs-X}}, \mathsf{h_{cs}},$$
$$\mathsf{D_{ce}}, \mathsf{DSY_{ce-X}}, \mathsf{DCG_{ce-X}}, \mathsf{ABS_{ce}}, \mathsf{MAT_{ce}}, \mathsf{c_{ce}}, \mathsf{h_{ce}}) \tag{2.2}$$

The set T is the same as the primitive history and the set L contains the names of the complex layers. For example, it could contain:

$$\{\text{user}, \text{impl}\}$$

The $\mathsf{D_{cs}}$, $\mathsf{DAD_{cs}}$, $\mathsf{DAT_{cs}}$, $\mathsf{ADO_{cs}}$, $\mathsf{ABS_{cs}}$, $\mathsf{MAT_{cs}}$, and $\mathsf{c_{cs-X}}$ sets and functions are used to define the complex storage system. These define the attributes and transformation

functions for each complex storage location type. The D_{ce}, $\mathsf{DSY}_{\mathsf{ce-X}}$, $\mathsf{DCG}_{\mathsf{ce-X}}$, $\mathsf{ABS}_{\mathsf{ce}}$, $\mathsf{MAT}_{\mathsf{ce}}$, c_{ce} sets and functions are used to define the complex event system.

Figure 2.5 shows the relationship between the sets and functions defined in the complex system. The bold boxes around the edges are sets and functions that must be defined for each system and are used to define the FSM variables. The boxes with dashed lines are sets that are defined using data from other sets and they are needed to arrange data in the proper ordering for the mapping functions. The boxes towards the middle with solid lines are the FSM variables. The following sections address each of these sets and functions.

2.4.2 Complex State History

To account for different complex storage types that may exist at each time, several sets and functions must be defined. The function and set names will be similar to those defined for the primitive state, but the subscript is "cs" instead of "ps." The first and second sections describe the complex state system definition, the third describes the transformation functions, and the fourth describes the complex state history.

System Definition

For each complex storage type, definitions are needed for its unique name, the names of its attributes, the domain of each attribute, and its transformation functions. The first three definitions in the preceding list are used to define the Q_X set and the last is used to represent the required transformations between layers.

The D_{cs} set contains the unique names of the complex storage types that are supported by the system. Note that a system may contain data for a complex storage type but not be able to process and transform it. For example, a system receives a file but does not have any programs to interpret it. In this case, the

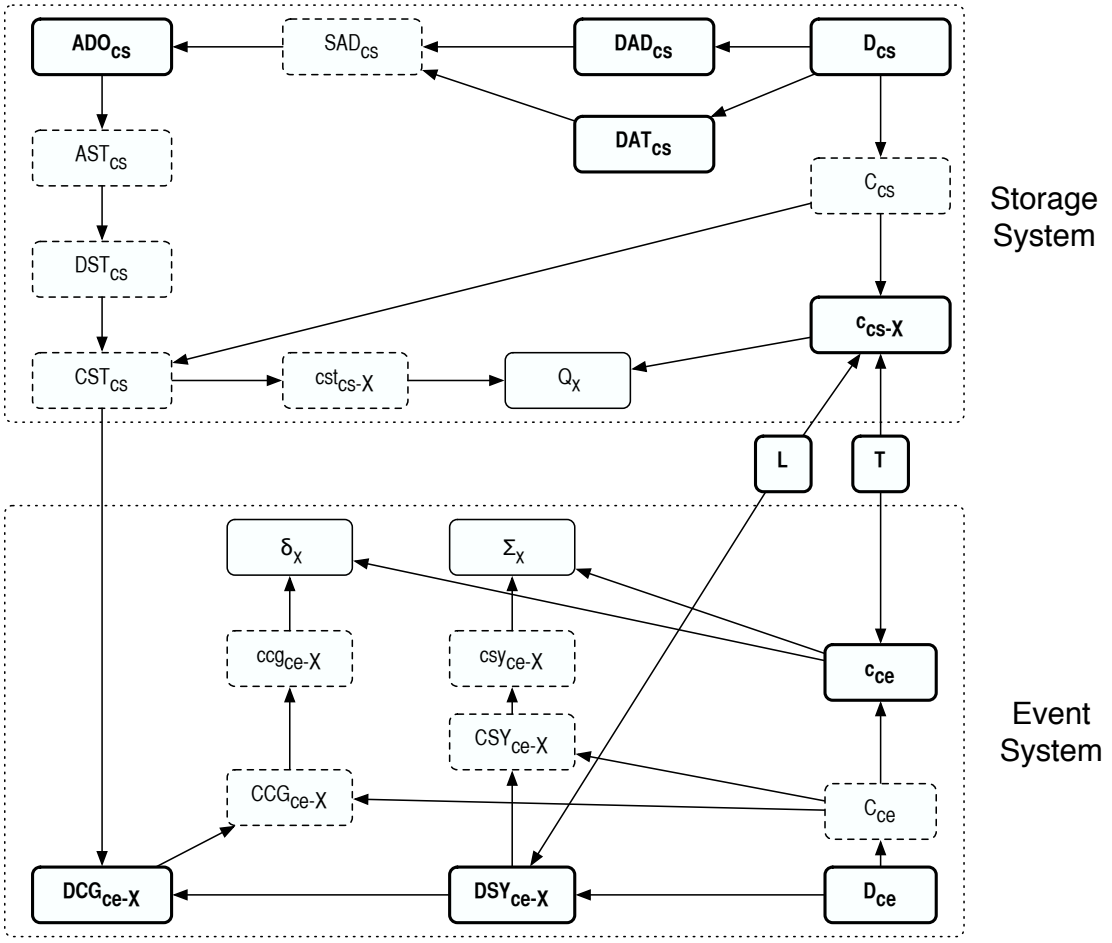

Figure 2.5. Graphical representation of the relationships of the sets and functions in the complex finite state machine.

complex storage type would not exist in the $\mathsf{D_{cs}}$ set because the system has no way of using the data as a complex storage location.

Each complex storage type has one or more attributes, which are name and value pairs. The set $\mathsf{DAT_{cs}}$ contains an entry for every complex storage type and the entry contains a list of attribute names.

$$\mathsf{DAT_{cs}} = \{\{\mathsf{n} : \mathsf{n} \text{ is an attribute name for type } \mathsf{d}\} : \mathsf{d} \in \mathsf{D_{cs}}\}$$

For example, the complex storage type may have attributes for *name*, *size*, and *date*. Its entry in $\mathsf{DAT}_{\mathsf{cs}}$ would be:

$$\{\text{name}, \text{size}, \text{date}\}$$

The $\mathsf{dat}_{\mathsf{cs}}$ function maps a complex storage type to a set of attribute names.

$$\mathsf{dat}_{\mathsf{cs}} : \mathsf{D}_{\mathsf{cs}} \rightarrow \mathsf{DAT}_{\mathsf{cs}}$$

Each attribute can have a unique domain, which contains the values that are valid for that location. To define a mapping from an attribute address to its domain, the systemwide addresses must be defined.

Complex storage locations do not have addresses as primitive storage locations do. Instead, a complex storage location is uniquely identified using its type and the lower-level data that are being transformed. At higher-levels, this method of identifying a complex storage location gets longer because each layer contains the addresses of *all* lower-level layers. To make it simpler, a unique integer i $(0 \leq i < n)$ is bound to each possible instance of a complex storage type, where n is the maximum number of possible instances of the complex storage type. The complex storage type and integer will be used as an address and a function ($\mathsf{iad}_{\mathsf{cs}}$) is later defined that maps the integer to lower-level addresses. Note that the integer is not a required part of the model, but exists for convenience and readability.

To account for the additional integer address that exists for each complex storage type, the $\mathsf{DAD}_{\mathsf{cs}}$ set is defined, which contains an entry for every complex storage type and the entry contains the set of addresses that are possible for the type.

$$\mathsf{DAD}_{\mathsf{cs}} = \{\{\mathsf{a} : \mathsf{a} \in \mathcal{Z} \text{ and is an address for type } \mathsf{d}\} : \mathsf{d} \in \mathsf{D}_{\mathsf{cs}}\}$$

For example, if there are 2^{32} possible instances of a complex storage type, then its entry would have the numbers 0 to $2^{32} - 1$. The $\mathsf{dad}_{\mathsf{cs}}$ function maps a complex storage type to its set of addresses.

$$\mathsf{dad}_{\mathsf{cs}} : \mathsf{D}_{\mathsf{cs}} \rightarrow \mathsf{DAD}_{\mathsf{cs}}$$

The $\mathsf{SAD_{cs}}$ set contains the systemwide complex storage type, address, and attribute tuples for all complex storage types that existed on the system.

$$\mathsf{SAD_{cs}} = \{(\mathsf{d}, \mathsf{ad}, \mathsf{at}) : \mathsf{d} \in \mathsf{D_{cs}} \text{ and } \mathsf{ad} \in \mathsf{dad_{cs}}(\mathsf{d}) \text{ and } \mathsf{at} \in \mathsf{dat_{cs}}(\mathsf{d})\}$$

The domain for each attribute is defined in the set $\mathsf{ADO_{cs}}$, which has an entry for every attribute and the entry contains the list of valid values.

$$\mathsf{ADO_{cs}} = \{\{\mathsf{v} : \mathsf{v} \in \mathcal{Z} \text{ and can be stored in attribute } (\mathsf{d}, \mathsf{ad}, \mathsf{at})\} : (\mathsf{d}, \mathsf{ad}, \mathsf{at}) \in \mathsf{SAD_{cs}}\}$$

The $\mathsf{ado_{cs}}$ function maps an attribute address triplet to its range in $\mathsf{ADO_{cs}}$.

$$\mathsf{ado_{cs}} : \mathsf{SAD_{cs}} \rightarrow \mathsf{ADO_{cs}}$$

The previous sets and functions defined the capabilities of each complex storage type. Now, the system is considered, which requires a mapping from each time to the list of complex storage types that existed. The $\mathsf{C_{cs}}$ set is a power set of $\mathsf{D_{cs}}$ and it contains all subsets of the complex storage type names. The *complex storage configuration* of the system is the set of complex storage types that are supported at each time. The function $\mathsf{c_{cs-X}}$ maps a time to the set of types that were supported at that time at layer $\mathsf{X} \in \mathsf{L}$:

$$\mathsf{c_{cs-X}} : \mathsf{T} \rightarrow \mathsf{C_{cs}}$$

To account for the integer addresses that were added to the model to bind an address to each complex storage location, a function $\mathsf{iad_{cs}}$ is defined to map a complex storage type and a unique integer address to the lower-level primitive and complex data used to define the complex storage location. This requires the $\mathsf{SEQ_{cs}}$ set, which is a power set of $\mathsf{SAD_{cs}}$ and it contains all subsets of the systemwide storage.

$$\mathsf{iad_{cs}} : \mathsf{D_{cs}} \times \mathsf{AD_{cs}} \rightarrow \mathsf{SEQ_{ps}} \times \mathsf{SEQ_{cs}}$$

Possible Complex States

Based on the number of addresses for each complex storage type, the attributes for each type, the domain of each attribute, and when each type existed, the possible

states can be defined. The possible states will be defined for each attribute, each type, and each storage configuration.

The possible states of each attribute are defined in the $\mathsf{AST}_{\mathsf{cs}}$ set. This set has one entry for every attribute address and the entry is a set that contains all possible device, address, attribute, and value tuples for that location.

$$\mathsf{AST}_{\mathsf{cs}} = \{\{(\mathsf{d}, \mathsf{ad}, \mathsf{at}, \mathsf{v}) : \mathsf{v} \in \mathsf{ado}_{\mathsf{cs}}(\mathsf{d}, \mathsf{ad}, \mathsf{at})\} : (\mathsf{d}, \mathsf{ad}, \mathsf{at}) \in \mathsf{SAD}_{\mathsf{cs}}\}$$

The $\mathsf{ast}_{\mathsf{cs}}$ function maps an attribute address to its possible states.

$$\mathsf{ast}_{\mathsf{cs}} : \mathsf{SAD}_{\mathsf{cs}} \rightarrow \mathsf{AST}_{\mathsf{cs}}$$

Using the possible states for each attribute, the possible states for each complex storage type are defined in the $\mathsf{DST}_{\mathsf{cs}}$ set. This set contains an entry for every complex storage type and each entry is a set of possible states. Because the state of a complex storage location is based on the state of its attributes, each possible complex storage location state is a set of attribute states. To formally define the possible states for each complex storage type, the Cartesian product of all possible attribute states is calculated.

$$\mathsf{DST}_{\mathsf{cs}} = \left\{ \left\{ \bigtimes_{\mathsf{ad} \in \mathsf{dad}_{\mathsf{cs}}(\mathsf{d}) \text{ and } \mathsf{at} \in \mathsf{dat}_{\mathsf{cs}}(\mathsf{d})} \mathsf{ast}_{\mathsf{cs}}(\mathsf{d}, \mathsf{ad}, \mathsf{at}) \right\} : \mathsf{d} \in \mathsf{D}_{\mathsf{cs}} \right\}$$

The $\mathsf{dst}_{\mathsf{cs}}$ function maps a complex storage type to its possible states.

$$\mathsf{dst}_{\mathsf{cs}} : \mathsf{D}_{\mathsf{cs}} \rightarrow \mathsf{DST}_{\mathsf{cs}}$$

The possible states for each complex storage configuration are now defined. The set $\mathsf{CST}_{\mathsf{cs}}$ contains an entry for every complex storage configuration and each entry is a set that contains all possible states for that configuration.

$$\mathsf{CST}_{\mathsf{cs}} = \left\{ \left\{ \bigtimes_{\mathsf{d} \in \mathsf{C}} \mathsf{dst}_{\mathsf{cs}}(\mathsf{d}) \right\} : \mathsf{C} \in \mathsf{C}_{\mathsf{cs}} \right\}$$

The function $\mathsf{cst}_{\mathsf{cs}}$ maps a complex storage configuration to the possible states that could exist.

$$\mathsf{cst}_{\mathsf{cs}} : \mathsf{C}_{\mathsf{cs}} \rightarrow \mathsf{CST}_{\mathsf{cs}}$$

In the previous discussions about FSMs, the function $Q_X(t)$ was defined as mapping a time to a set of possible complex states. Using the previously defined sets and functions, $Q_X(t)$ is equivalent to $cst_{cs}(c_{cs-X}(t))$, which maps the complex storage configuration at time t to the possible states.

$$Q_X(t) \equiv cst_{cs}(c_{cs-X}(t))$$

Transformation Function Definition

The transformation associated with each complex storage type is represented by a function. The function maps the values of one or more lower-level locations to the values of the complex storage location attributes. Each complex storage type has a different number of lower-level sources that it reads from and attributes that it defines.

The set ABS_{cs} contains an entry for each complex storage type and each entry is an abstraction transformation function. The abs_{cs} function maps a complex storage type to its abstraction transformation function.

$$abs_{cs} : D_{cs} \rightarrow ABS_{cs}$$

Similarly, the data materialization process is represented by a function that maps the attributes of the complex storage layer to lower-level data. Because the abstraction process removes lower-level details, the materialization process may not have access to the lost details. Therefore, the materialization function may return a set of possible lower-level states.

The materialization functions are stored in the MAT_{cs} set. Each complex storage type has an entry in this set and the entry contains the materialization function. The function mat_{cs} maps a complex storage type to its materialization transformation function.

$$mat_{cs} : D_{cs} \rightarrow MAT_{cs}$$

History Definition

The *complex state history* for a system is defined for each time value $\mathsf{t} \in \mathsf{T}$ and it represents the state of all complex storage locations at all levels. The function $\mathsf{h_{cs}}$ maps a time value to a complex state, which is defined based on all of the complex storage types that exist at all levels. The history maps to a state if and only if that state existed at time t.

$$\mathsf{h_{cs}} : \mathsf{T} \rightarrow \left\{ \mathsf{cst_{cs}} \left(\bigcup_{\mathsf{X} \in \mathsf{L}} \mathsf{c_{cs-X}}(\mathsf{t}) \right) \right\}$$

2.4.3 Complex Event History

A complex event causes a sequence of lower-level events to occur and the possible complex events are dependent on the programs that exist at each time. The next four sections define sets and functions to represent the complex event system of the model. The set and function names for the complex event system have the subscript "ce." The first section describes the complex event system definition, the second describes the transformation functions, and the third describes the history model.

System Definition

The complex event system is defined by the programs that exist, the event symbols for each program, the state change function for each program, and the mapping of which programs existed at each time. This section defines sets and functions to represent these variables.

The set $\mathsf{D_{ce}}$ contains the names of programs that existed on the system and could cause complex events. Each version of a program must have a unique name, which may include a version number or it could be a number that is calculated based on the instructions in the program.

Each program can cause a finite number of complex events and each event has a symbol associated with it. The $\mathsf{DSY_{ce-X}}$ set exists for each layer $\mathsf{X} \in \mathsf{L}$, where the

X in the subscript is replaced by the layer name. Each program has an entry in the set and it contains the set of complex event symbols that the program can cause at layer X.

$$\mathsf{DSY_{ce-X}} = \{\{\mathsf{s} : \mathsf{s} \text{ is an event symbol for program } \mathsf{d} \text{ at layer } \mathsf{X} \in \mathsf{L}\} : \mathsf{d} \in \mathsf{D_{ce}}\}$$

The function $\mathsf{dsy_{ce-X}}$ maps a program to its possible symbols at a given level.

$$\mathsf{dsy_{ce-X}} : \mathsf{D_{ce}} \rightarrow \mathsf{DSY_{ce-X}}$$

Each complex event symbol has a corresponding state change function. A set $\mathsf{DCG_{ce-X}}$ exists for each layer $\mathsf{X} \in \mathsf{L}$ and each entry in the set contains the state change function for a program in $\mathsf{D_{ce}}$.

$$\mathsf{DCG_{ce-X}} = \{\{\mathsf{f} : \mathsf{f} \text{ is a state change function for program } \mathsf{d}\} : \mathsf{d} \in \mathsf{D_{ce}}\}$$

The function $\mathsf{dcg_{ce-X}}$ maps a program to its state change function.

$$\mathsf{dcg_{ce-X}} : \mathsf{D_{ce}} \rightarrow \mathsf{DCG_{ce-X}}$$

The *program configuration* is defined as the set of programs that existed on the system at a given time. To define the configuration, the $\mathsf{C_{ce}}$ set is defined, which is a power set of $\mathsf{D_{ce}}$ and contains all subsets of the programs. The configuration is defined by the $\mathsf{c_{ce}}$ function, which maps a time to the set of programs that existed at that time.

$$\mathsf{c_{ce}} : \mathsf{T} \rightarrow \mathsf{C_{ce}}$$

Possible Events and State Changes

Based on information about each program and when they were connected, the possible complex events and state changes can be defined. These will be defined based on the program configurations.

To determine the symbols for a program configuration, the $\mathsf{CSY}_{\mathsf{ce-X}}$ set is defined. The set has an entry for every program configuration and the entry contains the set of events that could occur based on the programs that exist.

$$\mathsf{CSY}_{\mathsf{ce-X}} = \{\{\mathsf{s} : \mathsf{s} \in \mathsf{dsy}_{\mathsf{ce-X}}(\mathsf{d}) \text{ and } \mathsf{d} \in \mathsf{C}\} : \mathsf{C} \in \mathsf{C}_{\mathsf{ce}}\}$$

The function $\mathsf{csy}_{\mathsf{ce-X}}$ maps an event device configuration to the set of systemwide events that could occur at level X.

$$\mathsf{csy}_{\mathsf{ce-X}} : \mathsf{C}_{\mathsf{ce}} \rightarrow \mathsf{CSY}_{\mathsf{ce-X}}$$

Similar to the assumption with primitive events, it is assumed that there are no event symbol collisions.

The FSM $\Sigma_{\mathsf{X}}(\mathsf{t})$ function was defined such that it maps a time to the set of possible complex event symbols. This is equivalent to $\mathsf{csy}_{\mathsf{ce-X}}(\mathsf{c}_{\mathsf{ce}}(\mathsf{t}))$, which maps the program configuration to a set of event symbols.

$$\Sigma_{\mathsf{X}}(\mathsf{t}) \equiv \mathsf{csy}_{\mathsf{ce-X}}(\mathsf{c}_{\mathsf{ce}}(\mathsf{t}))$$

To determine the state change function for a program configuration, the $\mathsf{CCG}_{\mathsf{ce-X}}$ sets are defined for each $X \in L$. Each set contains an entry for every configuration and each entry is a state change function for every starting state and complex event symbol supported by the configuration of programs at layer X.

$$\mathsf{CCG}_{\mathsf{ce-X}} = \{\mathsf{f} : \mathsf{f} \text{ is equivalent to all } \mathsf{f}' \in \mathsf{dcg}_{\mathsf{ce-X}}(\mathsf{d}) \text{ and } \mathsf{d} \in \mathsf{C} \text{ and } \mathsf{C} \in \mathsf{C}_{\mathsf{ce}}\}$$

The function $\mathsf{ccg}_{\mathsf{ce-X}}$ maps a program configuration to the corresponding state change function at layer X:

$$\mathsf{ccg}_{\mathsf{ce-X}} : \mathsf{C}_{\mathsf{ce}} \rightarrow \mathsf{CCG}_{\mathsf{ce-X}}$$

The FSM $\delta_{\mathsf{X}}(\mathsf{t})$ function maps a time to the state change function for level X. This is equivalent to $\mathsf{ccg}_{\mathsf{ce-X}}(\mathsf{c}_{\mathsf{ce}}(\mathsf{t}))$, which maps the program configuration to its state change function.

$$\delta_{\mathsf{X}}(\mathsf{t}) \equiv \mathsf{ccg}_{\mathsf{ce-X}}(\mathsf{c}_{\mathsf{ce}}(\mathsf{t}))$$

The "use" and "def" terminology from programming languages are used and the $\mathsf{use_X}$ function is defined as a mapping from an event symbol in $\Sigma_{\mathsf{X}}(\mathsf{t})$ to a set of storage locations that are read from during the event. For this mapping, the set $\mathsf{USE_{cs}}$ is used, which contains all subsets of the systemwide storage locations ($\mathsf{SAM_{cs}}$) that are used by events.

$$\mathsf{use_X} : \Sigma_{\mathsf{X}} \rightarrow \mathsf{USE_{cs}}$$

Similarly, the $\mathsf{def_X}$ function maps the alphabet symbol in $\Sigma_{\mathsf{X}}(\mathsf{t})$ to a set of storage locations that the complex event writes to. For this mapping, the set $\mathsf{DEF_{cs}}$ is used, which contains all subsets of the systemwide storage locations ($\mathsf{SAM_{cs}}$) that are defined by events.

$$\mathsf{def_X} : \Sigma_{\mathsf{X}} \rightarrow \mathsf{DEF_{cs}}$$

Transformation Function Definitions

The model also needs to account for the transformation functions between the layers of complex events. To define the mapping of the transformation functions, the $\mathsf{SSY_{ce-X}}$ set is defined, which contains the event symbols for all programs at layer X.

$$\mathsf{SSY_{ce-X}} = \{\mathsf{s} : \mathsf{s} \in \mathsf{dsy_{ce-X}}(\mathsf{d}) \text{ and } \mathsf{d} \in \mathsf{D_{ce}}\}$$

The set $\mathsf{SEQ_{ce-X}}$ contains all sequences of event symbols at layer X that are mapped to by a higher-level complex event.

The abstraction functions map a sequence of event symbols from $\mathsf{SEQ_{ce-X}}$ to an event symbol in $\mathsf{SSY_{ce-X'}}$. The set $\mathsf{ABS_{ce}}$ contains all of the abstraction functions. The function $\mathsf{abs_{ce}}$ maps a pair of layer names to the corresponding transformation function.

$$\mathsf{abs_{ce}} : \mathsf{L} \times \mathsf{L} \rightarrow \mathsf{ABS_{ce}}$$

The set $\mathsf{MAT_{ce}}$ contains all of the materialization functions. The function $\mathsf{mat_{ce}}$ maps a pair of layer names to the corresponding transformation function.

$$\mathsf{mat_{ce}} : \mathsf{L} \times \mathsf{L} \rightarrow \mathsf{MAT_{ce}}$$

When a primitive event occurred, the change in state was instantaneous, but this is not the case with complex events. The change from each underlying primitive event is still instantaneous, but the complex event itself is not and may last for several time changes. In addition, when computers that can multitask multiple processes are considered, a complex event e_2 from one process can start after complex event e_1 from a different process and e_2 can end before e_1, even if they are at the same level. This can cause problems when reconstructing complex events.

Events can be occurring at all levels and the set $\mathsf{SL_{ce}}$ contains ordered lists of event symbols from each layer of complex- and primitive-level events. Each set is a slice of events that could be occurring at each time. For example, if we consider user-and primitive-level events then the lowest symbols in the ordered list will be for the primitive-level events and the highest symbols are for user-level events.

History Definition

The *complex event history* of a system is represented by a function $\mathsf{h_{ce}}$ that maps the time t to the set $\mathsf{SL_{ce}}$.

$$\mathsf{h_{ce}} : \mathsf{T} \rightarrow \mathsf{SL_{ce}}$$

$\mathsf{e} \in \mathsf{h_{ce}}(\mathsf{t})$ if and only if $\mathsf{e} \in \left\{\bigcup_{\mathsf{X} \in \mathsf{L}} \Sigma_{\mathsf{X}}(\mathsf{t})\right\}$, started before $\mathsf{t} + \Delta\mathsf{t} \in \mathsf{T}$, ended at $\mathsf{t} \in \mathsf{T}$ or later, and e caused the primitive event at t, $\mathsf{h_{pe}}(\mathsf{t})$

2.4.4 Complex History Model Proofs

This section provides proofs with regard to the necessity and sufficiency of the complex history model.

Theorem 2.4.1 *The 17 variable complex history model defined in Equation 2.2 is necessary to represent the complex history of a FSM for the times in T.*

Proof The necessity condition requires that a complex state or event occurred during the times in T only if it is represented in the complex history model. This is proved by contradiction.

If complex event e started before $\mathsf{t} + \Delta\mathsf{t} \in \mathsf{T}$, ended at $\mathsf{t} \in \mathsf{T}$ or later, caused the primitive event at time t, and $\mathsf{e} \notin \mathsf{h_{ce}}(\mathsf{t})$ then either $\mathsf{e} \notin \left\{\bigcup_{\mathsf{X}\in\mathsf{L}} \Sigma_{\mathsf{X}}(\mathsf{t})\right\}$ or multiple event slices started before $\mathsf{t} + \Delta\mathsf{t}$, ended at t or later, and caused the primitive event at time t. If $\mathsf{e} \notin \left\{\bigcup_{\mathsf{X}\in\mathsf{L}} \Sigma_{\mathsf{X}}(\mathsf{t})\right\}$ than the event could not have occurred in the FSM mapped to by the complex event configuration, which means that the event could not have occurred. Because the duration of a complex event is at least as long as the duration of a primitive event, Theorem 2.3.1 shows that only one complex event slice could have occurred during $\Delta\mathsf{t}$ and therefore multiple events could not have occurred. Therefore if $\mathsf{e} \notin \mathsf{h_{ce}}(\mathsf{t})$ then complex event e did not start before $\mathsf{t} + \Delta\mathsf{t}$, end at t or later, and cause the primitive event at time t.

If complex state $\mathsf{s} \in \left\{\bigcup_{\mathsf{X}\in\mathsf{L}} \mathsf{Q}_{\mathsf{X}}(\mathsf{t})\right\}$ existed at time t then there are two situations. If $\mathsf{t} \in \mathsf{T}$ and $\mathsf{h_{cs}}(\mathsf{t}) \neq \mathsf{s}$ then a contradiction exists about which state existed and this is not possible. If $\mathsf{t}' < \mathsf{t} < \mathsf{t}' + \Delta\mathsf{t}$ where $\mathsf{t}' \in \mathsf{T}$ and $\mathsf{t}' + \Delta\mathsf{t} \in \mathsf{T}$ and both $\mathsf{h_{cs}}(\mathsf{t}') \neq \mathsf{s}$ and $\mathsf{h_{cs}}(\mathsf{t}' + \Delta\mathsf{t}) \neq \mathsf{s}$ then Theorem 2.3.2 shows that it is not possible to have a different state between times t' and $\mathsf{t}' + \Delta\mathsf{t}$ and therefore if $\mathsf{h_{cs}}(\mathsf{t}) \neq s$ then state s did not exist at time t. ■

Theorem 2.4.2 *The 17 variable complex history model defined in Equation 2.2 is sufficient to represent the complex history of a FSM for the times in T.*

Proof The sufficiency condition requires that a complex state or event be represented in the model only if it existed or occurred.

Let $\mathsf{e} \in \left\{\bigcup_{\mathsf{X}\in\mathsf{L}} \Sigma_{\mathsf{X}}(\mathsf{t})\right\}$ be a complex event in the complex event history where $\mathsf{e} \in \mathsf{h_{ce}}(\mathsf{t})$, but e did not start before $\mathsf{t} + \Delta\mathsf{t}$, end at t or after, or cause the primitive event at time t. By definition of $\mathsf{h_{ce}}$, this is not possible and e must have occurred if it is in $\mathsf{h_{ce}}$.

Let $s \in \left\{\bigcup_{X \in L} Q_X(t)\right\}$ be a state in the complex state history where $\mathsf{h_{cs}(t) = s}$, but s did not exist at time t. By definition of $\mathsf{h_{cs}}$, this is not possible and s must have existed if it is in $\mathsf{h_{cs}}$. ■

2.5 The Inferred History

The two previous sections defined a model that can theoretically represent the full history of a computer. An investigator will need to determine parts of the computer's history and in some cases a single possible history cannot be determined and multiple histories will be possible. This section defines the inferred history model of a system. This model uses the same set and function names as the ideal history model, but it allows multiple possible histories to exist. The investigator will define the sets and functions in this model during an investigation. The *inferred history* of a system is the sequence of configurations, states, and events that an investigator believes to have existed and occurred between two times. Ideally, there will be one inferred history and it will be equivalent to what actually occured.

The inferred history model has the same sets and functions as the ideal history model except that the names will be in an italic font instead of sans serif. Further, many functions have an additional input set, which is a set of integers that represent the number of inferred histories that are being considered. For example $h_{ps}(t, \mathcal{3})$ represents the inferred state history with identifier 3 for time t while $h_{ps}(t, \mathcal{5})$ represents the inferred state history with identifier 5. The set H contains the history identifiers that have been defined.

Over the course of an investigation, the investigator defines the sets and functions in the inferred history based on his direct observations and supported hypotheses. The distinction will be made more clear in Section 3.2.2, but the concept is that an investigator's direct observation is one that he performs with his senses. He can make a direct observation about the state of a computer screen, but he cannot make a direct observation about data on the hard disk because he cannot sense the values.

Therefore, he must formulate and test hypotheses about the state of the hard disk before the state is added to the inferred history. If the hypothesis is not formally tested then a justification should be made with respect to why testing is not needed.

The sets and functions that exist for the inferred history are now defined. Because the concepts of these sets and functions have already been described and because many sets in both models are simply data from other sets in different tuples and orderings, this section focuses on the sets and functions that the investigator must define.

2.5.1 Inferred Primitive History

The inferred primitive history is defined by the following 11 sets and functions:

$$(T, D_{ps}, DAD_{ps}, ADO_{ps}, c_{ps}, h_{ps}, D_{pe}, DSY_{pe}, DCG_{pe}, c_{pe}, h_{pe}) \tag{2.3}$$

Each of them are now defined.

The T set contains the times that the investigator believes that the system has a history. For an investigation, the investigator may only be concerned with the times that are being investigated.

At the primitive level, one of the first things the investigator must determine is which storage devices were connected to the system during the times in T and what their storage capabilities are. The names of the storage devices are formally defined in the D_{ps} set, the sets of addresses for the storage devices are stored in the DAD_{ps} set, and the domain of each address is stored in the ADO_{ps} set.

The investigator must also make inferences about the times when each device was connected to the system. This is formally done using the c_{ps} function, which maps the time and history identifier to a set of storage devices. Different history identifiers are used when a unique mapping cannot be determined.

$$c_{ps} : T \times H \rightarrow C_{ps}$$

Using the storage capabilities of each device and knowledge about when they were connected, the states that were possible at each time can be determined. The function cst_{ps} maps a set of primitive storage devices to a set of possible states.

$$cst_{ps} : C_{ps} \rightarrow CST_{ps}$$

The Q_{prim} function maps the time and history identifier to the corresponding set of states. $Q_{prim}(t, h)$ is equivalent to $cst_{ps}\left(c_{ps}(t, h)\right)$.

The investigator must also determine which events were possible in the system. This is first done by identifying the event devices that existed during the times in T, which are formally defined in the D_{pe} set. The investigator must also determine the events that each device could cause and the state transition function for each event. These are formally defined in the DSY_{pe} and DCG_{pe} sets.

The investigator must next make inferences about the event devices that existed in the system at each time. The function c_{pe} maps a time and history identifier to the set of event devices that are believed to have existed.

$$c_{pe} : T \times H \rightarrow C_{pe}$$

Based on the mapping of which devices existed at each time, the possible event symbols and transition functions for each time can be defined. The function csy_{pe} maps a set of event devices to the possible event symbols.

$$csy_{pe} : C_{pe} \rightarrow CSY_{pe}$$

The FSM variable $\Sigma_{prim}(t, h)$ is equivalent to $csy_{pe}\left(c_{pe}(t, h)\right)$.

The state transition function at each time can be determined using the ccg_{pe} function, which maps a set of event devices to a state transformation function.

$$ccg_{pe} : C_{pe} \rightarrow CCG_{pe}$$

The FSM variable $\delta_{prim}(t, h)$ is equivalent to $ccg_{pe}\left(c_{pe}(t, h)\right)$.

Lastly, the investigator must make inferences about the states and events that existed or occurred. The inferred state history maps a time and history identifier to

a state that is believed to have existed. If a single state cannot be identified for a time, then a new inferred history can be created with a new history identifier.

$$h_{ps} : T \times H \rightarrow Q_{prim}$$

The inferred event history maps the time and history identifier to the event that is believed to have occurred. New inferred histories can be defined if a single event cannot be determined.

$$h_{pe} : T \times H \rightarrow \Sigma_{prim}$$

2.5.2 Inferred Complex History

The inferred complex history is defined by the following 17 sets and functions:

$$\begin{aligned}(&T, L, D_{cs}, DAD_{cs}, DAT_{cs}, ADO_{cs}, ABS_{cs}, MAT_{cs}, c_{cs-X}, h_{cs},\\ &D_{ce}, DSY_{ce-X}, DCG_{ce-X}, ABS_{ce}, MAT_{ce}, c_{ce}, h_{ce})\end{aligned} \tag{2.4}$$

Each of them are explained in this section.

The set L is defined based on the complex event layers that the investigator believes to have existed. In reality, programs have many layers of abstraction within them, but the investigator will likely not be able to determine them, especially if he does not have source code. Therefore, this set will only contain the layers that the investigator knows to have existed and that are being investigated.

Next, the investigator must make inferences about the complex storage types that existed on the system for the times in T. The names of which are formally defined in the D_{cs} set. For each complex storage type, the investigator may also define a set of addresses in DAD_{cs}. Recall that the addresses are included in the model for convenience and clarity and not necessity. The investigator must also make inferences about the attributes and storage capabilities of each attribute and define the DAT_{cs} and ADO_{cs} sets accordingly.

At each time in T, different complex storage types may have existed. Therefore, the investigator must make inferences about which ones existed and define the mapping between a time and the possible data complex storage types. The function c_{cs-X} maps a time and inferred history identifier to the set of types that existed. This defines the complex storage configuration.

$$c_{cs-X} : T \times H \rightarrow C_{cs}$$

Multiple inferred histories can be defined if a unique mapping cannot be determined.

When the existence of each complex storage type at each time has been inferred, the possible complex storage states can be determined. The function cst_{cs} maps a complex storage configuration to a set of possible complex storage states.

$$cst_{cs} : C_{cs} \rightarrow CST_{cs}$$

This is used to define the $Q_X(t, h)$ FSM function, which maps the time and history identifier to the corresponding set of complex storage states at layer X. $Q_X(t, h)$ is equivalent to $cst_{cs}\left(c_{cs-X}(t, h)\right)$.

The investigator must also make inferences about the abstraction and materialization transformation functions between the layers. These functions must be defined in the ABS_{cs} and MAT_{cs} sets.

To investigate complex events, the investigator must determine which programs existed on the system for the times in T. The names of which are formally defined in the D_{ce} set. Each program can cause different events at different levels of abstraction and the DSY_{ce-X} and DCG_{ce-X} sets must be defined with the inferred event symbols and state change functions for each program at each layer X.

To determine which events could have occurred at each time, the c_{ce} function is defined, which maps a time to the programs that existed.

$$c_{ce} : T \times H \rightarrow C_{ce}$$

If the investigator cannot determine a unique mapping then multiple histories can be defined. The set of programs that exist are the program configuration.

The function csy_{ce-X} maps a program configuration to the possible complex event symbols.

$$csy_{ce-X} : C_{ce} \rightarrow CSY_{ce-X}$$

The FSM variable $\Sigma_X(t,h)$ is equivalent to $csy_{ce-X}\left(c_{ce}(t,h)\right)$.

The state change function at each time is determined using the ccg_{ce-X} function, which maps a program configuration to a state change function.

$$ccg_{ce-X} : CONT_{ce} \rightarrow CCG_{ce-X}$$

The FSM variable $\delta_X(t,h)$ is equivalent to $ccg_{ce-X}\left(c_{ce}(t,h)\right)$.

The investigator may also need to define the transformation functions for both the abstraction and materialization of complex events. The investigator defines these in the ABS_{ce} and MAT_{ce} sets.

The investigator may next need to define the complex storage states and events that may have existed and occurred. The inferred complex state history maps the time and history identifier to a state value. If a single state cannot be determined, then multiple histories can be defined.

$$h_{cs} : T \times H \rightarrow \left\{ cst_{cs}\left(\bigcup_{X \in L} c_{cs-X}(t,h) \right) \right\}$$

The inferred complex event history maps a time to a slice of complex events that were occurring. If a single slice cannot be determined, then multiple histories can be defined.

$$h_{ce} : T \times H \rightarrow SL_{ce}$$

3 SCIENTIFIC DIGITAL INVESTIGATION PROCESS

Chapter 1 defined a digital investigation as a process that formulates and tests hypotheses to answer questions about digital events or the state of digital data. In this work, a specific process model is not proposed to replace the models described in Section 1.2.3. Instead, an investigation is considered to be a process that repeatedly uses the scientific method to formulate and test hypotheses. It can be shown that each phase of existing process models formulates and tests different types of hypotheses. The type of system being investigated and the questions that need to be answered by the investigation will dictate the hypotheses that must be formulated. Section 3.1 gives an overview of the general process and the remaining sections focus on the four phases of the process. Section 3.2 discusses the Observation phase, Section 3.3 discusses the Hypothesis Formulation phase, Section 3.4 discusses the Prediction phase and Section 3.5 discusses the Testing and Searching phase.

3.1 Overview

Investigations are started to answer questions about digital states and events. For example, "does file X exist?", "what events did the attacker cause?", or "did event Y occur?". To answer these questions, the investigator must formulate and test hypotheses about the previous states and events, which is equivalent to defining the sets and functions in the inferred history model.

There are several types of hypotheses that must be formulated. First, hypotheses may be formulated about the storage capabilities of the system. Next, hypotheses may be formulated to define specific states. If the investigation needs to answer questions about complex states, then hypotheses will need to be formulated about the existence of complex storage locations. If the investigation needs to answer

questions about events, then hypotheses will need to be formulated about their occurrence.

In a real investigation, the sets and functions in the model are not formally defined because that would require that a computer be represented by a FSM, but the next chapter will show that the analysis techniques that are used in practice are equivalent to defining various parts of the model. By comparing the theoretical techniques with the practical techniques, the assumptions that are made in practice are made more clear.

As an example of current practice, consider that an investigator has an image file that contains a copy of a system's previous hard disk state. When the copy was made, the storage capabilities of the hard disk had to be determined. This is equivalent to formulating and testing hypotheses about the primitive storage system sets and functions. When the investigator imports the image file into his analysis tool to view the file system, he is defining the sets and functions for primitive and complex states.

In this chapter, the scientific method is used as a process to formulate and test hypotheses. The general process has four phases.

1. **Observation:** Information and resources relevant to the investigation are collected and observed.
2. **Hypothesis Formulation:** Based on the observations, hypotheses are formulated about the system. Different levels of hypotheses will be formulated over the course of the investigation.
3. **Prediction:** To support or refute a hypothesis, predictions about what evidence will exist are made.
4. **Testing and Searching:** Based on the evidence predictions, tests and searches are conducted.

These phases can be seen in Figure 3.1. Some of these phases can be performed by an investigator and others will be performed by analysis tools. For example,

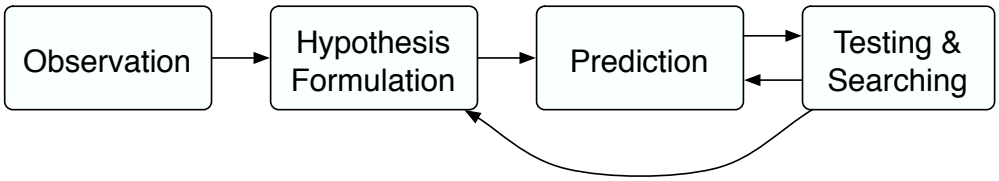

Figure 3.1. Hypotheses can be formulated and tested using the scientific method, which has four phases.

observations can be made by a tool alone or a person can make an observation with the help of a tool. After a hypothesis has been sufficiently tested and not refuted, then the relevant data can be added to the inferred history. Each of these phases is discussed in more detail in the following sections.

3.2 Observation Phase

In the *Observation* phase, an investigator, or program, makes observations about states and events for the purpose of formulating a hypothesis. Sources of observations include data defined in the inferred history and output from analysis tools. Some examples are given here:

- The list of running processes is observed using the `ps` tool.
- The list of files in a directory is observed using a specialized investigation tool.
- The contents of an e-mail are observed in an e-mail client.

This phase is equivalent to an investigator looking at a physical crime scene. In a digital crime scene, the investigator must rely on hardware and software to observe data and the next section discusses different types of observations and the second section discusses a graphical representation of observation zones. The third section discusses the topic of preserving the observed data.

3.2.1 Direct and Indirect Observations

In both the physical and digital world, there are different types of observations. A *direct observation* occurs when a component is aware of something based on its senses (i.e. it is the observer). An *indirect observation* occurs when a component is aware of something based on the observations of other components. A *component* can be software, hardware, or a person and senses for hardware or software include any form of data input.

For example, an investigator can directly observe the state of a monitor because he can see it, but he cannot directly observe the digital state of a hard disk sector. When a program displays the contents of a hard disk to the screen, the investigator is making a direct observation of the monitor and an indirect observation of the hard disk sector. An indirect observation would also occur if someone told the investigator about the contents of the sector.

In general, the investigator trusts direct observations over indirect observations because he trusts his senses more than other components or people. *Trust* is a belief in the accuracy and reliability of a component. Because indirectly observed data are not fact, the accuracy of the observed data should be tested when the data are used to formulate hypotheses. The amount of testing will depend on how much trust has been placed in each component. If the software and hardware being used to indirectly observe the state of a hard disk sector have reliably produced accurate data in the past then the investigator may not test each observation. If the software is new and has not been reliably used or if it is from an untrusted system then the investigator will be more likely to test the observation.

In current systems, all important observations that an investigator makes of digital states are indirect because the state of output controllers are not frequently of direct relevance to him. This means that he must evaluate the accuracy of nearly every observation. Consider if he uses an automated analysis program that formulates and tests hypotheses about various states and events. The program stores the data

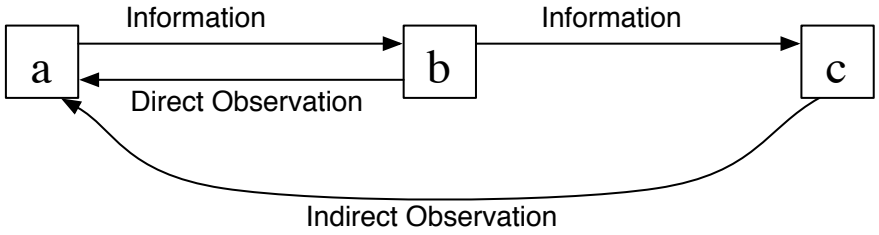

Figure 3.2. A component can directly observe the components that are incident to it. Node b can directly observe node a and node c can indirectly observe node a.

that have been defined in the inferred history and displays the results. Examples of such programs include ones that automatically abstract primitive data into complex storage locations or programs that automatically search for specific types of data. Because the investigator will be making an indirect observation of the program output, he must evaluate the output accuracy to determine if the output he is observing is the same as the actual program output.

3.2.2 Observation Zones

An observation can be graphically represented using an information flow graph $G = (V, E)$ where V is a set of vertices E is a set of directed edges. Each component is a vertex and an edge exists from vertex a to vertex b if information flows from a to b.

Component b can directly observe component a if a line exists from a to b. Component c can indirectly observe component a if a path exists from a to c and c cannot directly observe a. This can be seen in Figure 3.2.

An example information flow graph can be seen in Figure 3.3. The vertex for the investigator is on the top he can directly observe the monitor using his sight and the speakers using his hearing. The investigator can indirectly observe the state of the

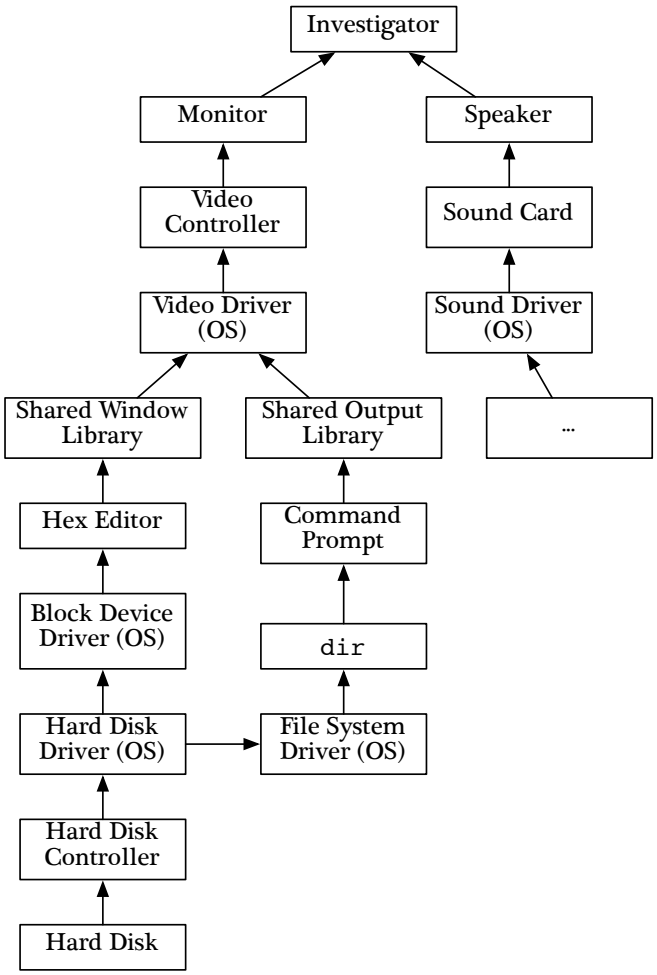

Figure 3.3. The information flow between components can show which observations are direct and indirect. In this case, there are two methods for the investigator to indirectly observe the state of the hard disk.

hard disk using either a hex editor or the `dir` program. A real system would have many more vertices and edges.

An observation can be represented by an *observation tree*, which is subtree of the graph and the root of the tree is the vertex for the observer. The vertices for all components that are used during the observation are included in the tree, along with the edges between the components. Because of system events, the tree could change every Δt time step based on which components were being displayed.

An example of an observation tree can be seen in Figure 3.4.A, which shows the tree from Figure 3.3 where the investigator is the observer and he is viewing the monitor, which is showing the contents of a hard disk sector. The hard disk sector contents are processed by the controller, drivers, and libraries.

To determine if the observed data can be trusted, the trust of each component is considered. Untrusted components could have faults or could be malicious and intentionally hiding or inserting data.

For each observation, the trusted observation zone can be defined. The *trusted observation zone* contains the vertices from an observation tree that have a path to the observer and no vertex in the path is untrusted. If the component being indirectly observed is not in the trusted zone, then the observation is of untrusted data and should be tested before it is used to formulate hypotheses.

Figure 3.4.B shows that the OS and application are not trusted in the observation from Figure 3.4.A. Therefore, the trusted observation zone contains only the investigator, the monitor, and the video controller. This scenario could happen when an investigator is analyzing a computer as it is running, instead of moving the disk to a system where all of the software is trusted. The indirect observation about the state of the hard disk uses untrusted components and is therefore untrusted. Figure 3.4.C shows the same observation, but this time the OS and applications are trusted and the indirect observation of the hard disk is trusted.

3.2.3 State Preservation

When an investigation starts, it is common to preserve the output of an observation tool. *Preservation* is defined as the process of making a copy of a primitive or complex state. When conducting a digital *forensic* investigation, which is an investigation where the resulting evidence is court admissible, the legal requirements may require preservation. Examples of preservation include:

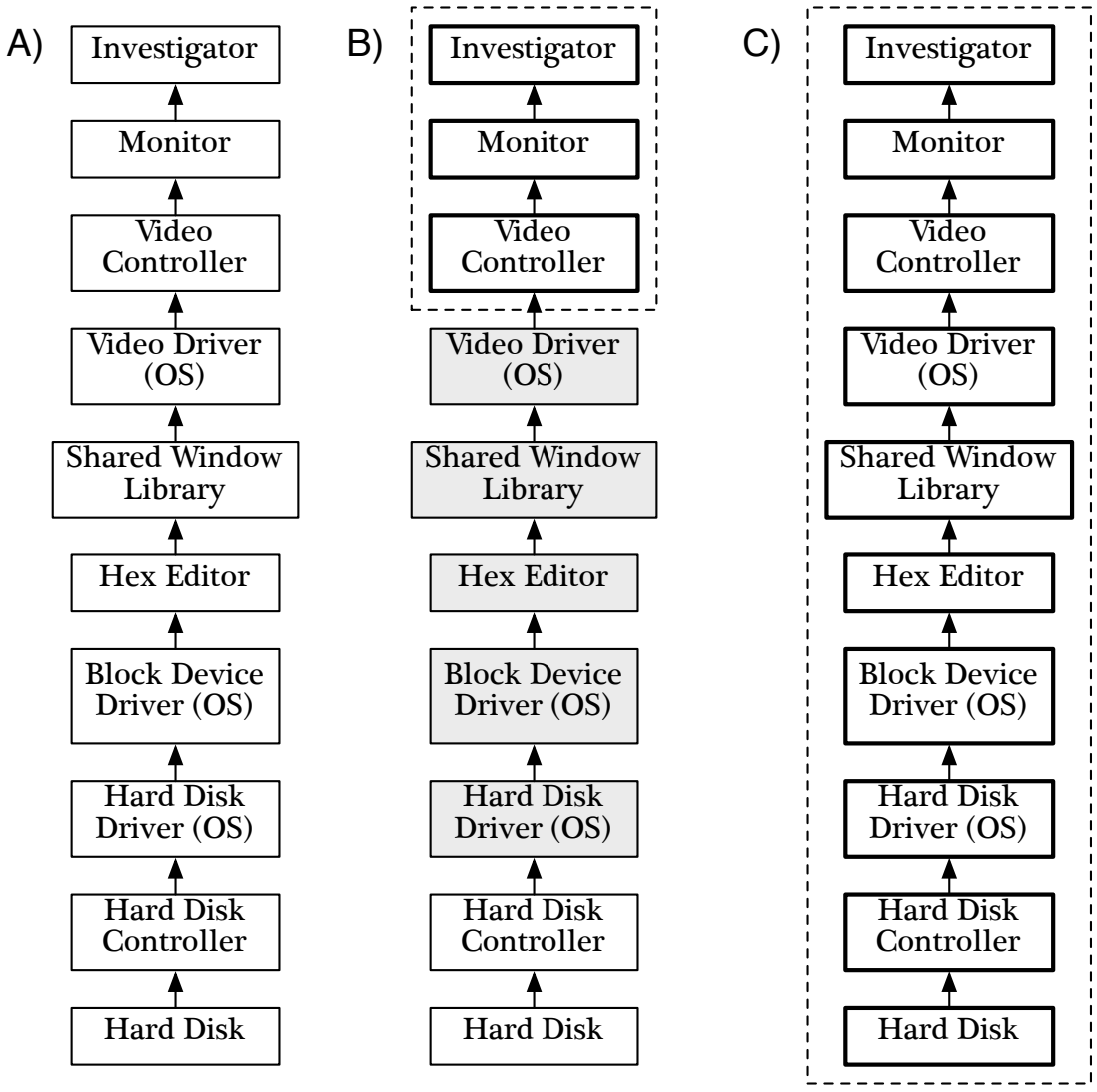

Figure 3.4. A) An observation tree contains the components that are being used to make an observation. This shows a hex editor being used to observe the contents of a hard disk sector. B) A trusted observation zone contains the vertices that have a path to the investigator and do not have untrusted components in the path. The untrusted components are shaded and the trusted observation zone includes the bold components in the dashed lines. C) This observation uses components that are all trusted.

- Copying each sector of a hard disk or each byte of memory to a file (frequently called an image) on another disk
- Copying the output of tools that list running processes and open network ports
- Copying network packets.

The copying process consists of a series of events that read data from the suspect system and write the data to another location. The resulting copy has its own history and when the investigator later wants to use the copy as though it were the state of the suspect system, he will need to be able to show that the state of the copy has not changed since it was made. This is frequently done by calculating the one-way hash when the copy is made and writing it in a notebook. The investigator will then need to show that the copying events were accurate and that the copy is equivalent to the system state.

3.3 Hypothesis Formulation Phase

The *Hypothesis Formulation* phase is where the investigator or program interprets the data observed in the previous phase and formulates hypotheses. In the formal approach, the hypotheses are about the variables in the inferred history. Examples include:

- Storage capabilities of a system.
- Existence of the primitive state of a hard disk.
- Occurrence of a complex event.

Categories of hypothesis types and how to test them are discussed in Chapter 4.

Hypotheses that define variables in the model must be formulated in a specific order. For example, before hypotheses are made about the contents of a file, hypotheses about the existence of the file must be formulated and tested.

In practice, because little logging exists and little is known about the details of programs, hypotheses are not always about specific events and specific times. For

example, based on the observation of a file and the programs that are installed on the system, a possible hypothesis is that "either program X or Y was used to download the file from the Internet." The investigator formulated this hypothesis based on knowledge that both programs are capable of downloading files from the Internet, but he has not enumerated all possible events for each program. Another general hypothesis is that the system is behaving strangely because it was compromised. The tests for this hypothesis will require additional hypotheses about specific types of attack events.

To be a scientific process, the hypotheses must be capable of being refuted. If a hypothesis is supported and not refuted, then the relevant data are added to the inferred history. If a hypothesis is neither supported or refuted, than an assumption can be made that it is true, but the investigator must be capable of justifying the assumption.

If a hypothesis is refuted based on data in the inferred history, then it does not mean that events and states in the hypothesis did not occur. It means only that they did not occur in that inferred history, but that inferred history may not be correct.

In theory, hypotheses could be formulated and tested for the occurrence of every known event at each time and every program on the system could be analyzed to determine which complex events could occur. In practice, that would be impossible and instead hypotheses and predictions are frequently made based on a combination of system and incident characteristics.

System characteristics are properties of hardware and software that make some states events more common for some systems than others. These characteristics allow us to formulate hypotheses based on only the type of software and hardware being investigated. Frequently, these hypotheses are based on the assumption that the hardware and software have not been modified to make them operate differently from similar hardware and software. For example, based on the type of OS, hypotheses about the file system types can be formulated.

Incident characteristics are the general properties of a crime or incident and are system independent. These characteristics may allow the investigator to conduct searches for specific types of evidence using only knowledge about the type of incident. Consider an investigation where a computer is suspected of being used to download contraband pictures. Using incident characteristics, hypotheses can be formulated that a web browser was used to download the files. Next, the system characteristics for the web browsers that are installed are used to predict where evidence may exist. Other examples of incident characteristics are keywords and hash databases. The one-way hash of files that are associated with a type of incident can be calculated, saved, and searched for in subsequent investigations (for example HashKeeper [Uni05a]).

3.4 Prediction Phase

Each hypothesis must be tested and the *Prediction* phase identifies evidence that, if it exists, would support or refute a hypothesis. Based on the predictions, experiments will be conducted in the next phase. The system and incident characteristics described in the previous section also apply to the prediction making process. Some common types of predictions are based on:

- The contents of a file in the inferred history.
- The results from running a simulation.
- The output of executing a program on the suspect system.

For example, if the investigator's hypothesis is that the partition contains an NTFS file system then his prediction would be that valid attribute values will exist after applying the abstraction transformation functions for the NTFS boot sector. If the investigator's hypothesis is that the user downloaded contraband images, then he will predict that the system has JPEG files. For a scientific process, predictions must be made that, if true, would refute the hypothesis.

If the hypothesis being tested is going to define a state of the system in the inferred history based on an indirect observation, then a prediction should be made to test the accuracy of the observed data. Consider if the investigator indirectly observed the state of a hard disk sector using a program and he formulates a hypothesis that the observed state is equal to the actual state. A prediction can be made that an independent program would show the same data.

3.4.1 Searching the Inferred History

If a prediction is about a specific state or event occurring, then the inferred history will need to be searched. We must therefore consider what types of searches can be conducted. For example, can we search for "all reliable files?"

To formally determine the types of inferred history predictions, the predictions are converted into questions. For example, if the prediction is that "file X will exist" then the question "does file X exist?" is used. The questions are then represented using domain relational calculus, which is typically used to describe queries of relational databases. The events and states in the inferred history model have a relational design.

The basics of domain relational calculus are given here and a more detailed overview can be found in database books [SKS02]. A domain relational calculus expression has the following form:

$$\{< x_1, x_2, \ldots, x_n > |\ P(x_1, x_2, \ldots, x_n)\}$$

where $x_1, x_2, \ldots, x_n$ are domain variables and P is a formula composed of atoms. An atom can be one of the following:

- $< x_1, x_2, \ldots, x_n > \in r$ where r is a relation with n attributes and $x_1, x_2, \ldots, x_n$ are variables.
- $x\ \Theta\ y$ where x is a variable, y is a variable or constant, and Θ is a comparison operator, such as $<$, $=$, or $\geq$.

Formulas are made up of atoms and can be joined with an OR ($\vee$) or AND ($\wedge$) operations. Formulas also use the *there exists* ($\exists$) and *for all* ($\forall$) operators.

To support realistic searches during a digital investigation, the comparison operators must be expanded to include subsets, regular expressions, and transformation functions. For example, to compare the MD5 hash of a file with a specific value.

Any search that compares values in the sets and functions of the inferred history with a value can be represented with domain relational calculus. For example, we could search for:

- The value that was in primitive storage location a of device d at time t
- The file complex storage location that had the value v at time t
- The value that complex event e wrote to location a on device d

Consider a search for the time when CPU event `STO R13` wrote the value 4 to register `R13`. This would be represented as:

$$\Big\{ < t > | \exists h \Big(h \in H \wedge \Big(t \in T \wedge \Big(\exists e \big(< e > \in h_{pe}(t, h) \wedge e = STO\ R13 \wedge \big(\exists d1, a1, v1 (< d1, a1, v1 > \in h_{ps}(t + \Delta t, h) \wedge d1 = CPU \wedge a1 = R13 \wedge v1 = 4) \big) \big) \Big) \Big) \Big) \Big\}$$

The above query examines all histories in H and all times in T. All events in all inferred histories are then examined to identify the times in which a `STO R13` event occurred. It then uses the inferred state history at the next time step to determine if register `R13` has the value 4.

It is important to consider what types of searches that can be conducted because the supported questions are not always the questions that are asked. For example, we can not search for "a log file that has reliable contents" unless the definition of a log file is more specific and a metric can be applied to each file to determine if the contents are reliable.

3.5 Testing and Searching Phase

The last phase of the process is where the predictions are tested. Tests may involve running a simulation on another system, querying the hardware for data, executing programs on a running system, or searching the inferred history. Based on the test results, new predictions may be made and hypotheses may be revised.

Example searches of the inferred history that may be conducted include:

- Files whose SHA-1 hash is not in a hash database
- Files whose attributes are set to hidden
- Sectors with a keyword or pattern in them

If the test supports the hypothesis then the investigator, or automated analysis program, can choose to define the relevant functions and sets in the inferred history. He may also choose to conduct more tests and obtain more support before defining the sets and functions.

If the test refutes a hypothesis then the data used in the test will dictate what actions the investigator can perform next. If the test relied on data from the inferred history, then the refuted hypothesis cannot be used to define sets and functions in *that* inferred history. If the data used to refute the hypothesis was defined based on a direct observation, such as the state of the video card, then the hypothesis that was tested must be revised or no longer considered because it conflicts with a direct observation, which the investigator will likely have a high amount of trust in.

If the data used to refute the hypothesis was defined based on another hypothesis, then the investigator can choose to define a new inferred history. We refer to the original inferred history as $h1$ and the new inferred history as $h2$. $h2$ will contain the same data as $h1$ except that it will not contain the data d that refuted the hypothesis that was tested. Additional data d' may also need to be removed from $h2$ if the hypothesis for d' was supported by the conflicting data d, which no longer exists in that inferred history. d' should not exist in the new inferred history $h2$ if $h2$ does not contain the necessary data to support it.

For example, consider if the inferred history $h1$ contains a file f that was defined using a hypothesis and deleted file recovery techniques. If the existence of that file refutes another hypothesis then a new history $h2$ can be defined and the recovered file will not exist in it. If other files and events were previously defined in $h1$ and supported because of the existence of the recovered file f, then they too will need to be removed from history $h2$. Later analysis may reveal that one or both of these inferred histories did not occur.

If a hypothesis is refuted based on data that is not in the inferred history, then the reliability and accuracy of the test data should also be considered before refuting the hypothesis. For example, if a tool is executed on the suspect system and the output conflicts with the hypothesis than the suspect system could have a rootkit or other malicious software that will produce incorrect data. The tool may also be faulty and produce inaccurate data.

In practice, tools are used extensively during this phase because they allow the investigator to search the inferred history for evidence. Some searches are fully automated where the investigator enters what he is looking for and the tool searches for it. Other searches require the investigator to find evidence by visually comparing data on the screen with what he is expecting. An example of the former is keyword searching and an example of the latter is using a directory listing to find a specific file. The visual comparison process is prone to errors because the investigator may misread data or become distracted by other data or other physical events around her, but it is not always easy to fully represent a search in a digital form.

4 CATEGORIES AND CLASSES OF ANALYSIS TECHNIQUES

The previous chapter outlined a general four phase process that can be used to formulate and test hypotheses about the system capabilities, states, and events. This chapter focuses on the techniques that an investigator can use during the Hypothesis Formulation and Prediction phases to define the 11 sets and functions in the inferred primitive history model (Equation 2.3) and the 17 sets and functions in the inferred complex history model (Equation 2.4).

Seven categories of analysis techniques are defined and each focuses on specific sets and functions in the inferred history models. It is shown that the categories completely define all sets and functions. Based on current practice and the design of the history model, the techniques in each category are organized into 31 unique classes. Formal completeness is not shown for the classes of techniques, but validation is provided by showing that the techniques described in the literature are supported by the classes. Section 5.2 will compare existing tools with these classes.

The seven categories are shown in Figure 4.1. The line in between the boxes represents a dependence. A line from box A to box B means that the analysis techniques associated with box B cannot be performed until analysis techniques from A are performed to define specific variables. This dependence will be addressed in more detail in Section 4.4.

The remaining sections describe the seven categories, proofs, examples, and related work. Section 4.1 describes the four categories that can be used to define the inferred primitive history of the system and Section 4.2 describes the three categories for the inferred complex history. Section 4.3 shows that the seven categories are complete and Section 4.4 describes the dependencies between the classes of analysis techniques. Section 4.5 gives examples of how the techniques would be used and Section 4.6 describes related work that applies to the analysis techniques.

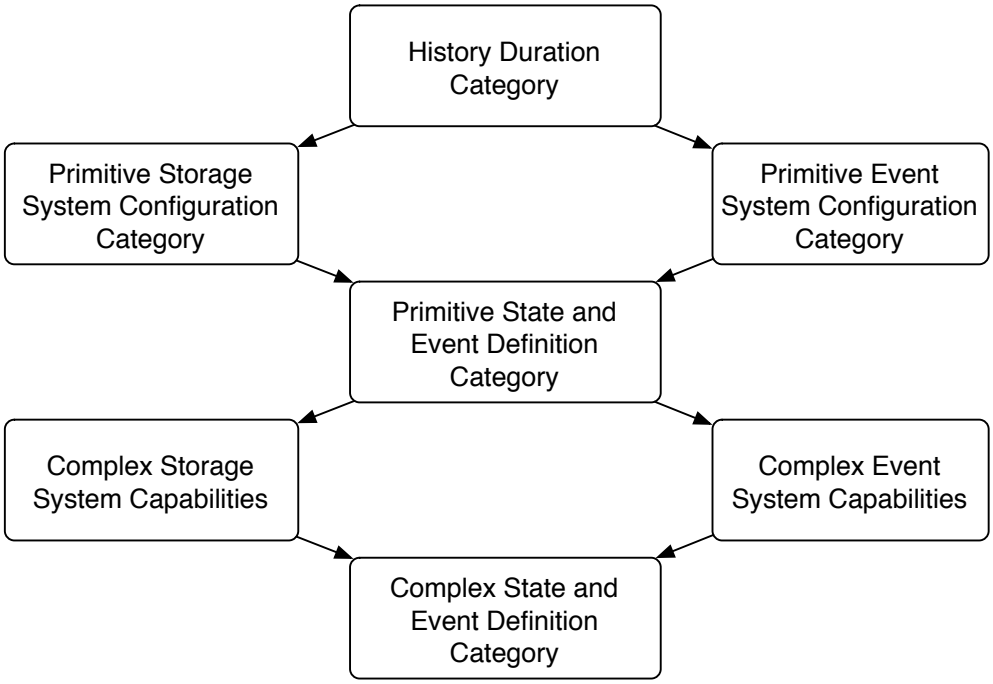

Figure 4.1. The seven categories of analysis techniques are shown here with directed lines representing the dependence between the analysis techniques.

4.1 Primitive History Categories

This section describes four categories of analysis techniques that can define the inferred primitive history. The first category defines the set T, which contains the times that the system has a history. The next two categories are used to define the primitive storage and event capabilities. The fourth category is used to define the states and events that are believed to have occurred.

4.1.1 History Duration Category

The *History Duration* Category contains one analysis technique and it can be used to formulate and test hypotheses about the contents of set T, which contains the times that the system being investigated has a history. The set T is defined for

both the primitive and complex history models. Recall that the duration Δt between the times in T is based on the fastest state change.

In an investigation, T needs to contain only the times that are relevant to the investigation and the investigator needs to show that the system existed during those times. As the investigation progresses, the timeframe of interest may expand and new hypotheses about the times in T may need to be formulated and tested.

The method for formulating and testing hypotheses about the duration of the history is *operational reconstruction*, which uses event reconstruction and temporal data from the storage devices to determine when events occurred. The only evidence of an event occurring is the state that was changed and this technique tries to identify the state changes that have a time associated with them.

During the Observation phase, the investigator recognizes temporal data and formulates the hypothesis that the system was operational during the observed times and therefore should be in the set T. Examples of temporal data include file times, dates in e-mail headers, and log messages. Note that all information on a storage device can be overwritten and therefore evidence of previous events can be erased and not observed.

When this technique is used, the Prediction and Testing phases of the general process should test if the system clock is reliable and if the observed temporal data is from activity on this system. For example, are the file creation times earlier than the manufacture date of the system?

In practice, an investigator should verify that the system existed during the timeframe being investigated. Each Δt time step will probably not be defined or tested because the granularity is too small.

4.1.2 Primitive Storage System Configuration Category

The *Primitive Storage System Configuration* Category contains analysis techniques to define the primitive storage capabilities. The techniques in this category will define the following sets and functions in the inferred primitive history definition:

- D_{ps}: The names of the storage devices connected during the times in T.
- DAD_{ps}: The number of addresses for each storage device in D_{ps}.
- ADO_{ps}: The domain of each address on each storage device in D_{ps}.
- c_{ps}: The function that maps a time to the set of storage devices that were connected.

These sets and functions are used to define the Q_{prim} FSM set.

There are two types of hypotheses that are formulated in this category. The first defines the D_{ps}, DAD_{ps}, and ADO_{ps} sets, which represent the name and storage capabilities of each device. The second type of hypothesis defines the c_{ps} function, which represents the devices that were connected at each time. Figure 4.2 shows a table where the column headings are the types of hypotheses and the rows include the classes of analysis techniques that can be used to formulate and test the hypotheses. The set and function names that are defined in the inferred history model are also given. The following two sections describe the classes of techniques associated with these two types of hypotheses.

Storage Device Capability Hypotheses

We first consider the device name, address, and domain hypotheses. There are two classes of techniques that can be used to formulate and test these hypotheses. The first makes direct observations about what others have stated about the capabilities and the second makes observations based on the results of queries to the device. When one is used to formulate a hypothesis, the other can be used to test it.

Primitive Storage System Configuration Category

	Storage Device Capability Hypotheses	**Storage Device Connection Time Hypotheses**
Classes of Analysis Techniques	- Storage Device Capability Observation - Storage Device Capability Query	- Storage Device Connection Observation - Storage Device Connection Reconstruction
Model Variables Defined	ADO_{ps}, D_{ps}, DAD_{ps}	c_{ps}

Figure 4.2. The Primitive Storage System Configuration Category of analysis techniques can formulate and test two types of hypotheses and includes four classes of analysis techniques.

The first class of techniques is *Storage Device Capability Observation* and it uses direct observations of the storage device to determine its storage capabilities. Examples include reading the device labels for the model number, size, and technical specifications. This class of techniques makes a direct observation of the label or specification, but not of the actual storage locations. Presumably, the investigator trusts what he sees and the observation of the technical data will be trusted. But, the investigator must formulate a hypothesis that the actual storage system capabilities are equal to the capabilities on the label or technical manual. This would reveal if the label or model numbers on the device were incorrect. This class of techniques requires the device type information and specifications to be correctly observed and processed.

The second class of techniques is *Storage Device Capability Query*. In this case, the investigator uses a program to query the hardware device for information. Hypotheses are formulated based on the program output and the trustworthiness of the output is based on the trust associated with the software and hardware used. If untrusted hardware or software is used, then testing should be performed to confirm the results. The details of the querying methods will be storage device dependent,

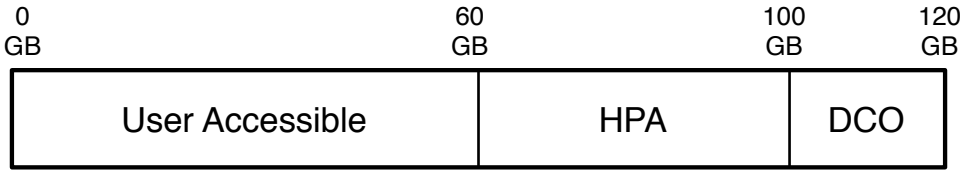

Figure 4.3. There are at least three different ATA commands to determine the number of addresses that a drive has and all three could return different values.

but examples include executing a command to return device capability information or exhaustively trying all addresses and storage values until an error occurs.

Different querying methods may provide different results. For example, an ATA hard disk could report three different sizes from different ATA commands if a host protected area (HPA) and device configuration overlay (DCO) exist [Car05b]. HPAs can be used by a vendor to hide recovery files and a DCO can be used by the hard disk manufacturer to change the apparent disk size. Figure 4.3 shows a 120 GB disk that has 20 GB at the end allocated to a DCO and 40 GB allocated to an HPA. There are only 60 GB of the disk that a user can access and that would be accessible by trying to read every sector.

The storage device querying class of techniques must be able to determine the storage capabilities using methods that are appropriate for the device and accurately present the data to the investigator or other analysis tool. When multiple querying methods exist, the tools should identify which techniques were used so that the investigator can act accordingly. In practice, both of these techniques are used. Many hard disk vendors print the disk size on the labels and the vendor websites can be used to determine the size based on the model number.

Storage Device Connection Time Hypotheses

The second type of hypothesis in the primitive storage device category defines when each primitive storage device was connected to the system. There are two classes of techniques that this section considers. The first uses direct observations to determine what could have been connected and the second uses evidence from events caused by the removal or insertion of the devices.

The *Storage Device Connection Observation* class of techniques formulates and tests hypotheses based on direct observations of the storage devices. The investigator can formulate hypotheses about the devices that are found to be connected to the system when the investigation starts. These hypotheses can be tested by verifying that the computer was setup to support the device. The investigator can also formulate hypotheses about removable media that are found around the computer, but specific times will not be known without additional reconstruction analysis.

The *Storage Device Connection Reconstruction* class of techniques makes observations about evidence from the events associated with the insertion and removal of a storage device. If it exists, the evidence to support the reconstruction would be found in a log. The storage device may have a log, the file system may record the last mount time, or the OS may record the devices that existed at boot time. To test these hypotheses, the reliability of the log will come into question because it could have been modified to remove references to a specific device. The reconstruction can also use general temporal data on the storage device.

In could be possible for there to be multiple hypotheses about connection times that cannot be refuted. If so, then a new inferred history can be defined for each hypothesis about c_{ps}. The requirements for the connection reconstruction class of techniques are that they must understand the effects of the device insertion and removal process from the perspective of both the storage device and system so that evidence can be found in multiple locations.

In practice, these classes of techniques are not formally performed for all investigations. They may be considered when it is not clear if the drive that was found in a system is the original drive. They may also be considered for removable media in cases where documents and other intellectual property have been stolen and the investigator needs to track down when the media was connected.

4.1.3 Primitive Event System Configuration Category

The *Primitive Event System Configuration* Category defines the capabilities of the primitive event system, whereas the previous category defined the capabilities of the primitive storage system. The techniques in this category define the following sets and functions in the inferred primitive history:

- D_{pe}: The names of the event devices connected during the times in T.
- DSY_{pe}: The event symbols for each event device in D_{pe}.
- DCG_{pe}: The state change function for each event device in D_{pe}.
- c_{pe}: The function that maps a time to the set of event devices in D_{pe} that were connected during the times in T.

These sets and functions are used to define the FSM Σ_{prim} set and δ_{prim} function.

Examples of primitive event devices include central processor units (CPU), memory management units, and the processors on video cards, network cards, and hard disk controllers. In practice, because questions are not asked about the states or events of these devices, their capabilities are not considered during an investigation.

When the capabilities of the primitive event devices need to be defined, there are two general types of hypotheses that must be considered. One defines the D_{pe}, DSY_{pe}, and DCG_{pe} sets, which represent the name, event symbols, and state change function for each event device. The second defines the c_{pe} function, which represents the times that each event device was connected to the system. Figure 4.4 shows a table where the column headings are the types of hypotheses and the rows include

Primitive Event System Configuration Category

	Event Device Capability Hypotheses	**Event Device Connection Time Hypotheses**
Classes of Analysis Techniques	- Event Device Capability Observation - Event Device Capability Query	- Event Device Connection Observation - Event Device Connection Reconstruction
Model Variables Defined	D_{pe}, DSY_{pe}, DCG_{pe}	c_{pe}

Figure 4.4. The Primitive Event System Configuration Category of analysis techniques can formulate and test two types of hypotheses and includes four classes of analysis techniques.

the classes of analysis techniques and the set and function names that are defined. The following two sections describe the classes of techniques associated with these two types of hypotheses.

Event Device Capability Hypotheses

We first consider hypotheses about the capabilities of the event devices. Similar to storage devices, there are two classes of techniques for formulating and testing hypotheses about event device capabilities. The first uses direct observations about the devices and the second queries the device.

The first class of techniques is *Event Device Capability Observation*, where direct observations are made during the Observation phase about the make and model of event devices that could have been connected to the system. To determine the symbols and state change functions for each, a developer's manual or specification can be used. Based on these direct observations of the chip's exterior or manual, a hypothesis can be formulated about the device's capabilities. Testing this hypothesis may reveal if the model number is incorrect, if there are undocumented instructions, or if the logic for an instruction has changed. The requirements for this class of

techniques is that they can correctly observe and process the device type information and specifications.

The second class of techniques is *Event Device Capability Query*, where a program is used during the Observation phase to query the event device and determine its capabilities. The details of the querying techniques will depend on the device. One example is that the device could return a list of instructions that it supports. Or, the investigator could exhaustively try all possible opcodes until an error is generated. Similarly, each event and starting state could be simulated to determine the state change function.

The event device querying class of techniques must be able to determine the event capabilities using methods that are appropriate for the device. Most event devices do not provide a list of supported instructions, but if multiple methods exist to query the device, then the tools should identify which techniques were used so that the investigator can act accordingly. In practice, the primitive event capabilities of a system are not defined during the investigation because the investigation questions are not about primitive events.

Event Device Connection Time Hypotheses

We next consider hypotheses about when an event device was connected to the system. There are two classes of techniques for this type of hypothesis. The first uses direct observations and the second uses evidence from the device's existence.

The first class of techniques is *Event Device Connection Observation* and it formulates and tests hypotheses based on direct observations of the event devices. Hypotheses can be formulated about the devices that are connected to the system when the investigation starts. These hypotheses can be tested by verifying that the computer was setup to support the device. Hypotheses can also be formulated about devices that were found around the system, but specific times cannot be determined without additional analysis.

The second class of techniques is *Event Device Connection Reconstruction*, where the events associated with the insertion and removal of the event devices are reconstructed. In many cases, the event devices that are of interest to an investigator are not frequently removed and added and logs are not kept.

The requirements for the connection reconstruction class of techniques are that they must understand the effects of the device insertion and removal process from the perspective of both the event device and system. This is so that, when possible, evidence can be found in multiple locations.

4.1.4 Primitive State and Event Definition Category

The final category for the primitive history is the *Primitive State and Event Definition* Category, which can be used to define the states and events that are believed to have occurred. The primitive state includes the state of hard disk sectors, registers, and bytes of memory and primitive events include the machine-level instructions. In the formal model, this category defines the following functions:

- h_{ps}: The function that maps a time and history identifier to a primitive state.
- h_{pe}: The function that maps a time and history identifier to a primitive event.

There is only one type of hypothesis that is considered in this category and it specifies a state or event (or both). There are five classes of analysis techniques that can be used to formulate and test this type of hypothesis and they are listed in the table in Figure 4.5, which shows the classes of techniques and the model variables that are defined.

The five classes of analysis techniques have directional components to them, as can be seen in Figure 4.6. The observation class of techniques can be used to define a single state of an output device that was directly observed. The system capabilities and sample data classes of techniques can be used to formulate and test hypotheses about individual states and events. The reconstruction and construction classes of techniques are based on the concept that the primitive state at time t is the ending

Primitive State and Event Definition Category

	State and Event Occurrence Definition
Classes of Analysis Techniques	- Primitive State Observation - Primitive State and Event System Capabilities - Primitive State and Event Sample Data - Primitive State and Event Reconstruction - Primitive State and Event Construction
Model Variables Defined	h_{pe}, h_{ps}

Figure 4.5. The Primitive State and Event Definition Category of analysis techniques can formulate and test one type of hypothesis and includes five classes of analysis techniques.

state of the event that occurred at time $t - \Delta t$ with a specific starting state. If the starting state, event, and ending state are in the inferred history, then we have the following:

$$h_{ps}(t) = \delta_{prim}\,(t - \Delta t)\,(h_{ps}(t - \Delta t), h_{pe}(t - \Delta t))$$

Reconstruction goes backwards and uses the ending state of an event ($h_{ps}(t)$) to make inferences about the previous event and starting state while construction goes forwards and uses the starting state of an event ($h_{ps}(t - \Delta t)$) to make inferences about an event and ending state.

Because all five classes of techniques are different approaches to defining the same two functions, a hypothesis can be formulated using a technique from one class and tested with techniques from other classes. Section 4.2.3 will show that primitive states and events can also be tested by materializing complex states and events. The five classes of analysis techniques to define states and events are described in the following subsections.

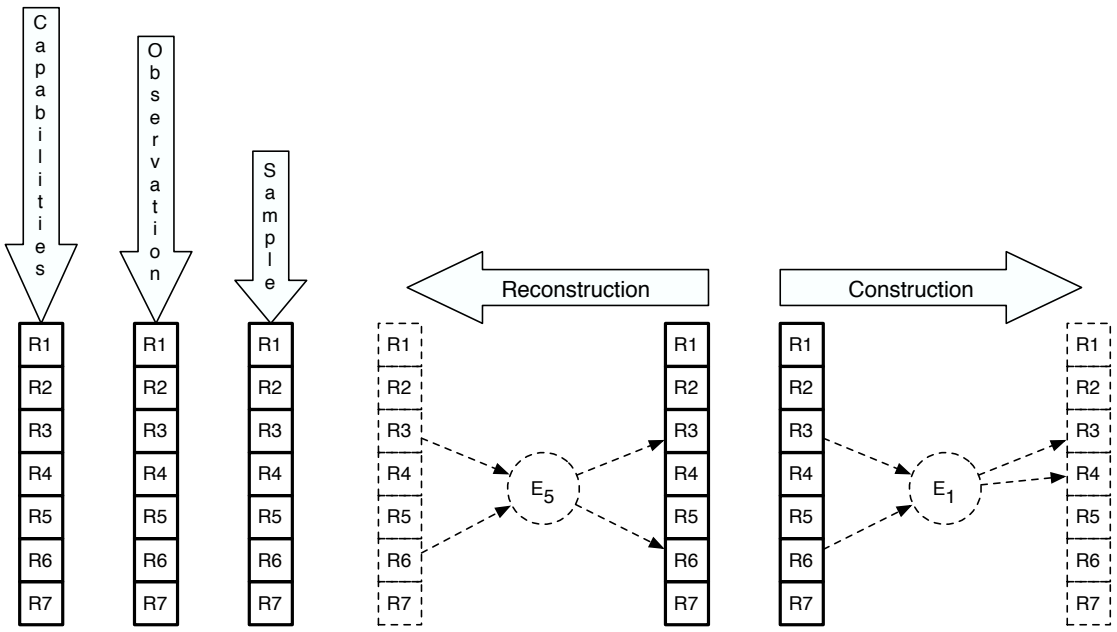

Figure 4.6. The five classes of analysis techniques for defining primitive states and events have directional components to them.

State Observation Class

The *Primitive State Observation* class of techniques uses a direct observation of the system to define a state in the inferred history. This technique is unique because it involves a direct observation of an output device and it is assumed that the investigator trusts his senses. Therefore, the observation is fact and it need not be tested unless it conflicts with a direct observation from another investigator. The observed state is defined in the inferred history for the times that the observation was made. Because the size of Δt is small, assumptions may be made about the start and ending times.

In practice, the investigator is not interested in the state of an output device controller and instead he is interested in the state of other system data that is encoded in the device controller state. To define the state of the system data, event

reconstruction techniques will be needed, which is discussed in Sections 4.1.4 and 4.2.3.

State and Event System Capability Class

The *Primitive State and Event System Capabilities* class of techniques uses the primitive system capabilities to formulate and test state and event hypotheses. To formulate a hypothesis, the investigator makes observations about the primitive system capabilities and chooses a possible state or event from $Q_{prim}(t)$ or $\Sigma_{prim}(t)$. This approach is not efficient for real systems, which have a large number of possible states and events.

To test a hypothesis, this class of techniques is efficient and it can be used to refute those that have values that are not valid for the system configuration. Example tests include:

- Does device d have an address a (i.e., DAD_{ps})?
- Is the value v within the valid range of values for address a on device d (i.e., ADO_{ps})?
- Could event e have occurred at time t (i.e., c_{pe}) ?

This class of techniques must accurately formulate and test hypotheses based on the storage and event capabilities of the system. In practice, this class of techniques is not used for hypothesis formulation, but it could be used for testing.

State and Event Sample Data Class

The *Primitive State and Event Sample Data* class of techniques uses sample data from observations of similar systems or from previous executions of the system being investigated. The sample data includes metrics on the occurrence of events and states.

When formulating a hypothesis with this technique, states and events are chosen based on how likely they are to occur (instead of an arbitrary state like the system capabilities techniques did). Testing will reveal if there is evidence to support the state or event.

Sample data can also be used to test hypotheses and support is given to only the hypotheses whose events and states have a probability that is above a threshold. With this approach, the existence of supporting data does not mean that the event or state occurred. It means that the sample data supports its occurrence. Similarly, if the sample data does not have data to support the hypothesis, that does not mean that the hypothesis is false.

This class of techniques must use sample data from systems that are similar to the system being investigated from both a capabilities and use perspective. This class of techniques is not formally used in practice because sample data does not exist and its usefulness is not clear at the primitive level.

State and Event Reconstruction Class

The *Primitive State and Event Reconstruction* class of techniques uses a state in the inferred primitive history to formulate and test hypotheses about the previous event and state. Formally, this class of techniques is solving for $e \in \Sigma_{prim}(t - \Delta t)$ and $s \in Q_{prim}(t - \Delta t)$ in:

$$h_{ps}(t, h) = \delta_{prim}(t - \Delta t)(s, e)$$

To formulate a hypothesis using this class of techniques, the investigator makes observations about a state in the inferred history and formulates hypotheses about the events that could have caused it. The time in the hypothesis should be equal to one time step before the state being observed. Otherwise, there will not be evidence in the inferred history to support the hypothesis.

The hypothesis can also include the starting state of the event. The starting state of event e will be the same as the known ending state except for the locations

that are defined by the event (i.e. $def_{prim}(e)$), which are given a value of *undefined*. In some cases, the inverse logic of the event can be used to calculate the unknown values in the starting state. For example, subtraction could be used to reconstruct the starting state of an addition event.

Reconstruction can be used to test hypotheses for events whose ending state is defined in the inferred history. The hypothesis is refuted if it is impossible for the event in the hypothesis to create the known state. If the hypothesis includes both the starting state and the event then the investigator can simulate the state and event and compare the result to the known ending state.

If the hypothesis states that only an event e could have occurred at time t, then a prediction can be made that another event could cause the known state at time $t+\Delta t$. If the prediction is tested and found to be true, then the hypothesis is refuted.

In practice, primitive-level reconstruction occurs informally when the clone of a disk is analyzed. A clone of a disk is created to preserve the state of a system and is created by copying the contents of one sector of a disk to the same sector on a second disk. Consider a situation at time t where an investigator causes primitive events to occur that read the state of the suspect storage device and write the state to a second storage device. Note that the second storage device (the clone) has its own history.

We can see this example in Figure 4.7. This shows a clone of the original disk being made at time t using a preservation process. The two disks have independent histories for the subsequent time steps.

Now consider that at time t' ($t' > t$), the investigator wants to use the clone to define a state of the system in the inferred history at time t, which is the bold drive symbol in Figure 4.7. At time t', the investigator knows only the state of the clone, which is the shaded drive symbol in the figure. Reconstruction must be performed to determine the state of the suspect system at time t. First, he must reconstruct the state of the clone back to when it was made. This is typically achieved by calculating

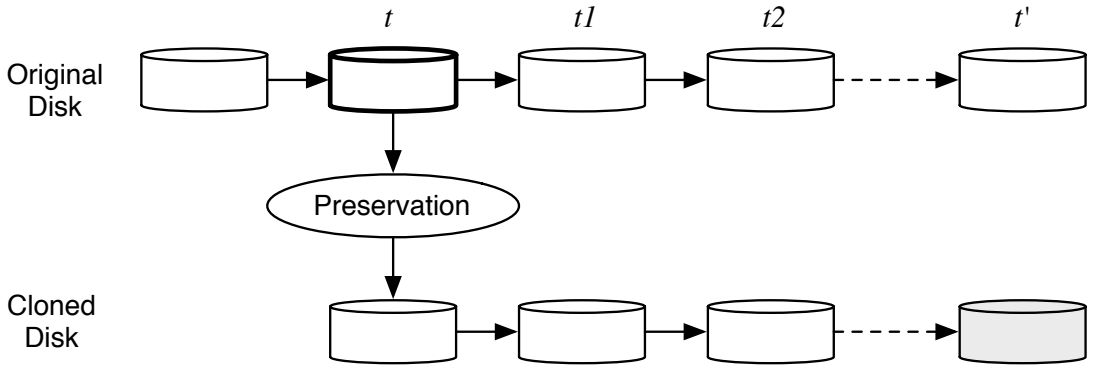

Figure 4.7. After a clone of a disk is made to preserve the state, each disk has their own history and reconstruction must be performed to define the preserved state of the disk to ensure that the current state is the same as the original.

the one-way hash of the clone when it is made so that it can be later shown that it did or did not change (this assumes that both hashing programs are accurate).

If the investigator can reconstruct the state of the clone back to its creation time, he next formulates hypotheses about the state of the suspect system based on knowledge about the preservation events. Typically, the tools used for the preservation process are well tested and the events are well understood. Therefore, the investigator can apply the inverse process to define the state of the suspect storage device at time t in its inferred history. If the tools used to make the clone were not trusted, then additional tests may need to be conducted on the tools or the clone to determine if the clone is an accurate copy. Note that if the copy of the system was made to an image file instead of a cloned disk, then the reconstruction could be a complex event reconstruction process, which has the same basic process and is described in Section 4.2.3.

When the primitive state of a system is defined from an indirect observation then reconstruction is also being informally performed. Recall that a direct observation occurs when a component is aware of something based on its senses and an indirect

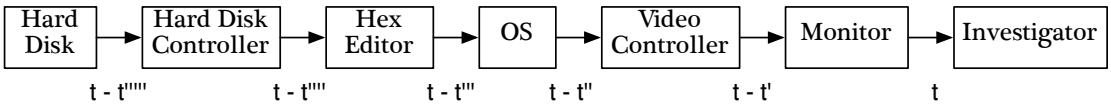

Figure 4.8. An indirect observation made at time t must be reconstructed to the original state being observed. In this figure, the information flow from the hard disk to the investigator is shown.

observation occurs when a component is aware of something based on the observations of other components. If a program is displayed on a monitor that shows the contents of a disk sector, then the investigator is making a direct observation about the state of the monitor and an indirect observation about the state of the sector. To define the state of the sector in the inferred history based on the monitor observation, event and state reconstruction must occur to define the states of the storage locations in the components that were used to make the observation, which are shown in an observation tree as defined in Section 3.2.2.

If the state being defined in the inferred history was in the observer's trusted observation zone, then less testing may need to be conducted during the reconstruction because the observer trusts all components that were used to make the indirect observation. Figure 4.8 shows an example where an investigator observes the monitor at time t and the original state of the drive was observed at time $t - t''''$. It is not practical to reconstruct the state of each of the components, but different tools can be used to verify the data and the trustworthiness of the tools can be considered.

Outside of reconstructing the state from a clone or direct observation, reconstruction is not frequently performed at the primitive level because the ending state of CPU registers and memory are not known. In addition, the investigation does not typically rely on the execution of specific primitive-level events. In the future, as memory is more frequently preserved and analyzed, this will be more common.

The requirements for this class of techniques include understanding the logic associated with the event capabilities of the system. For efficiency, it is also important to be able to identify unique signatures of events, if any exist.

State and Event Construction Class

The *Primitive State and Event Construction* class of techniques uses a state in the inferred primitive history to formulate and test hypotheses about the next event and state. Formally, this class of techniques is solving for $e \in \Sigma_{prim}(t)$ and $s \in Q_{prim}(t + \Delta t)$ in:

$$s = \delta_{prim}(t)(h_{ps}(t, h), e)$$

To formulate a hypothesis using this class of techniques, the investigator makes observations about a state in the inferred history and formulates hypotheses about the events and states that occurred after it. The time in the hypothesis should be equal to the same time as the state being observed. Otherwise, there will not be evidence in the inferred history to support the hypothesis.

The hypothesis can also include the ending state of the event. To determine the ending state in the formulation process, the investigator can simulate event e from the known state and include the ending state in the hypothesis. Construction is typically less useful than reconstruction for hypothesis formulation because the resulting hypotheses cannot be refuted unless the ending state is already known or the values in the hypothesis are not supported by the system capabilities. After all, the change in state from an event is the only proof that an event occurred [Sch94].

Construction can sometimes be used to test hypotheses for events whose starting state is defined in the inferred history. This technique works if the system prevents certain events from occurring when the system is in a certain state. If the system would prevent the event in the hypothesis from occurring when the system is in the known state, then the hypothesis is refuted.

In some systems, the state has the next event to occur encoded in it. For example, the next instruction could be pointed to by the program counter register, or equivalent. When this information exists, it can be used for hypothesis formulation and testing. However, some systems can override this value if a hardware interrupt, or similar, occurs. Therefore, a starting state alone may not be proof that the event occurred.

In practice, this class of techniques is not performed because it is rare that the ending state of the event is unknown. In many cases, an investigation is about past events and it is more common to have the ending state of the system and need to work backwards. A less formal use of this class of technique occurs when a program is analyzed to determine the events that it would have caused if it was executed. Based on the analysis, hypotheses may be formulated about the occurrence of the events. This is less formal because the resulting hypothesis is not about the event that immediately follows the state where the program was recovered from. The requirements for this class of techniques include understanding the logic associated with the event capabilities of the system.

4.2 Complex History Categories

The previous four categories can be used to define the inferred primitive history of a system. In practice, the storage system may be defined at the primitive level, but most of the analysis is performed at the abstraction levels. As will be shown, formulating and testing hypotheses about the complex system is more difficult than those for the primitive system because complex events and storage locations are software-based and not as obvious when they exist (in comparison to a physical component with a manufacturer and serial number on it).

Three more categories are now defined for the inferred complex history. The first two can be used to define the complex storage and event capabilities of the system

and the last can be used to define the complex states and events that are believed to have occurred.

4.2.1 Complex Storage System Configuration Category

The *Complex Storage System Configuration* Category of analysis techniques defines the complex storage capabilities of the system. These techniques are required if the investigator needs to formulate and test hypotheses about complex states. The following sets and functions in the complex history model are defined by this category:

- D_{cs}: The names of the complex storage types that existed during the times in T.
- DAT_{cs}: The attribute names for each of the complex storage types in D_{cs}.
- ADO_{cs}: The domain of each of the attributes in DAT_{cs}.
- DAD_{cs}: The set of unique identifiers for the possible instances of each complex storage type in D_{cs}.
- ABS_{cs}: The abstraction transformation functions for each of the complex storage types in D_{cs}.
- MAT_{cs}: The materialization transformation functions for each of the complex storage types in D_{cs}.
- c_{cs-X}: The function that maps a time and history identifier to the complex storage types that existed at layer $X \in L$.

These sets and functions are used to define the set Q_X, which defines the possible complex states. In many places in the following sections, the subscript X is used to refer to any of the abstraction levels in the L set, which is defined in the next category of analysis techniques.

There are two types of hypotheses that are formulated in this category. The first defines the D_{cs} set and c_{cs-X} function, which represent the names of the complex

Complex Storage System Configuration Category

	Complex Storage Existence Time Hypotheses	**Complex Storage Capability Hypotheses**
Classes of Analysis Techniques	- Program Identification - Data Type Observation	- Complex Storage Specification Observation - Complex Storage Reverse Engineering - Complex Storage Program Analysis
Model Variables Defined	D_{as}, $c_{as\text{-}x}$	DAT_{as}, DAD_{as}, ADO_{as}, ABS_{as}, MAT_{as}

Figure 4.9. The Complex Storage System Configuration Category of analysis techniques can formulate and test two types of hypotheses and includes five classes of analysis techniques.

storage types and when they existed. The second defines the DAT_{cs}, ADO_{cs}, DAD_{cs}, ABS_{cs}, and MAT_{cs} sets, which represent the attributes, domains, and transformation functions for each complex storage type. Figure 4.9 shows a table where the column headings are the types of hypotheses and the rows include the classes of analysis techniques that can be used to formulate and test the hypotheses. The set and function names that are defined in the inferred history model are also given. The following two sections describe the classes of techniques associated with these two types of hypotheses.

Complex Storage Existence Time Hypotheses

This section addresses the hypotheses that identify the names of the complex storage types and when they existed on the system. In theory, these hypotheses require the state of the system to be reconstructed for each time in the inferred history so that the programs at each time can be identified. There are two classes of analysis techniques for this type of hypothesis.

The *Program Identification* class of techniques uses a reconstructed state and searches it for the existence of programs. For each program identified, the analysis techniques outlined in the following section can be used to determine the complex storage types and specifications. This class of techniques must be able to identify the program types that are supported by the system, which could be in a compiled or interpreted form.

The *Data Type Observation* class of techniques identifies the type of complex data that exist in a system state to make inferences about the corresponding programs that might also exist. This process will miss the complex storage types that were not materialized onto the storage devices being analyzed and it may contain types that were not supported by the system. For example, the data type could have been received by the system, but no program existed to process it. One difficulty with this approach is determining if data are random or from an unknown complex storage type. Also, multiple complex storage types may work on the same data and this process will identify all of the types. This class of techniques must be able to identify the known types.

In practice, these classes of techniques are not performed. Instead, any complex storage type that is supported by the analysis tool is considered.

Complex Storage Capability Hypotheses

This section addresses the hypotheses for the complex storage types and their specifications. There are three classes of techniques that can be used to formulate and test these hypotheses. All of these techniques are manual and assume that the investigator has a copy of the program that created the complex storage locations. The classes of analysis techniques described in the previous section can be used to determine which programs existed.

The *Complex Storage Specification Observation* class of techniques uses observations of a specification to formulate or test hypotheses. The specification must

describe the complex storage type attributes, attribute domains, and transformation functions. Because the observation is of the specification and not the program then the corresponding hypotheses about the program could be incorrect if the program did not properly implement the complex storage types. An example of this class of techniques is the observation of the FAT [Mic00] or HFS+ [App04] file system specifications. This class of techniques must be able to properly process and interpret technical specifications.

The *Complex Storage Reverse Engineering* class of techniques uses design recovery reverse engineering techniques [CI90] to determine the complex storage details. One approach is to use the program being analyzed to store known data in a complex storage location and then analyze the materialized version to determine where the data were stored and how they were transformed. The general requirements for this class of techniques are that they understand the possible transformations, are able to analyze the materialized data, and make associations with the materialized and abstracted data.

The *Complex Storage Program Analysis* class of techniques analyzes the programs that create the complex storage locations. Static program analysis may identify the code that performs the transformations or dynamic program analysis may identify the instructions used in the abstraction and materialization process. The general requirements for this class of techniques are that they be able to analyze the executable program or source code and identify the instructions that are used to abstract or materialize data.

By design, some complex storage types are difficult to fully learn. Encryption is an abstraction where lower-level data are transformed using algorithms and a secret key. To learn this process, the key must also be learned, which is typically designed to be long enough to make exhaustive searching ineffective. Steganography is another example of abstraction where content is embedded in another file, such as an image. When the image is abstracted using the steganography tools, the content can be extracted.

In practice, the programs and complex storage types for each system are not enumerated for each investigation. Instead, analysis tools are programmed to support a finite number of complex storage types and the types are applied the corresponding data type is identified. Therefore the three classes of techniques that were described in this section are typically performed by tool vendors and not investigators. This approach makes the assumption that the programs on the system being investigated are equivalent to the programs that were originally analyzed with respect to the complex storage types.

4.2.2 Complex Event System Configuration Category

The *Complex Event System Configuration* Category of analysis techniques can be used to define the complex event system capabilities. The following sets and functions are defined by classes of techniques in this category:

- D_{ce}: The names of the programs that existed on the system during the times in T.
- L: The names of abstraction layers that contain complex events.
- DSY_{ce-X}: The event symbols for the complex events for each program in D_{ce} and at each level in L.
- DCG_{ce-X}: The state change functions for the complex events for each program in D_{ce} and at each level in L.
- ABS_{ce}: The abstraction transformation functions.
- MAT_{ce}: The materialization transformation functions.
- c_{ce}: The function that maps a time to the set of programs that existed.

The definitions of these sets and functions are used to formally define the Σ_X and δ_X sets.

Making inferences about complex events can be even more challenging than about complex storage locations because complex storage locations are designed to be both

Complex Event System Configuration Category

	Complex Event Existence Time Hypotheses	Complex Event Capability Hypotheses	Complex Event Transformation Function Hypotheses
Classes of Analysis Techniques	- Program Identification - Data Type Reconstruction	- Complex Event Specification Observation - Complex Event Program Analysis	- Development Tool and Process Analysis
Model Variables Defined	D_{ce}, c_{ce}	L, $DSY_{ce\text{-}x}$, $DCG_{ce\text{-}x}$	ABS_{ce}, MAT_{ce}

Figure 4.10. The Complex Event System Configuration Category of analysis techniques can formulate and test three types of hypotheses and includes five classes of analysis techniques.

abstracted and materialized and have a long life because of backward compatibility, while complex events are typically only designed from the top down and the rules can more easily change without loss of backwards compatibility. Further, many of the materializations are performed by software engineers and not tools, so learning the actual rules that were used may be impossible.

There are three types of hypotheses that can be used to define the sets and functions in the history model. One type of hypothesis defines the D_{ce} set and c_{ce} function, which represents the programs that existed on the system and the times that they existed. The second type defines the L, DSY_{ce-X}, and DCG_{ce-X} sets, which represent the abstraction layers, symbols, and state change functions for each event. The third type defines the ABS_{ce} and MAT_{ce} sets, which represent the abstraction and materialization transformation functions. Figure 4.10 shows a table where the column headings are the types of hypotheses and the rows include the classes of analysis techniques that can be used to formulate and test the hypotheses. The set and function names that are defined in the inferred history model are also given. The following three sections describe the classes of techniques associated with these three types of hypotheses.

Complex Event Existence Time Hypotheses

We first consider the techniques to formulate and test hypotheses about which programs existed. This requires the reconstructed state of the system at each time or it requires assumptions to be made about the states. The first technique is *Program Identification* and was previously described in Section 4.2.1. This technique searches a reconstructed state to identify the programs that exist.

The *Data Type Reconstruction* class of techniques is used to identify the data types that exist in the reconstructed states. The investigator makes inferences about which programs must have existed to create them. This approach works when few programs can create a specific data type, but is not as useful for general data types. This approach will not identify programs that did not store complex data and it may identify programs that never existed on the system. This class of techniques must be capable of identifying the type of complex data and know the programs that can create certain types of complex storage locations.

Complex Event Capability Hypotheses

This section addresses the hypotheses for the abstraction levels, complex event symbols, and state change functions. There are two classes of techniques that can be used to formulate and test these hypotheses. Both of these techniques are manual and assume that the investigator has a copy of the programs. The previous two classes of techniques can be used to determine which programs existed.

The *Complex Event Specification Observation* class of techniques uses the program specification, when one exists, to determine which abstraction layers may exist and which events could have occurred. For large programs, a formal design specification may exist that outlines the high- and medium-level events. For implementation-level events, the programming language will have a specification that describes and lists the possible events. In some cases, the specification may not have been updated as development on the program continued and it will not be an accurate repre-

sentation of the program. Because an observation of a specification is an indirect observation of the program, the program should be tested and compared with the specification. This class of techniques must be capable of understanding the technical specifications.

The *Complex Event Program Analysis* class of techniques analyze the programs to determine the events that they can cause. This is a difficult problem because most programs are stored as either primitive- or implementation-level events. Therefore, the programs themselves must be abstracted to determine which user-level events could have occurred, which requires the abstraction transformation functions, which are discussed in the next section.

In practice, comprehensive analysis is not conducted on every program on the system. The investigator has knowledge about common user-level events and system events, and may analyze specific programs in detail when needed. It is not necessary to identify every possible complex event at every level. Only the events that are relevant to the investigation must be understood.

It is also not required to identify the exact complex events in the program. For example, each program has complex events based on the programming language that was used to develop the program. The investigator may not be able to determine the exact language used and instead he may use a general language representation. If the complex events for the general language are equivalent to the complex events for the original language then the general language can be used in the investigation.

Complex Event Transformation Rule Hypotheses

The third type of hypothesis in the Complex Event System Configuration Category is about the abstraction and materialization rules. These are difficult to determine and only one class of techniques is defined. The *Development Tool and Process Analysis* class of techniques analyzes the programing tools and development process to determine how complex events are defined.

Defining the rules is a difficult problem because different abstraction layers require different techniques. For example, the abstraction rules from primitive events to implementation-level events are based on compilers and interpreters. The abstractions from implementation-level to user-level are based on the overall size and complexity of the program as well as the experience of all software engineers that have been involved with the program. In most cases, the exact rules can not be determined.

Fortunately, the analysis techniques needed by a digital investigator to learn the possible events are not unique to digital investigations. Large institutions must maintain legacy code and make periodic updates to it. This requires developers to quickly learn how a program works and make the needed updates. In software engineering, *program understanding* "is the task of recapturing the abstract design of a system in part or in full, from its source code. [PPBH91]" Program understanding techniques analyze the source code of a program to abstract it into higher-level events. An overview of some of the research approaches is given in Section 4.6.1. Program understanding is still largely a manual process and the tools are relatively difficult to use [HM04].

Fortunately, not every investigation will need to fully understand all events. The investigator needs to understand only the abstraction levels and events for which he is trying to answer questions about. For example, if he is answering questions about only user-level events then he may not need to enumerate the implementation-level events unless he is going to abstract up to user-level events from a known primitive-level event history.

4.2.3 Complex State and Event Definition Category

The previous two categories can be used to define the complex system capabilities and the *Complex State and Event Definition Category* can be used to define the

Complex State and Event Definition Category

	State and Event Occurrence Definition
Classes of Analysis Techniques	- Complex State and Event System Capabilities - Complex State and Event Sample Data - Complex State and Event Reconstruction - Complex State and Event Construction - Complex Data Abstraction - Complex Data Materialization - Complex Event Abstraction - Complex Event Materialization
Model Variables Defined	h_{ce}, h_{cs}

Figure 4.11. The Complex State and Event Definition Category of analysis techniques can formulate and test one type of hypothesis and includes eight classes of analysis techniques.

complex states and events that are believed to have occurred. In the formal model, this category defines the following functions:

- h_{cs}: The function that maps a time and history identifier to a complex state.
- h_{ce}: The function that maps a time and history identifier to a complex event.

These functions must be defined if an investigation needs to answer questions about complex states and events, such as "does this file exist" or "did this event occur."

There are eight classes of analysis techniques in this category and they are listed in the table in Figure 4.11, which shows the classes of techniques and the model variables that are defined. The eight classes of analysis techniques have directional components to them, as can be seen in Figure 4.12, which shows complex event $E1$ reading complex storage locations $D1$ and $D2$ and defining location $D1$. The lower-level events and storage locations are also shown with dotted lines.

The system capabilities and sample data classes of techniques can be used to formulate and test hypotheses about individual states and events. The reconstruction

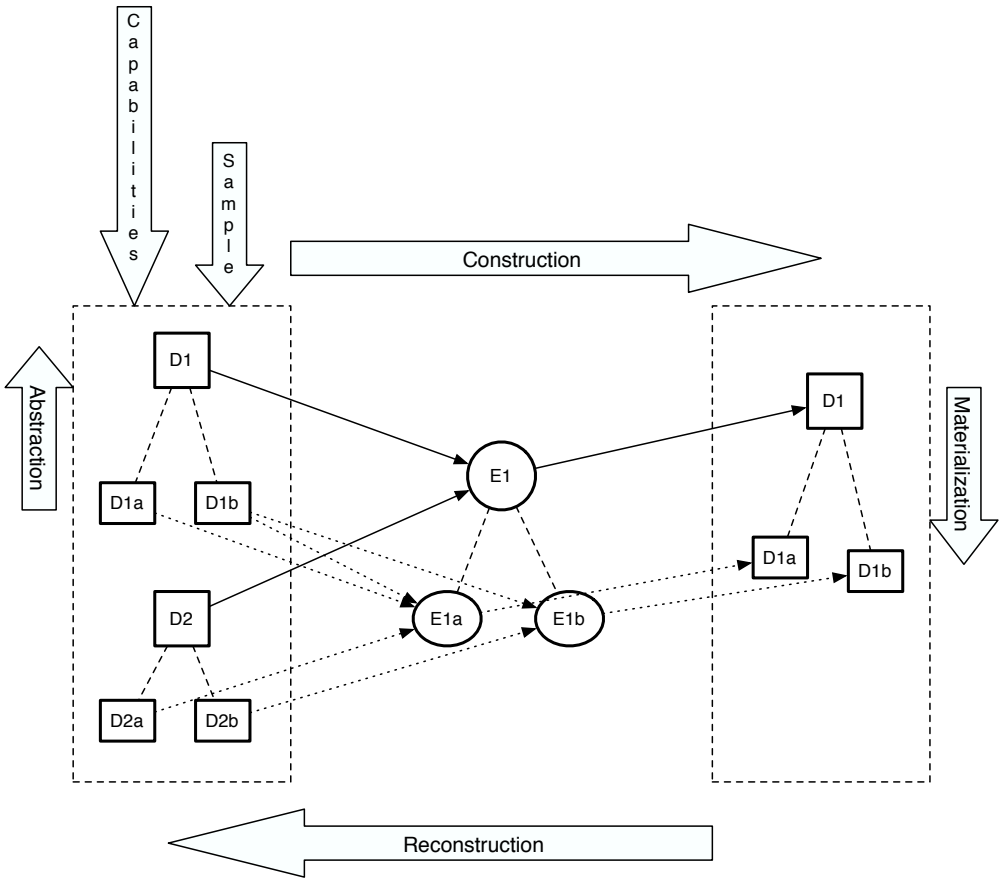

Figure 4.12. The eight classes of analysis techniques for defining complex states and events have directional components to them.

and construction classes of techniques are based on the concept that the ending state is based on the previous event and starting state:

$$h_{cs}(t) = \delta_X\left(t - y\Delta t\right)\left(h_{cs}(t - y\Delta t), h_{ce}(t - y\Delta t)\right)$$

Reconstruction uses the ending state of an event ($h_{cs}(t)$) to make inferences about the previous complex event and starting state while construction uses the starting state of an event ($h_{cs}(t - y\Delta t)$) to make inferences about the event and ending state.

While reconstruction and construction use forwards and backwards-based techniques, the abstraction and materialization classes of techniques are bottom-up and

top-down. They are based on the concept that a complex event or storage location is defined by a transformation of lower-level events or states:

$$a_1 = abs\_func(a'_1, a''_1)$$

The abstraction class of techniques define higher-level states and events based on lower-level states and events and the materialization classes of techniques define lower-level states and events based on higher-level states and events.

Because all eight techniques are different approaches to defining the same two functions, the investigator can formulate a hypothesis using one technique and test it with the other techniques. The classes of analysis techniques to define states and events are described in the following subsections.

State and Event System Capability Class

The *Complex State and Event System Capabilities* class of techniques uses the complex system capabilities to formulate and test state and event hypotheses. To formulate a hypothesis, the investigator makes observations about the complex system capabilities and chooses a possible complex state or event from $Q_X(t)$ or $\Sigma_X(t)$. This approach is not efficient for real systems, which have a large number of possible complex states and events.

To test a hypothesis, this class of techniques is efficient and it can be used to refute those that have values that are not valid for the system configuration. Example tests include:

- Does complex storage type a exist (i.e. D_{ce})?
- Are the attributes created by the transformation within the valid range of values (i.e. ADO_{cs})?
- Could complex event e have occurred at time t (i.e. c_{ce})?

A hypothesis about an unknown complex storage type, unknown event, invalid address, or out of range value can be refuted using these questions. This class of

techniques must accurately formulate and test hypotheses based on the complex storage and event capabilities of the system.

State and Event Sample Data Class

The *Complex State and Event Sample Data* class of techniques uses sample data from observations of similar systems or from previous executions of the system being analyzed. The sample data includes metrics on the occurrence of events and states.

When formulating a hypothesis with this technique, states and events are chosen based on how likely they are to occur (instead of an arbitrary state like the system capabilities techniques did). Testing will reveal if there is evidence to support the state or event.

Sample data can also be used to test hypotheses and support is given to only the hypotheses whose events and states have a probability that is above a threshold. With this approach, the existence of supporting data does not mean that the event or state occurred. It means that the sample data supports its occurrence. Similarly, if the sample data does not have data to support the hypothesis, that does not mean that the hypothesis is false.

This class of techniques must use sample data from systems that are similar to the system being investigated from both a capabilities and use perspective. In practice, this technique is not formally used as described because the appropriate sample data does not exist.

Sampling does occur in a less formal way when system and incident characteristics are used. System and Incident characteristics were previously described in Section 3.3 and are characteristics of the software, hardware, and incident type that are used to formulate hypotheses. For example, based on the OS type, we can formulate hypotheses about what the most common file systems are. We can also use the OS type to determine which applications may exist and where logs may be found. The

incident type may identify what types of evidence may exist and what events may have occurred.

State and Event Reconstruction Class

The *Complex State and Event Reconstruction* class of techniques uses a state in the inferred complex history to formulate and test hypotheses about the previous complex event and state. Formally, this class of techniques is solving for $e \in \Sigma_X(t - y\Delta t)$ and $s \in Q_X(t - y\Delta t)$ in:

$$h_{cs}(t, h) = \delta_X(t - y\Delta t)(s, e)$$

$y\Delta t$ represents y time steps of Δt and the value of y is dependent on the duration of the event being considered.

As with Primitive State and Event Reconstruction, which was discussed in Section 4.1.4, this class of techniques can be used to formulate hypotheses based on the ending state of an event and it can be used to test event hypotheses when the ending state is known. There are two differences between the complex and primitive techniques though. The first difference is because the duration of a complex event is longer than Δt and therefore the starting time of the event in a multitasking or distributed system will not be clear. An independent event could have started and ended before the reconstructed event started and ended and therefore the exact starting time will be unknown.

One way to solve this problem is to use lower-level primitive and complex states and perform the event and state reconstruction at lower-levels. The lower-levels may show when each event read from and wrote to each location and may show when distributed systems communicated. This is similar to debugging distributed programs by considering when they communicated and where an error occurred [CBM90] or using Lamport clocks to sequence events [Lam78].

A second difference between primitive and complex reconstruction is about choosing which events in Σ_X to consider. Some of the events in Σ_X may be equivalent to

each other, which means that the events cause the same state changes. Depending on which questions need to be answered, the investigator may not need to consider only one of the equivalent events.

In practice, event and state reconstruction often occurs, but less formally. More commonly, the hypothesis describes a general class of events instead of a specific event and the starting state of the event is not given. Common examples of event reconstruction include:

- Analyzing the headers of an e-mail message to determine which servers it was processed by.
- Analyzing a web browser history file to determine which web sites were visited.
- Analyzing the Most Recently Used (MRU) keys in the Windows registry to determine previous files and commands.
- Searching for the program that created a specific.

When these techniques are used, the goal is typically to identify general behavior and events and not to identify a specific complex event in a specific program. For example, the version of the e-mail server is not typically important when reconstructing the server communications in an e-mail header. Another example is to consider that all data are the effect of an event that created them. Therefore, hypotheses can be formulated about the application that created a file. Another approach is to formulate hypotheses about other objects that were effects of the same event that created the evidence and make predictions based on creation times and other temporal artifacts. Relational information about evidence can also be used to make predictions about other objects that are related to it [Cas04].

Although subtle, a common use of complex event and state reconstruction is when a preserved copy of a system is analyzed. The basic concept of this was previously discussed in Section 4.1.4 when the reconstruction of the system state was performed using a disk clone. Now, we consider if the state of the system was preserved to a complex storage location using complex events. To determine the previous state of

the system, the investigator must reconstruct the inferred history of the copy to the time that it was created and then he must reconstruct the preservation events.

A similar process occurs when the investigator observes the state of the system on an output device, such as a monitor. The monitor could be showing the output of a program that is listing the running processes, but the investigator can only directly observe the state of the monitor controller. Complex event and state reconstruction must be used to formulate a hypothesis about the corresponding data stored in kernel memory. The hypothesis should be tested to determine if data was hidden or introduced before it was displayed on the monitor. If the data that was indirectly observed is in the observer's trusted observation zone, as defined in Section 3.2.2, then less testing may need to be performed because all components used in the observation are trusted.

A more obvious example of data and event reconstruction is deleted file recovery. The investigator or tool observes a deleted file and formulates a hypothesis about the deletion event and previous state. The previous state is determined using knowledge about the deletion process and lower-level data. File carving is another example of event reconstruction, where knowledge about how a file is allocated and deleted is used to carve a deleted file out of unallocated space based on the file's header and footer values.

A common source of event reconstruction evidence is log and history files. Event logs are updated by events and therefore are evidence of the event. To reconstruct an event based on the log, the log must be reconstructed to its state after the event. During the reconstruction, evidence of log modifications should be searched for. Logs may exist at the primitive-level [DKC+02], implementation-level [Buc05] [KC03], or user- level (such as syslog, Windows Events, or a history file in a web browser).

Similarly, the temporal data associated with files are commonly used in event reconstruction. To test a hypothesis that uses file times, the investigator must search for evidence of an event that could have changed the times to an arbitrary value. He should also test the OS to determine when it updates the times .

The requirements for this class of techniques include understanding the logic associated with the complex events on the system. For efficiency, it is also important to be able to identify unique signatures of events, if any signatures exist. Section 6.3.1 describes assigning certainty to reconstructed events.

State and Event Construction Class

The *Complex State and Event Construction* class of techniques uses a state in the inferred complex history to formulate and test hypotheses about the next complex event and state. Formally, this class of techniques is solving for $e \in \Sigma_X(t)$ and $s \in Q_X(t + y\Delta t)$ in:

$$s = \delta_X(t)(h_{cs}(t, h), e)$$

The exact length of $y\Delta t$ is based on the event under consideration and the number of other processes and events that occurred in a multitasking system.

As with Primitive State and Event Construction, which was discussed in Section 4.1.4, this class of techniques can be used to formulate hypotheses based on the starting state of an event and it can be used to test event hypotheses when the ending state is known.

There are two differences between the complex and primitive techniques though. The first difference is because the duration of a complex event is longer than Δt and therefore the ending time of the event in a multitasking or distributed system will not be clear. An independent event could have started and ended before the constructed event ended and therefore the exact ending time will be unknown. Further, a complex event reads and writes values while it is executing and therefore it may not be clear when the values were read or written if multiple processes were running.

There are several potential problems here. One is that when simulating an event to determine the ending state, the investigator will not know about other events that change the state of locations that are used by the event. A second problem is that the investigator may not know the ending time of the event because he does

not know which events occurred. One way to solve this problem is to use lower-level states and perform the event and state construction at lower-levels.

As with primitive-level construction techniques, complex-level construction techniques as described in this section are not frequently used in practice. Instead, the construction concept is used to predict what events may have occured. Section 4.5.1 has an example where the investigator observes an e-mail that instructs a recipient to download a file. A hypothesis was formulated that the file was downloaded and this was tested using reconstruction techniques by looking for the downloaded file. Similarly, if the investigator observes that a program was downloaded then he may formulate a hypothesis that it was run at some point in the future, which can be tested using reconstruction. The requirements for this class of techniques include understanding the logic associated with the complex event capabilities of the system.

Data Abstraction Class

The *Data Abstraction* class of analysis techniques is a bottom up approach and uses lower-level data in the inferred history and the abstraction transformation rules defined in ABS_{cs} to formulate and test higher-level complex state hypotheses. Because many current investigations start with only primitive data, this type of hypothesis is one of the first steps in the investigation process so that the investigator can search the complex state for evidence.

To formulate a hypothesis using this class of techniques, data in the inferred history are examined to determine which complex storage location types could be applied. The hypothesis states the complex storage type and the resulting attribute values, which are determined using the abstraction transformation function. One of the easiest ways to refute a hypothesis formulated using this technique is to compare the attribute values with the range of each attribute, which would occur during the Complex State and Event System Capability techniques from Section 4.2.3.

It is not practical to consider every complex storage type and every location and instead signatures are used to identify the complex storage types. System characteristics can also be used to more efficiently formulate hypotheses. For example, when considering the sectors of a hard disk, the investigator may first consider the complex storage types for partitioning and volume systems in the first few sectors of the disk instead of considering file format types in those sectors. Next, file system types may be tested for in each partition and within each file he may test for different file formats.

In some cases, data may be capable of supporting multiple complex storage types. When this occurs, other characteristics may exist to support some complex storage types over others, even if they cannot refute the hypotheses. For example, a file may contain data that can be abstracted as ASCII as well as other binary encodings. If the file has a ".txt" extension, then this could be considered support for the ASCII abstraction, but it does not rule out the binary encodings.

Another example can be shown using the *NTFS Autodetect Test* [Car05c]. This test image contains a single disk partition that has data structures for both NTFS and Ext2 file systems. The design of NTFS and Ext2 is such that, for some disk sizes, the critical NTFS data are stored in different parts of the partition from critical Ext2 data. Therefore, the partition in this example image can be processed to produce both NTFS and Ext2 file systems. Both abstractions are valid and therefore neither hypothesis can be refuted.

This class of techniques can also be used to test hypotheses about complex storage locations when the lower-level data are known. For example, a hypothesis from event reconstruction may define the existence of a file at time t', but the hypothesis would be refuted if no lower-level data exists to support the existence of the file.

Many of the current investigation activities involve these types of hypotheses. In some cases, abstraction is added where it may not have originally existed. For example, analyzing an executable file to determine its control and data flow is an example of abstracting the file contents. The requirements for this class of tech-

niques include understanding the transformation rules, attributes, and domain of the complex storage capabilities of the system.

Data Materialization Class

The *Data Materialization* class of techniques is the reverse of data abstraction and it uses high-level complex storage locations in the inferred history to formulate and test hypotheses about lower-level complex or primitive objects. To formulate a hypothesis, the type of complex data is determined and the materialization function is applied. By design, the materialization process will produce either multiple possibilities for lower-level data or will have undefined values. This is because the abstraction process hides the lower-level details. For example, the primitive address where data are stored may not be known from the higher-level complex storage locations.

This class of techniques can be used to formulate hypotheses when a high-level object is defined in the inferred history and the underlying structure of the object is unknown. For example, event reconstruction techniques may have identified that a file existed with a certain content, but its location is not known.

This class of techniques can be used to refute hypotheses about complex storage locations when higher-level locations are defined in the inferred history. In practice, this class of techniques is rarely used for either formulation or testing because the lower-level primitive data are typically known and it is more efficient to perform the bottom-up approach of abstraction. The requirements for this class of techniques include understanding the transformation rules, attributes, and domain of the complex storage capabilities of the system.

Event Abstraction Class

The *Event Abstraction* class of analysis techniques is the bottom-up approach to defining complex events. It uses lower-level events in the inferred history to formulate and test hypotheses about higher-level events.

To formulate a hypothesis using this class of techniques, the investigator observes a sequence of events in the inferred history and groups them together to define a higher-level complex event using the transformation functions defined in ABS_{ce}. Unlike with complex storage locations, lower-level events can be part of only one complex event. If there are multiple events that could abstract a sequence of events, then new inferred histories could be defined for each.

This class of techniques can be used to refute hypotheses about complex events when lower-level events for the same time are defined in the inferred history. If the event in the hypothesis did not cause the known event then it is refuted.

In practice, this class of techniques is not formally performed and there has been little work to address it from the digital investigation perspective, but program understanding techniques apply. For example, [CG95] and [War00] describes approaches to abstract primitive-level events to implementation-level events using flow graphs and intermediate representations.

Event Materialization Class

The *Event Materialization* class of techniques is the reverse of event abstraction. This class of techniques is a top-down approach where the investigator uses high-level events defined in the inferred history to formulate and test hypotheses about lower-level complex and primitive events. This can occur, for example, when the investigator knows a user performed an action or that a high-level system event occurred, but he does not know the lower-level details.

To formulate a hypothesis using this technique, the investigator uses the materialization functions in MAT_{ce} to map a known event to the lower-level events. By the

design of abstractions, the materialization process will map one higher-level event to multiple sequences of lower-level events because the lower-level details have been removed from the complex event.

This class of techniques can be used to refute a primitive or complex event hypothesis if a higher-level event exists for the same time. If the higher-level event did not cause the event in the hypothesis then the hypothesis is refuted.

In practice, this class of techniques could be used when trying to prove or disprove a higher-level hypothesis. If two high-level events are equivalent at an abstraction level and it is important to determine which one occurred, then materialization will show the lower-level events and possibly if evidence exists to support them.

4.3 Completeness of Categories

This section gives proofs for the completeness of the seven categories of analysis techniques. The number of categories can not be proven to be correct because one could argue that there should be six categories or 38. The point is not how many categories exist, but can they completely define the inferred primitive and complex histories.

Theorem 4.3.1 *The History Duration, Primitive Storage System Configuration, Primitive Event System Configuration, and Primitive State and Event Definition Categories of analysis techniques completely define the 11 variables in the inferred primitive history as defined in Equation 2.3.*

Proof This is proved by exhaustion. The 11 variables and the corresponding analysis technique category are given in the following table.

Variable	**Category**
T	History Duration Category
D_{ps}	Primitive Storage System Configuration Category
DAD_{ps}	Primitive Storage System Configuration Category
ADO_{ps}	Primitive Storage System Configuration Category
c_{ps}	Primitive Storage System Configuration Category
h_{ps}	Primitive State and Event Definition Category
D_{pe}	Primitive Event System Configuration Category
DSY_{pe}	Primitive Event System Configuration Category
DCG_{pe}	Primitive Event System Configuration Category
c_{pe}	Primitive Event System Configuration Category
h_{pe}	Primitive State and Event Definition Category

■

Theorem 4.3.2 *The History Duration, Complex Storage System Configuration, Complex Event System Configuration, and Complex State and Event Definition Categories of analysis techniques completely define the 17 variables in the inferred complex history as defined in Equation 2.4.*

Proof This is proved by exhaustion. The 17 variables and the corresponding analysis technique category are given in the following table.

Variable	**Category**
T	History Duration Category
L	Complex Event System Configuration Category
D_{cs}	Complex Storage System Configuration Category
DAD_{cs}	Complex Storage System Configuration Category
DAT_{cs}	Complex Storage System Configuration Category
ADO_{cs}	Complex Storage System Configuration Category
ABS_{cs}	Complex Storage System Configuration Category
MAT_{cs}	Complex Storage System Configuration Category
c_{cs-X}	Complex Storage System Configuration Category
h_{cs}	Complex State and Event Definition Category
D_{ce}	Complex Event System Configuration Category
DSY_{ce-X}	Complex Event System Configuration Category
DCG_{ce-X}	Complex Event System Configuration Category
ABS_{ce}	Complex Event System Configuration Category
MAT_{ce}	Complex Event System Configuration Category
c_{ce}	Complex Event System Configuration Category
h_{ce}	Complex State and Event Definition Category

■

Proofs are not given for the completeness of the classes of analysis techniques, but the current analysis techniques can be represented in the classes. The literature on digital investigation analysis techniques focus on describing the complex storage types that exist, the types of evidence that can be found in the data, and the types of tools that can be used.

It is difficult to enumerate the current analysis techniques because a formal taxonomy does not exist. Based on the review of several introductory books [Cas04] [DCP05] [JBR05] [KH01] [NPES04], the main focus of hypothesis formulation and testing techniques is in the Primitive Observation, Data Abstraction, Complex State and Event Reconstruction, and State and Event Sample Data classes of techniques.

The primitive-level, Complex State and Event Construction, Data Materialization, Event Abstraction, and Event Materialization classes of techniques were not addressed in the literature because they are not supported by current tools and because their usefulness is not clear.

The Primitive Observation class of techniques occurs every time an investigator observes the state of an output device, whether the output device is a trusted monitor in the lab or a monitor that belongs to a suspect. If untrusted components were used to define the observed state, then the data in the observation could not be accurate.

The Data Abstraction class of techniques includes the processing of disk images into partitions, file systems, and application formats including e-mails, graphic images, and the Windows registry. This class of techniques also includes the extraction of unallocated space and slack space of files, cracking encryption, performing binary analysis to abstract executable files, and the abstraction of network packets into streams.

The Complex State and Event Reconstruction class of techniques includes the analysis of shell and web browser history files, e-mail headers, log and event files, file temporal data, deleted file recovery, and importing disk image files into an analysis tool. It also includes the observation of configuration files when the investigator analyzes them for evidence of a suspect modifying the system.

The State and Event Sample Data class of techniques includes formulating hypotheses and predictions based on the type of incident. Data hiding techniques that are common to intrusion cases are described and search methods are given. Log and configuration files and file types that commonly contain evidence for a certain type of crime were also given.

4.4 Category and Class Dependencies

The categories and classes of analysis techniques have dependencies among them. In this section, we outline some of those dependencies. Figure 4.13 shows the seven

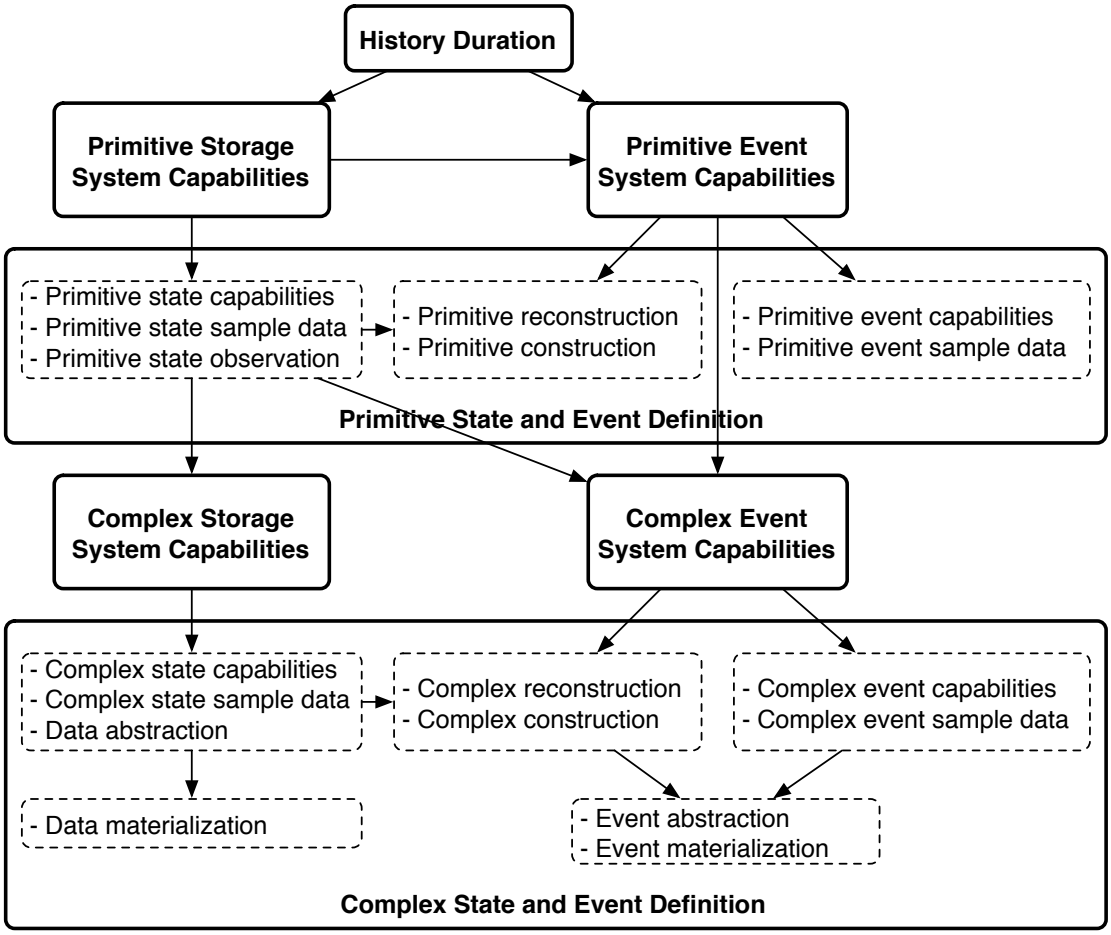

Figure 4.13. The seven categories of analysis techniques are dependent on the data defined by other categories.

categories of analysis techniques. A line from box A to box B represents a dependency of B on A where the techniques in box B depend on data that are defined by techniques in box A.

For example, the investigator cannot define a state until he has defined the storage capabilities and he cannot perform primitive state and event reconstruction until he has defined at least one primitive state. The capabilities of the complex storage and event systems cannot be determined until the state of the programs has been determined.

This set of dependencies shows what techniques must be considered during an investigation. The figure shows that if the investigator wants to perform complex event reconstruction then he must formulate hypotheses or make assumptions about the primitive storage and event systems, at least one primitive state, the complex storage and event capabilities, and at least one complex state.

4.5 Examples

We now consider some basic examples of applying the various classes of analysis techniques. We assume that the system configuration techniques have already been performed by the investigator and that he has a state of the primitive system defined from preservation.

4.5.1 File Searching

We first consider a situation where the investigator observes an e-mail message that instructed the recipient to download a file named `info.txt` from a specific URL. Based on this observation, he formulates a construction-based hypothesis that the recipient downloaded the file.

He tests the hypothesis several ways. First, he uses event reconstruction techniques and tests if a file named `info.txt` exists in the known state in the inferred history. If the file complex storage locations have not been defined yet, then they must be before the file search is conducted. Once the file abstractions are defined, the investigator can search them for one that has a name attribute whose value is "info.txt."

If the file is found, then he must test to see if it is the actual file from the website or if it is another file with the same name. He can test this by downloading the file and comparing its contents with the file that was found on the system. Note that this test does not guarantee that the file currently on the website is the same one that existed when the e-mail was sent. He can also make predictions about

which program was used to download the file and what evidence would exist from the downloading event. The test for these predictions would include analysis of the web browser application logs. This example uses an iterative hypothesis formulation and testing process where each test is more specific. After sufficient evidence has been found, a downloading event can be defined in the inferred history. An event may also be defined to represent when the recipient read the e-mail.

To formulate additional hypotheses, the contents of the `info.txt` file can be analyzed. If the `info.txt` file was not found, then the investigator can reconstruct previous states to determine if it was downloaded and deleted.

4.5.2 Image Searching

The second example is from an investigation about the existence of contraband images (photographs). From incident characteristics, it is known that the JPEG format is commonly used to encode such images. Therefore, the investigator formulates a hypothesis that "the contraband data are stored in a JPEG complex storage type".

The investigator predicts that files will be found that start with 0xffd8, which is the JPEG header signature. To test this, he searches the content of all file complex storage locations. Each file that is identified by this search is processed and viewed to identify those that are contraband. It is possible for JPEG files to be embedded in other files, such as zip and tar archive files. Therefore, the investigator also predicts that archive files may contain JPEG files. This requires the zip and tar abstraction to be applied to the appropriate files. Additional JPEG header searches are then conducted based on the archive file contents. If the searches do not find any files, then reconstruction techniques can be used to recover deleted files.

4.5.3 Event Occurrence

In this example, one of the investigation questions is "which program created file X?". To answer this question, the investigator formulates and tests hypotheses about each program that existed at the time that file X was created. Therefore, the first hypotheses must be formulated and tested about the file's creation time.

The investigator first considers that the value t_c in the "created time" attribute for the file's complex storage location is equal to the time that the file was created. This is an example of using complex event reconstruction. The investigator must first find support for this hypothesis, which can be done by testing the OS to verify it defines the time when a file is created. Before he can test the OS, he must determine which OS existed at the time in the hypothesis, t_c. This is equivalent to determining which complex events were in Σ_X at the time.

The investigator starts with the hypothesis that the current OS also existed at t_c. He tests this hypothesis using system logs that identify which updates were made. If an update is found after t_c then his hypothesis is refuted and he can then use the update log to formulate a new hypothesis. He can also test the OS version hypothesis using version numbers in boot logs and by comparing the release date of the OS version with t_c. These are all examples of complex event reconstruction. When time values in the file system metadata or logs are used, then he must also formulate hypotheses about the accuracy of the system time and whether it has been modified. The integrity of the OS must also be considered during this process because it could have been modified using rootkits or other malicious software.

If the OS version at time t_c is determined, then it can be used to test the hypothesis that it defines the creation time value when a file is created. If it does, then he has supported his hypothesis that the creation time could correspond to when the file was created. Note that this is only one explanation for the time.

He next formulates hypotheses to reconstruct the state at t_c so that the programs and complex events can be enumerated in the Σ_X set. This may require deleted file recovery techniques.

When the set Σ_X has been defined for time t_c, he can then use reconstruction to consider which programs could have created file X. In many cases, a single program cannot be identified because some programs can create any file in any location. Consider command line programs that move or copy files or "dragging and dropping" using a graphical interface. There are also network programs that can download arbitrary files and save them to arbitrary locations. All of these must be considered and hypotheses formulated for them.

To test each program hypothesis, he looks for other evidence of the event, such as a log of the files that each program downloaded. The existence of this log and its contents may support or refute the hypothesis that it was used to download the file. This will require that the state of the log be reconstructed to t_c and events that modified its contents should be considered. If a single program cannot be identified, then multiple inferred histories are defined.

In addition to the hypothesis that the creation time on the file corresponds to the time that the file is created, the investigator may also formulate a hypothesis that the creation time was modified by another program after it was created. To support this hypothesis, the program that modified the time must be located to show that the event could have occurred and evidence must also be found that the program was run. To fully test this hypothesis the investigator will need to reconstruct the previous states to catalog which programs existed. If no such program is found, then the hypothesis is refuted.

Another hypothesis that must be considered is that the OS was not used to create the file and that another program bypassed the OS. Therefore, the creation time may not have been set when the file was created. This can be tested by reconstructing the states of the OS to determine if it is possible to bypass it. The OS design will determine if this is possible or not. If it is possible, then the investigator must also

formulate hypotheses about programs and events at any time in T that could have bypassed the OS and created the file. The testing of these hypotheses will require more state and event reconstruction. Depending on the OS design and programs that existed, there could be many hypotheses that cannot be refuted.

4.6 Related Work

The techniques and procedures described in this chapter are related to research in other areas of computer science and digital investigations. This section outlines some of the related work that is not included in existing investigation tools, which are described later in Section 5.2.

4.6.1 Program Understanding

Program understanding is an area of research in software engineering and its goal is to analyze source code so that a software engineer can more quickly determine how the program works and how to maintain it. These concepts are also needed with digital investigations so that the possible complex events on a system can be enumerated when the investigator is defining Σ_X and δ_X. Different techniques are needed for each complex event layer. We examine the major layers in the following two sections.

4.6.2 Decompiling

Decompiling is associated with transforming primitive events to implementation-level events. Typically, decompiling analyzes an executable program, but the same techniques can be applied to a inferred primitive history trace. The original materialization was performed by a compiler or interpreter and they can be analyzed to determine the abstraction rules.

When analyzing the compilers, there are two general types of rules that should be considered [PPBH91] [Cif94]. One type of rule is based on instructions that cause a change of control flow. For example, instructions for function calls, swapping processes, and branching. These indicate areas that could be different abstraction levels within the implementation-level category.

The second type of rule is based on the sequences of statements for common algorithms. These allow us to abstract a sequence of events with a higher-level event. Control flow graphs (CFG), data flow graphs (DFG), and program flow graphs (PFG) are frequently used for this type of analysis.

Because of optimizations, the lower-level events that make up a higher-level event may not be contiguous. Therefore, a clean grouping of events cannot be performed. Several approaches to extract procedures from code have been proposed and they use CFGs and PFGs to rearrange the events such that the relevant events are contiguous and can be extracted (or abstracted) [GN93] [KH03].

4.6.3 User-level Events

The relationship between user-level events and implementation-level events is not defined by rules in programs. They are defined by software engineers and domain-specific requirements. Therefore, different techniques are needed to learn the rules and possible events.

There are several techniques that have been proposed to analyze programs to create higher-levels of abstraction [PPBH91] [Rug95]. Many of these are designed as tools to assist a developer and are not fully automated. Some analyze the code and include lexical analysis to organize the source code into units and provide links between them. Syntactic analysis can create a parse tree, which allows control flow analysis to be performed to determine the order of instructions within a procedure and the order of procedures within a program. Data flow analysis can also be used to show the dependence between instructions and variables or a program dependence

graph can be created to show both the control and data flow. A variation of data flow analysis is slicing, which isolates the instructions that have an effect on a given variable. Slicing will be discussed again in Section 5.6. There has also been work on different approaches to presenting the program data. SHrIMP is a system that uses a fisheye-view to allow the user to focus on areas of a program that has millions of lines of code [SMW95].

The approaches mentioned in the previous paragraph present the program in a more useful format than pure source code. They still require a human to recognize how to abstract the events. There are more automated techniques that recognize general signatures for algorithms and common functions. These are called "cliches" [RW90], "beacons" [Bro83], "plans" [SE84], and design patterns [GHJV95]. These signatures are equivalent to defining the transformation functions in the ABS_{ce} set to map a set of implementation-level events to a user-level event. Some have also proposed using program completeness proofs to fully understand what a program does [BM82].

4.6.4 Automated Event Hypothesis and Prediction Formulation

There has also been work in automating the process of formulating predictions for hypotheses about specific types of events. A basic example is the `chkrootkit` tool [Mur05]. This tool formulates predictions for the hypothesis that a system has a rootkit installed. To test this hypothesis, the tool searches for file and system signatures of specific rootkits. In the context of the model discussed in this paper, it uses incident and system characteristics and reconstruction to predict what evidence would exist if a rootkit were installed.

The DERBI system tested hypotheses about different intrusion scenarios [Tys01]. The system uses "evidence schemas" to describe what evidence may exist if a sequence of intrusion events occurred. For example, access times on certain files or modifications to log files. In the context of the process in this work, the DERBI sys-

tem uses both system and incident characteristics to formulate and test hypotheses about a system intrusion. The formalization model proposed by Leigland and Krings is similar and it describes the components of a system and the expected evidence that would exist after a specific type of attack [LK04].

Elsaesser and Tanner developed a system that uses planning to reconstruct events [ET01]. A computer network is described to the analysis system based on which computers are connected to each other and what trust exists between them. Host configuration is also defined in the analysis system. Next, different attack plans are considered and evaluated to determine if they could have occurred. For example, the software or hardware are tested to ensure that a specific attack could occur. A simulator can also be used from a known state to determine if the events occurred. The logs and evidence from the simulated system is then compared to the logs and evidence from the suspect system. In the context of the process in this work, the Elsaesser and Tanner system requires that the investigator formulate and test the system configuration hypotheses. The system then formulates and tests different event hypotheses.

Stallard and Levitt developed a program that would formulate consistency-based predictions to test if redundant information was inconsistent [SL03]. These would test a hypothesis that events occurred to remove evidence. For example, it could process the `lastlog` file on a Linux system and determine when each user was logged in. Based on this information, searches were conducted to identify files that were modified by users during times when, according to the log file, they were not logged in. A file modified by a user when he was not supposed to be logged in could be an indication that the `lastlog` file was modified.

Carney and Rogers used statistical tests to test hypotheses about which program created a file [CR04]. The motivation for this approach was to determine if it could be determined if a file was downloaded by the user or planted there by an attacker. File creation times and references to the files in question were used as metrics.

To help with general predictions, the Autopsy Forensic Browser tool was modified to make suggestions for additional searches based on evidence that was found [CS05]. The investigator would identify evidence to the tool and it would make suggestions to search for files in the same directory, with similar temporal data, or similar file names. The goal of these searches was to find files that were related to the evidence, which is a basic form of reconstruction. For example, searching for other files in the same directory or for files with the same creation time may find files that were installed at the same time.

Spatial outlier analysis has also been used to make predictions for hypotheses about the existence of files whose attributes were modified to hide the file [CS05]. The theory behind the procedure was that some attributes of a file would be statistically different from the other file attributes in the directory. For example, the times and name could be changed such that they are consistent with other files in the directory, but the starting block could be much larger. Predictions were made based on single and multiple attributes. Predictions were also made to find directories with hidden files.

4.6.5 Digital Event Reconstruction

There are several approaches to event reconstruction that have been described in the literature. One approach is to define a finite state machine of the system and go backwards from the final state using other states that were "observed" [GP04]. This process requires that Q, Σ, and δ be fully understood and therefore is used only with small systems, such as slack space of a file [Gla05] and printer queues (where only high-level complex events are considered). While both this reconstruction approach and the history model in this paper use FSMs, there are many differences. The biggest is that the history model does not require that the system be modeled as a FSM for an investigation to occur and it is not for only reconstruction. The history

model can be used even when reconstruction is not needed and it can be used to identify what assumptions are being made during an investigation.

A related concept is the use of FSMs with attack graphs. To determine how a network could be attacked and compromised, a FSM of a network can be defined and encoded in model checking software [SHJ+02]. The software will then evaluate paths from a starting state to states where a compromise occurs. This could also be used for reconstruction to determine which paths lead to the final state.

Another proposed approach is to test event reconstruction hypotheses with colored Petri Nets [Ste03] [Ste04a]. A Petri Net model is defined for the systems being investigated and the events are simulated to determine if they could have occurred. Similar to the FSM modeling, this is done at the level of high-level complex events.

Carrier and Spafford previously proposed an approach that analyzes programs on the system to determine which events could occur and then tries to map them with possible effects based on content, temporal data, or location [CS04a]. This defines part of the Σ set and may not necessarily reconstruct the full state of the system back to when the event occurred. Therefore, it assumes that the programs that exist in the observed state are the same as those that existed at the time of the incident. This approach uses the concepts of functional, relational, and temporal analysis techniques [Cas00].

One difficulty with event reconstruction is determining when the event occurred. The concepts used in prerequisite-based event correlation [NCR02] [TL00] can be used to define the relative ordering of events. Prerequisite-based event correlation has defined preconditions and postconditions for an event and therefore if the postcondition of event e_1 is a precondition of event e_2 then one may conclude that e_1 occurred before e_2. Gladyshev and Patel also propose methods for calculating the relative ordering of events and bounding the possible event times [GP05].

Garfinkel proposed Cross Drive Forensic Analysis [Gar05a], which is a high-level reconstruction of events that copy data between systems. This approach compares data, such as credit card numbers or e-mail addresses, of multiple systems to deter-

mine if they may have come form the same source. This can be represented as a high-level of event reconstruction where specific system-level events are not reconstructed, but general "copying" events are considered that would be comprised of many system-level events.

4.6.6 Analysis Layers

In [Car03], several "analysis categories" were proposed. Based on the categories and classes of techniques described in this chapter, the original categories are more accurately defined as analysis layers. The layers are defined based on the layers of abstraction that exist in storage systems.

Each layer contains information about complex storage location types and the complex events that read from and write to them. This information is used when defining the capabilities of the complex systems. The following are some, but not all, of the analysis layers. As programs create different types of complex storage locations, different types of analysis layers will be defined.

- Volume Analysis: The analysis of the volume system layer, which includes partitions, volume spanning, and other forms of data abstractions that organize data into volumes. Events in this layer include creating volumes and reading volume contents.
- File System Analysis: The analysis of the file system layer, which typically exists in the volume contents. Events in this layer include creating, deleting, and reading files.
- Application Analysis: The analysis of the application layer, which typically exist in file contents. Events in this layer include creating and updating the various data that specific applications need to store in files.
- Network Analysis: The analysis of the network layer, which includes the raw network packets and abstractions for the various network layers. The analysis of logs generated by network services, such as a firewall or web server, falls

under Application Analysis and not Network Analysis because the logs are an effect of an event. With Network Analysis, there are no events, just states. The events occur in programs, which falls under Application Analysis.

- Memory Analysis: The analysis of the memory layer, which includes the abstraction of the primitive bytes of memory to virtual memory and process spaces. Events in this layer include primitive events that load and store data and complex events that allocate and use memory.

5 APPLICATION TO PRACTICE

The previous chapters outlined a model and techniques that are capable of defining a computer's inferred history. This chapter focuses on applying the theoretical model to current practices. Section 5.1 applies the history model and classes of analysis techniques to the process models that were defined in Section 1.2.3. Section 5.2 considers the existing investigation tools based on the classes of analysis techniques. Section 5.3 describes trust and observation tools. Section 5.4 applies the model to the foundational concepts of the forensic sciences for physical world investigations and Section 5.5 addresses the Daubert guidelines for entering scientific or technical evidence into a U.S. court. Section 5.6 shows that software debugging is a special case of a digital investigation.

5.1 Process Models

As described in Chapter 1, there is not a unique process model for digital investigations. In this section, two of the process models are reviewed based on the classes of analysis techniques presented in the previous chapter. It is shown that the differences between the models is based on what types of hypotheses are formulated in each phase. The operations phases, which organize and train the investigators, of each model are not addressed in the following sections.

5.1.1 NIJ Electronic Crime Scene Model

The process model published in the NIJ guide [Uni01] has five high-level phases: Preparation, Collection, Examination, Analysis, and Reporting. The Preparation and Reporting phases are operations-based and do not apply to this work. In the

Collection phase, the investigator formulates hypotheses about the primitive storage and event capabilities and preserves the state of the primitive and complex system.

Presumably, the Examination phase is when the investigator formulates event reconstruction hypotheses about the preserved state and data abstraction hypotheses are formulated to define the complex storage locations. After establishing the primitive and complex state, the investigator can begin to formulate hypotheses about the incident, which uses Sample Data techniques. Hypotheses may also be formulated for basic event reconstruction of deleted files to determine the original content.

The Analysis phase is where more detailed analysis occurs to determine if the data identified in the Examination phase are from events related to the incident. The techniques used in this phase vary depending on the type of investigation. For example, in an intrusion investigation, this phase may perform reconstruction on files identified in the Examination phase. While many of the classes of analysis techniques that were described in Chapter 4 are not specifically addressed in this model, both phases are general enough that any technique can be performed in them.

The technical differences between the Examination and Analysis phases in this model are not clear. The distinction exists because some law enforcement agencies are organized this way to make the best use of limited technical resources. For example, an investigation for contraband photographs will have a technical person extract all digital photographs from the system in the Examination phase and another, possibly less technical, person will view each in the Analysis phase to determine if the images are contraband.

One way to describe this separation uses the hypothesis formulation and testing process. The Examination phase observes the data and formulates a hypothesis about the data that are evidence and the Analysis phase tests the hypothesis. For example, the Examination phase searches for and identifies the images and formulates a hypothesis that the images are contraband. The hypothesis is tested in the Analysis phase when they are viewed. Another way to describe the separation is that more specific hypotheses are formulated and tested in the Analysis phase.

5.1.2 Integrated Digital Investigation Process Model

The Integrated Digital Investigation process model [CS03] is based on the phases of a physical crime scene investigation. This process model has five phases for the digital crime scene investigation: Preservation, Survey, Documentation, Search, and Event Reconstruction.

The Documentation phase focuses on the documentation of evidence is therefore considered operational. The Preservation phase implements the preservation process and allows states to be saved for later analysis. This is also where the investigator could define the primitive and complex system capabilities. The Survey phase is where the obvious evidence is identified, which would require that the remaining system capabilities and complex storage locations are defined. In this phase, the investigator formulates hypotheses using the Sample Data techniques that identify the most common evidence based on the incident and system characteristics. The Searching phase formulates additional hypotheses and conducts more searches based on the evidence found during the Survey phase. The Event Reconstruction phase uses event reconstruction techniques to formulate hypotheses about events that occurred.

This model is specific about the analysis techniques that must occur in each phase, but it is not clear if the event construction techniques are supported by this framework. The Searching phase does not limit hypotheses from being formulated using construction techniques, but because reconstruction is specifically identified then the use of construction is not clear.

5.2 Existing Investigation Tools

In this section, existing digital investigation tools are compared with the classes of analysis techniques. In general, the focus of the most common tools is on the data abstraction and observation techniques. Little support exists for event analysis. In addition, data are presented as fact and the hypotheses that were formulated and tested are not always clear. For example, most file system analysis tools will

automatically attempt to determine the partition and file system types when a disk image is imported. This process formulates hypotheses about different complex storage types and tests them, but the investigator is not made aware of which types were tested and how many attempts were made to refute the hypotheses.

There are many tools that will preserve the state of a disk by making a copy of it. Popular examples include dcfldd [Har05], dd [GNU05], EnCase [Gui05], FTK [Acc05a], iLook [Int05], ProDiscover [Tec05], Safeback [New05], and SMART [ASR05]. Event reconstruction must occur when the copy is analyzed and this process is made easier by calculating the one-way hash value of the copy when it is made. The hash value is sometimes stored with the preserved data in a proprietary or open data format [ACC+06].

The contents of memory can also be preserved using software such as dd for Windows [Gar04], EnCase Enterprise, and ProDiscover IR. Tribble [CG04] is an example of a hardware device to preserve memory. The state of a network can be observed and preserved using tools such as Ethereal [Com05], NetDetector [NIK05], NetIntercept [San05], NetWitness [Man05], and tcpdump [JLM05].

Without reverse engineering the tools, it is not clear how they determine the primitive storage capabilities of devices. Some will check for hidden ATA data, such as host protected areas (HPA) and device configuration overlays (DCO) that were discussed in Section 4.1.2. One method to test that the preservation process produced an accurate copy is to use multiple tools, but the copying process is time intensive and it is not common for an investigator to preserve a drive using multiple tools. Some investigators will calculate the one-way cryptographic hash of the data before it is preserved and the hash will be compared to the preserved copy.

The primitive event capabilities of a system are not considered by existing investigation tools. This is because the primitive-level events are not typically used in digital investigations. There is also no formal support for determining when each storage and event device was connected to the system. This information is typically determined from the manual analysis of log and configuration files.

To define states and events in the inferred primitive history, the tools will import preserved data to define a state. This process was described as event reconstruction in Chapter 4, but the tools use the assumption that the preserved data is accurate. Analysis tools such as EnCase, FTK, iLook, ProDiscover, The Sleuth Kit [Car05d], SMART, and WinHex [X-W05] allow the investigator to observe primitive disk sectors and searches can be manual or automated. There is no formal support for representing which primitive events occurred at each time, but event hypotheses can be tested and refuted by searching the primitive states.

To determine the complex system capabilities, there is no formal support. All of the tools support some complex storage types, but the transformation rules are defined by the vendor and are not system specific. Using the concepts of the history model, this is equivalent to defining all known complex storage types in the D_{cs} set and not only the types that existed on the system. If unknown complex storage types are encountered, the tools do not provide a formal way of learning the rules. Typically, the data can be viewed in hexadecimal or imported into the original program (which was not designed for digital investigations). Password recovery tools, such as Password Recovery Toolkit [Acc05b], will try to determine an encryption key, which is equivalent to learning part of the transformation function.

The possible complex events are also not enumerated by analysis tools, which is because general event reconstruction and construction techniques are not supported by analysis tools. Deleted file recovery is one of the few reconstruction hypotheses that can be automatically formulated and the logic associated with the technique is encoded into the analysis program.

The data abstraction class of techniques is one of the main features of current investigation tools for both system and network analysis. For example, many volume systems, file systems, and application abstractions are supported by EnCase, FTK, iLook, ProDiscover, The Sleuth Kit, SMART, and WinHex. The kntlist [Gar05b] and memparser [Bet05] tools abstract a copy of Windows memory. Network analysis

tools such as Ethereal, NetDetector, NetIntercept, and NetWitness abstract raw network data into streams.

The event abstraction class of techniques is not supported by many of the dead analysis tools, but is supported by event monitoring and correlation tools. These tools receive and abstract data from running systems after a complex event occurs.

When it comes to state and event reconstruction, most tools support only deleted file recovery. Some tools present the temporal file and network data in a timeline format so that the investigator can manually perform event reconstruction to explain why files were accessed or changed, but it is not automated. Most tools do not allow complex events to be manually defined, with the exception of Autopsy [Car05a].

5.3 Trusted Software and Hardware

In Section 3.2.2, the concept of a trusted observation zone was defined. The zone included the closest observation components that the investigator trusted to reliably produce accurate data. An investigator relies on many complex components when observations are made about digital data.

A component is trusted because it is believed to reliably produce accurate data. To become trusted, a component should be tested to ensure that its results are accurate. Components may be tested using a combination of techniques, such as a code reviews and white or black box testing.

To date, there are few formal standards for testing the components that are used to make observations in a digital investigation. The NIST CFTT [Uni05b] group has tested disk imaging and write blocking components and the tests cover all of the components in the observation path and not only the investigation tools. For example, when testing the imaging abilities of the `dd` tool in Linux, a fault was identified in the Linux kernel where the last sector of an odd sectored disk would not be copied [Uni02]. The `dd` tool was being tested, but a bug in the OS was instead found.

Many components are used during a digital investigation. Standard OSs and hardware are used to run specialized software and their accuracy is just as important as the accuracy of the specialized components, yet it is the specialized components that are referred to when testing is discussed. One reason for this is because the hardware and OS reliably produce accurate data during normal operation. Otherwise, no one would buy the hardware or software. In general, specialized digital investigation tools do not require special features of the hardware or software and therefore the standard components are trusted because they are reliable during normal operation.

Any change in a component could change its accuracy and the investigator will trust each component differently. In an ideal scenario, the investigator would trust all of the components required to make an observation and would be able to determine when a component changes. This is difficult because components from the suspect's system, such as the hard disk, are typically needed to determine the state of the system.

When all of the components are trusted, then this is typically called a *dead analysis*. The *dead* refers to the fact that the suspect's OS and applications are not being executed to analyze the data. When untrusted components exist, it is typically called a *live analysis* because this situation occurs when the system being analyzed is running. In most discussions of live versus dead analysis, the trustworthiness of the hardware is not considered – only the OS and applications are.

The current distinction between the live and dead terms does not directly translate to trusted versus untrusted concepts. Hardware devices, such as Tribble [CG04], can be used to accurately acquire memory contents of a running system even if the OS and applications are not trusted. Tribble is a PCI card that can directly read memory and write the contents to a storage device and does not rely on the OS. Using the concept of the trusted observation zone, the data in memory is in the trusted observation zone of the PCI card if the hardware involved is trusted.

In 1984, Thompson described a compiler that would modify programs to introduce insecure code [Tho84]. A code review of the program being compiled would not

reveal the issue and the point was that even code that is manually reviewed cannot be trusted. This is even more true today with the additional complexity of programs and systems and it makes assigning trust values to software and hardware during an investigation difficult.

5.4 Forensic Science Concepts

There are general concepts on which physical forensic science is based [IR01]. This section addresses each of them and considers if they apply to digital investigations.

5.4.1 Transfer

In the physical world, an important concept is *transfer*, which occurs when matter divides and attaches to other objects. It can be seen in one of the foundational principles, Locard's exchange principle [IR01]:

> No one can commit a crime with the intensity that the criminal act requires without leaving behind numerous signs of it: either the offender has left signs at the scene of the crime, or on the other hand, has taken away with him – on his person or clothes – indications of where he has been or what he has done.

Transfer is based on the fracture of physical matter. "Matter divides into smaller component parts when sufficient force is applied [IR01]." The following three corollaries apply:

- "Some characteristics retained by the smaller pieces are unique to the original item or to the division process. These traits are useful for individualizing all pieces to the original item."
- "Some characteristics retained by the smaller pieces are common to the original item as well as to other items of similar manufacture. We rely on these traits to classify the item."

- "Some characteristics of the original item will be lost or changed during or after the moment of division and subsequent dispersal; this confounds the attempt to infer a common source."

The concept of transfer is used as the basis for many other important concepts that are addressed in the following sections.

If the complex history model in this work is considered, the atoms in the physical world make up the primitive state. Abstractions are defined by grouping atoms into molecules and the molecules are grouped into the macroscopic objects that people interact with. The transfer concept is based on an event that breaks an object up into two or more objects, but the same underlying atoms and molecules exist in the various parts.

There are differences with the abstractions of physical matter and the abstractions of data. In the digital world, a single primitive storage location can be part of multiple higher-level complex storage locations, but a molecule is in only one macroscopic object at a time. This will make the divisibility concept more difficult to apply to the digital world because lower-level data could be in multiple parts after a division.

In special cases, the divisibility concept exists in digital systems. For example, a file is typically composed of several hard disk sectors and a data structure exists to link them together. When the file is deleted, the links between the sectors may be lost and the content is divided. This is an example of divisibility and transfer. However, there is nothing to prevent the OS from wiping the sectors with 0s during the deletion process. Wiping the contents of the file when it is deleted is equivalent to being able to tear out the binding of a book and turn the torn pages into cups of coffee so that there is no evidence that the book ever existed.

While the divisibility concept does not exist as strongly as in the physical world, there is a weaker concept of transfer. This concept exists because values defined by a digital event are consequences of the event logic and the values used by the event. Therefore, an investigator cannot draw conclusions about the construction

of complex storage locations, but he can draw conclusions about what their value is. For example, if a location is defined with the value 4 then he knows that the value 4 either exists in the event logic, in the input data, or it was computed from a combination of input data and logic.

In many physical investigations, the question of *source* is important and it is based on the division of matter concept. For example, the investigator may have a carpet fiber. Its sources include the piece of carpet that it came from and the factory where it was made. Similarly, the sources of a fired bullet include the gun that shot it and the factory that made it. In all cases, the source is an entity that caused and controlled an event that affected the evidence.

In a computer, a source of an object is defined as an event or object that had an effect on the object's value. Therefore, sources of digital data include the process that defined it, the copy of the program that was used to load the process, the location where the program came from, and the source code that was compiled to form the program. For example, the sources of a digital document could include a word processor program, a file whose content was copied into this document, and a keyboard that a user typed with.

5.4.2 Identification

The *identification* concept of a physical investigation determines what an object is. For example, it could determine the molecular composition of a liquid or powdery substance. This would be used, for example, when the investigator needs to determine if a substance is an illegal drug and he does not care about where it was made. Using the history model and classes of techniques defined in this work, identification uses abstraction techniques to formulate hypotheses about the characteristics and attributes of the object.

In the digital world, identification is performed when the complex storage type of data must be determined. Identification can occur using several techniques. One is

to use signatures of the data, which is what the `file` Unix command does. Another approach is to use machine learning [dV04].

Hash databases can also be used to identify known data. To identify a file, its one-way hash is calculated and the data base is referred to. The largest public hash database is the National Software Reference LIbrary (NSRL) from NIST [Uni05c]. There has also been research on identifying a file based only on its fragments [Kin04].

5.4.3 Classification

The *classification* concept deals with class characteristics, which are "traits that are produced by a controlled process [IR01]." These are used to group objects into categories based on their sources. The identification concept in the previous section is about the object itself, while this concept is about the source of the object. Classification in the physical world can be done by basic visual observations as well as more technical techniques. For example, we can group the source of objects into general categories, such as cars, computers, and bags. The objects can then be further grouped into categories based on the manufacturer and then factory. Using the classes of analysis techniques, this concept uses event and state reconstruction techniques to formulate hypotheses about the general class of the objects that caused the object to exist.

Classification can also be performed in the digital world. This can be used to associate data with hardware types, OS types, and programs. This uses event reconstruction and makes inferences about the system capabilities and specific events that created the data. Event reconstruction is being performed at high-levels and classes of events are considered instead of specific events.

For digital data, the classification structure and hierarchy or applications needs to be better defined. Such a structure would have uses outside of only classification. For example, an investigator could use the classification to describe which file types are relevant to an investigation and which can be ignored. The NIST NSRL hash

database describes the application type of the files, but version 2.10 has 424 unique application types. This is a large number for an investigator to consider for every situation and multiple levels could help to make it more manageable.

5.4.4 Individualization

Individualization is the concept that an object can be linked back to a unique source, instead of only a general class. For example, a bullet would be tied back to the specific gun that fired it, not only the class of gun types that could have fired it. This concept uses individualizing characteristics, which are "traits that are produced by a random, uncontrolled process [IR01]." Uniquness is "a belief, not a fact [IR01]." It is believed to exist in nature, such as in DNA and fingerprints, and it also exists as objects wear down from use. For example, most objects that come out of a factory are nearly identical because they are not made using a random process, but each object will be used differently in the future, which may make them unique. Handmade objects may have unique characteristics about them as well.

In the digital world, there is, typically, no randomness. Events are deterministic and there are a finite number of inputs. Objects do not "wear" in a random process from use. The current design of systems does not introduce any randomness into different installations of programs. Therefore, the best an investigator can expect would be to link data to a version of a program, but not a specific installation. Microsoft Word used to store a computer's MAC address when the computer created a document, which links the document back to its source, but it is trivial to change or remove the address. The ability to reliably tie data back to its original source using its characteristics would require characteristics that are based on encryption and digital signatures and identifiers that are unique to each program and system.

Formally, this is part of the event reconstruction class of techniques where the investigator identifies the program that caused the event. This is a lower-level form of reconstruction than is used with classification.

5.4.5 Association

The *association* principle focuses on inferences "of contact between two objects, the source of the evidence, and the target on which it was found [IR01]." This comes into play after individualization has occurred and it links the source of the evidence with the crime. For example, if a gun is found to be the source of a fired bullet, the investigator must associate the gun with a person who could have brought it to the crime scene. The association must be performed for the evidence to be relevant in the conviction of a person.

Association is likely a larger problem with digital investigations than with physical investigations because there is an analog-digital interface between the digital events and the person. The user interacts with digital events using a keyboard, mouse, and monitor. The nuances and randomness of the user are sampled into finite values. Research has been conducted to use mouse and keyboard characteristics as a method of detecting unauthorized access [Lan00] or as an authentication method [Tra05]. These characteristics have also been proposed for use in digital investigations [SGM06].

With physical objects, the object may be in the possession of the person and DNA or fingerprint evidence may exist to link the object with the person. In some digital investigations, fingerprints on the keyboard are used to link a person to a computer [CAV04] and the existence of a password on an account can reduce the number of people who could have used it. Biometric authentication and more strict access control could help in the future with linking a person to digital events.

Formally, this principle is part of event reconstruction and tries to identify the human element in the causes of an event. This is in comparison to the classification and individualization principles that determine the source object in the event cause.

5.4.6 Reconstruction

The final principle of forensic sciences is *reconstruction*, which is "the ordering of events in relative space and time based on the physical evidence [IR01]." The most common types of reconstruction with physical evidence are blood splatter [BG02], bullet ballistics, and car accidents. In all three of these cases, the event involved is well understood using concepts from physics and unknown variables can be determined, such as the source location or car speed. Reconstruction also occurs for other types of events, but the results can be highly subjective when considering the actions and responses of victims and perpetrators. Inman and Rudin describe several reconstruction reports for the murder of Marilyn Sheppard in 1954 [IR01]. Many of them do not fully consider and test alternative hypotheses.

In the digital world, we previously saw the reconstruction class of techniques to reconstruct complex or primitive events. There are also examples in the digital world where there are few unknowns and event reconstruction can be reliably used. For example, if the investigator preserves the state of the system using a reliable tool and maintains the integrity of the copy, then he can later reconstruct the preserved state. Also, if the system has a log that records which events occurred, the investigator understands the events being logged, and he can show that the log is reliable then he can reconstruct the logged events. It is not clear how many other types of events can be reconstructed in a digital system without special logging systems. One benefit of the digital world over the physical is the existence of computer programs, which show the events that could have occurred. The difficulty is that it may not be possible to identify which programs were used.

5.5 Daubert Guidelines

The Daubert Guidelines were introduced in Chapter 1 and they are used in many U.S. states when entering scientific or technical evidence into court. The guidelines consider if a procedure has been published, if it is accepted by the relevant

community, if it has been tested, and its error rate. To date, it has not been clear what types of digital investigation procedures the Daubert guidelines should apply to.

With the inferred history model and the basic process outlined in Chapter 3, it is more clear. With physical crimes, Daubert has been used with respect to the testing phase in the scientific method. A hypothesis is formulated and there are different ways to test it. Each hypothesis testing procedure may undergo a Daubert hearing.

With a digital investigation, there are many hypothesis testing procedures because each set and function in the inferred history is defined by formulating and testing hypotheses. Examples of hypothesis testing procedures include:

- Determining primitive and complex system capabilities.
- Reconstructing previous primitive and complex states and events.
- Abstracting and materializing states and events.
- Searching the primitive and complex states and events.

If testing procedures must be evaluated under the Daubert guidelines, then the abstraction, materialization, reconstruction, and searching techniques should be published and tested.

5.6 Software Debugging

The process of software debugging is similar to a digital investigation and we propose that debugging is a special case of a digital investigation. Debugging focuses on the states and events associated with one or more programs at the implementation level of abstraction. Debugging typically focuses on which implementation-level events occurred to cause an error and questions such as "why was the incorrect value produced" are asked.

Debugging has several advantages over a general digital investigation because a person who is debugging a program generally has the source code and some knowl-

edge about the program. He can also more easily try to reproduce the events that could have caused the error and may have a smaller set of events to consider.

Many debugging methods, such as dynamic program slicing [KL88] [ADS91] [Gop91] [Tip95], use the notion of an execution trace to help determine the previous events. This is similar to the primitive event histories, but the execution trace is typically stored for only the CPU and not all computational devices. Similar to an investigator, a person debugging a program would want to view the previous events at a higher level than the machine instructions. There are several languages that can be used to define classes of complex events [BW82] [OCH91]. When a debugging monitor recognizes the events described in these languages, it represents it with the complex event. The same languages could be used in investigation tools to define how lower-level events should be grouped.

Backward static program slicing may also be useful for general digital investigations. Backward static program slicing is a debugging technique that considers one or more variables at a specific location in a program. The technique creates one or more slices of the program that contain the instructions that may be directly or indirectly dependent on the variables. A *s*lice with criterion (n, V) is the subset S of statements in program P that, when executed, computes the same values for the variables in V at vertex n in P. Also, when the program P halts then S will also halt. A slice is not an execution trace, instead it is the program P with zero or more statements removed from it.

There are many methods for statically determining a backward program slice, such as using control and data flow graphs [Wei84] [Lyl84], program dependence graphs [OO84], and information-flow relations [BC85]. Backward program slicing will not help us determine which branch was executed or how many times a loop was executed, but it will allow the investigator to reduce the instructions that he needs to consider. Therefore, if an investigator is reconstructing a system and has the program that was executed, static slicing could be useful to determine which events occurred.

6 SUMMARY

6.1 Accomplishments

This work has shown that it is possible to define the concepts of a digital investigation based on computing concepts. Previously, digital investigation concepts were organized into different phases of process models and the connections between different process models or the completeness of the models was not clear. It was also not clear what it meant to be a digital investigation tool.

In this work, the concept of an object's history was introduced that, in the physical world, includes the object's senses and direct observations. For a computer, the history includes the states and events that occurred. During an investigation, the investigator learns as much of the history as possible.

To define a theoretical model of a computer's history, it was assumed that the computer could be represented by a finite state machine (FSM). The basic FSM model was too limited and it was extended to include support for removable devices and abstractions. Removable device support allows the history model to represent the devices that were connected to the system at each time and support for abstractions allow the history model to include the occurrence of complex states and events, which are crucial to investigations that focus on files and user events. This work does not require that a system be reduced to a FSM though during a real investigation.

The digital investigation process was defined based on the scientific method and the computer history model. Formally, a digital investigation formulates and tests hypotheses about the configuration, state, and event variables in the history model. Humans cannot directly observe digital states and events and therefore all statements about digital events and states are hypotheses, some of which can be tested more accurately and easily than others.

Based on the types of hypotheses that can be formulated and tested, seven categories and 31 unique classes of analysis techniques were defined. A proof of completeness was given to show that the categories could be used to define all variables in the model. This creates a technology-independent classification that applies to any system that can be represented by a FSM. The classification also more clearly shows what procedures and methods are used in a digital investigation. This will help to determine what may fall under the guidelines for entering scientific or technical evidence into a court.

6.2 Implications of Work

One implication of this work is the ability to show how the various process models are different. It has been understood that an investigator can complete an investigation using many of the process models, but it was not clear what they had in common. This work shows that the models focus on different types of hypotheses in each of the phases and it shows more of the details that were not formally defined in a general "analysis" phase that many models have.

This work has shown the links between computer science and digital investigations. The design of a system impacts what types of evidence will exist and therefore can support or hinder an investigation. Because complex storage locations and complex events are created by programs and are frequently investigated, understanding how programs work is important to an investigation. There are many areas of software engineering that apply to digital investigations. Program understanding techniques can help an investigator analyze a program to determine what events it can cause and other forms of static and dynamic program analysis can be used to identify the evidence that a program creates. Debugging can be viewed as a special case of a digital investigation where source code is available, the questions are about implementation-level events, and it is typically easier to recreate the environment

where errors were observed. In the future, these computer science concepts can help to solve crimes and resolve incidents.

When the theoretical model is compared to best practices, it becomes clear where assumptions are being made. Many of them can be justified, but it is important to understand that they are being made. For example, when identifying the configuration of the primitive system the possible primitive events and state transitions are not typically identified and it is assumed the CPU has standard logic. This is typically a safe assumption because any changes to the logic would require the user to have custom programs that recognized the non-standard logic.

A more significant assumption is about what programs existed and what their capabilities were. To accurately identify the complex events that could have occurred at each time in the computer's history, the state of the system must be reconstructed to each time because the possible complex events are dependent on the programs that exist. In practice, an assumption is sometimes made that the programs that exist in the final state of the system and the programs that can be recovered define the possible events in the previous states.

The assumption about which programs exist is important when the Trojan horse defense is considered in criminal cases. The Trojan horse defense is the digital version of the "Some Other Dude Did It" (SODDI) defense for physical crimes [BCH04]. With the Trojan horse defense, the defendant reports that an attacker or malware planted the evidence of the crime on his computer. When this defense is used, it could be important to determine which program was used to create evidence. Was the file created because the user sitting at the computer downloaded it using a web browser or was it created because someone broke in? To formally answer this question, the state of the system will need to be reconstructed back to the time that the file was created. This is typically not going to be possible and therefore assumptions must be made and justified.

Assumptions are also made about the state change function of complex events. Because program analysis tools do not exist that can enumerate the possible user

events, investigators will "experiment" with programs to determine what the programs can do and what the resulting evidence is. When similar evidence is found on the suspect system, the investigator may formulate a hypothesis that the suspect performed the same event that the investigator previously did. Yet, there could be other events that create similar evidence that the investigator is not aware of. There could also be small differences in the evidence from different events and these may not be known. These assumptions are typically made because there is not detailed knowledge about all common events.

By defining an investigation as a hypothesis formulation and testing process, scientific principles can be more clearly applied. It has been debated whether or not digital investigations were a science and this work shows that some of the procedures can be considered scientific if the corresponding hypotheses are scientific and appropriately tested. A scientific approach would occur if the investigator formulates a hypothesis that can be refuted and he tries to refute it during the testing process.

This work has also shown the differences between the forensic science disciplines in the physical world and "digital forensics." The physical world disciplines focus more on comparing unknown substances with known samples, identifying the source of an object, or reconstructing events with few unknown variables. Conversely, digital investigations focuses on more types of hypotheses including the search for evidence, the existence of abstractions, and general event reconstruction.

Two related and important concepts in physical analysis are transfer and divisibility. These are based on objects interacting at the molecular level and allow some objects to be linked to a source. These concepts do not strongly exist in the digital world because any transfer or divisions are caused by software and the developers of the software may choose to have divisibility or not.

In the physical world, some events have been identified that can be reconstructed with a high level of certainty. These are events where there are few unknown variables and they are solved for during the reconstruction. Because of the complexity of

digital events and the number of equivalent event sequences, it is not clear what types of events and states can be accurately reconstructed.

Another difference with the physical world is the lack of direct observations or indirect observations that can be easily trusted in the digital world. There are few facts with a digital investigation that do not need some level of testing or justification. While a direct observation can be made of an output device, event and state reconstruction is needed to define the state of a hard disk sector or the state of a complex storage location. This requires that each component involved in the observation must be tested and understood. With the physical world, there are more empirical observations that are performed by sight. Chemicals, special lights, and microscopes may be used to make observations easier, but these are typically well understood and do not change. Instruments may be needed for some physical observations, but these can be tested in the lab.

A final difference with the physical world is the rate of change. There are few constants in the digital world, such as gravity, laws of motion, and molecular biology. The "laws" of a computer can change with each update and can be changed by an attacker. Forensics in the physical world takes advantage of the constant "laws of nature," but digital investigations have different "laws" each time. Because of the constant change, it remains to be seen if digital investigations will be able to obtain equivalent rigor as the physical forensic sciences.

6.3 Future Directions

The final section of this chapter addresses future work that can be performed based on this work. The areas of certainty values, a complex event database, event and state logging devices, and partitioned systems are discussed.

6.3.1 Certainty Values

This work shows that the entire investigation process can be described as a series of hypotheses. Frequently, statistics are used to test scientific hypotheses and a certainty value is assigned to them. It is not clear how certainty values can be calculated for some types of digital investigation hypotheses.

Different types of hypotheses will require different types of tests and certainty value calculations. For example, tests for hypotheses about indirect observations of the current state will be based on the components that were used to make the observation and how much trust the investigator has in the components. If the software and hardware are trusted and have been tested to be accurate, then a high certainty value may exist. If the software is not trusted, such as during incident response on a running system, then calculating the certainty value is more difficult. It is not clear how to assign certainty values to these hypotheses or how to measure the notion of trust so that independent parties can compare the amount of trust they have in a component.

The certainty value for hypotheses about the existence of complex storage locations is based on the certainty value of the lower-level data as well as the certainty value of the hypotheses for the abstraction transformation rules (the ABS_{cs} set). The certainty value of the transformation rules would be based on how close they are to the rules used by the original program, but it is not clear how they would be calculated because many of the rules are currently determined from reverse engineering techniques.

When event and state reconstruction are considered, it becomes clear that without reliable logs that record which events occurred then low certainty values exist because there is typically not a unique path to each state. Therefore, there could be multiple previous events and the number increases for each step backwards. Appendix A describes a basic simulator with four 4-bit registers and fifteen instructions. When the previous state and event were reconstructed, there were 44 possible

inferred histories. There are 362,659,196 inferred histories if seven time steps are reconstructed. If we assume that all events are equally likely then the certainty of each history will be $2.76 \times 10^{-7}\%$. These numbers will be much larger if we considered an actual computer with thousands of unique instructions and storage locations.

While the lack of certainty associated with event reconstruction is unfortunate, it is not much different from physical world investigations. A crime scene investigation does not typically try to reconstruct every event and movement at the crime scene unless video evidence, or similar, exists. Only a small number of events are reconstructed.

Casey proposed a scale of certainty based on the amount of supporting or contradictory evidence and the difficulty of tampering with the data [Cas02]. While this is a potentially useful approach with respect to the value of data, it does not directly help with event reconstruction and data abstraction certainty values. It is also not formal with respect to how to measure the difficulty of tampering with data.

6.3.2 Event and State Logging Devices

The certainty values of event reconstruction are typically low because little evidence of events exist and even less of it is not prone to tampering. Therefore, future systems and work should focus on secure logs of events and states.

Dunlap et al. developed the ReVirt system [DKC+02], which was a virtual machine that created data checkpoints and recorded the non-deterministic instructions, such as user inputs, so that the system could be replayed. To investigate the system, the investigator would stop the replay and install and execute tools to collect data about the system state. The ReVirt system defines some known states and events and relies on the computer to perform the state transitions during replay.

The second generation of their work was a system called Backtracker [KC03]. Instead of recording primitive events and data, this system records system call information and keeps track of dependencies between complex storage locations. There-

fore, event reconstruction of system call-level events can be performed. If the system is investigated, the investigator may be able to use the dependencies to trace back to the source of the incident.

Buchholz proposed a system that binds labels to information as it passes through a system [Buc05]. This allows complex event reconstruction to occur at levels beyond only system calls.

There are other systems that focus only on the file system. S4 is a "self-securing" storage server that keeps copies of data after it has been modified [SGS+00]. This allows deleted data to be recovered. In the history model and analysis techniques, S4 allows the investigator to reconstruct primitive and complex states from the final state to the time when the logging started. The investigator does not have specific event information, such as which program modified the data, but he has the previous values.

The Repairable File Service [ZC03] expands on the ideas of S4 and uses a client-side system call logger to record which files are dependent on each other. This is to track down where corruption has spread to and can also be used for complex event reconstruction.

Any type of additional logging will require a secure method for storing the logs. The trusted computing designs propose tamperproof hardware [Tru04] that could be used to record and store logs. Although, some trusted computing designs could also make investigations more difficult because hard drives may work only in a specific system and because data could be encrypted with a key stored in the tamperproof hardware [Mas05].

While additional logging will help to reconstruct events during a digital investigation, future techniques cannot rely only on the logs. Video cameras can help to reconstruct physical crimes, but they do not exist in every house and location where a crime may occur. Criminals do not install cameras in their house and they will not likely install logging tools on their computers. Therefore, additional logging will help corporate investigators because they have control over the systems that they

investigate, but it will not help law enforcement and other investigators who must investigate a suspect's systems.

6.3.3 Complex Event Database

One of the major difficulties associated with complex event reconstruction is determining which events exist in programs and what their traces are. The traces, or evidence, of an event could change between each program version and the investigator must keep track of these updates. A database, or central repository, with this type of information would be useful to more quickly determine which events occurred. For example, when a temporary file is found, the database could be used to find out which event may have created the file. Such a database would be similar to databases that exist to match tool marks, cartridge cases, shoe marks, drug tablet logos [Ger02], and tireprints [Tir05].

The database would also be similar to the NIST NSRL, which currently contains the hash values of known files. The event library would also contain a description of the possible complex events that a program may cause. For example, the library could contain the list of files that each program reads from or writes to. When one of the files change between two observed states, the library can be used to identify programs and events that could have been involved.

The library could focus on the types of events that are frequently asked about in an investigation. For example, it could contain events that deal with only file storage or network traffic. These are the types of events for which long term evidence may exist. The database could also contain only the user-level events, because they are most frequently searched for and the lower-level implementation details can be ignored

One of the major limitations to this situation is that determining the user-level events of programs is not trivial. As was described in Section 4.6.1, the program

understanding and comprehension work is not fully automated, especially for user-level events, which tend to be domain specific.

6.3.4 Partitioned Systems

When investigating a live system that is critical to a business or other organization, there is currently a tradeoff between shutting the system down so that trusted software can be used and keeping the system running so that the server can still function. Future system designs could create better partitions so that if one partition is compromised then it can be investigated from other partitions that are still trusted. Using the trusted observation zone concept from Section 3.2.2, a partitioned system would allow the primitive state of a running system to be in the investigator's trusted observation zone.

One way of achieving this is using virtual machines. The host machine can investigate the virtual machine if the virtual machine is compromised. The state of memory and hard disk can be easily preserved without using the compromised system.

Another method could use an OS that confines applications and prevents an attacker from compromising the entire system because of a vulnerability in one application. For example, mandatory access control (MAC) security policies can be defined for Security-Enhanced Linux [Nat05] that confine a compromised application. Therefore, the damage from a compromise is limited to the application's domain and can be more easily identified during an investigation. The investigator needs to focus only on the objects to which the application had access.

Another benefit of a partitioned system is that the design may allow trusted partitions to still be used as servers and service requests. This is similar to conducting a physical investigation in a large building. The entire building is not closed off during the investigation, only those places that are believed to have evidence.

LIST OF REFERENCES

LIST OF REFERENCES

[Acc05a] Access Data. *Forensic Toolkit*, 2005. Available at: `http://www.accessdata.com`.

[Acc05b] Access Data. *Password Recovery Toolkit*, 2005. Available at: `http://www.accessdata.com`.

[ACC+06] Frank Adelstein, Brian Carrier, Eoghan Casey, Simson L. Garfinkel, Chet Hosmer, Jesse Kornblum, Jim Lyle, Marcus Rogers, and Philip Turner. Standardizing Digital Evidence Storage. *Communications of the ACM*, 49(2), February 2006.

[ADS91] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Dynamic Slicing in the Presence of Unconstrained Pointers. In *Symposium on Testing, Analysis, and Verification*, 1991.

[App04] Apple Computer Inc. *Technical Note TN1150 HFS Plus Volume Format*, March 2004.

[ASR05] ASR Data Acquisition and Analysis. *SMART*, 2005. Available at: `http://www.asrdata.com`.

[Ass05] Association of Chief Police Officers. *Good Practice Guide for Computer based Electronic Evidence*, 2005. Available at: `http://www.nhtcu.org`.

[BC85] Jean-Francois Bergeretti and Bernard A. Carre. Information-flow and Data-flow Analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7(1), 1985.

[BC04] Nicole Lang Beebe and Jan Guynes Clark. A Hierarchical, Objectives-based Framework for the Digital Investigation Process. In *Proceedings of the 2004 Digital Forensic Research Workshop (DFRWS)*, 2004.

[BCH04] Susan Brenner, Brian Carrier, and Jef Henninger. The Trojan Horse Defense in Cybercrime Cases. *Santa Clara Computer and High Technology Law Journal*, 21(1), 2004.

[Bet05] Chris Betz. *memparser*, 2005. Available at: `http://www.dfrws.org/2005/challenge/memparser.html`.

[BG02] Tom Bevel and Ross M. Gardner. *Bloodstain Pattern Analysis: With an Introduction to Crime Scene Reconstruction*. CRC Press, second edition, 2002.

[BM82] Victor R. Basili and Harlan D. Mills. Understanding and Documenting Programs. *IEEE Transactions on Software Engineering*, SE-8(3), May 1982.

[Bro83] Ruven Brooks. Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, 18, 1983.

[Buc05] Florian Buchholz. *Pervasive Binding of Labels to System Processes*. PhD thesis, Purdue University, 2005.

[BW82] Peter Bates and Jack C. Wileden. EDL: A Basis for Distributed System Debugging Tools. In *Proceedings of the Fifteenth Hawaii International Conference on System Sciences*, 1982.

[Car02] Brian Carrier. *Open Source Digital Forensic Tools: The Legal Argument*, 2002. @stake Technical Report.

[Car03] Brian Carrier. Defining Digital Forensic Examination and Analysis Tools Using Abstraction Layers. *International Journal of Digital Evidence (IJDE)*, 1(4), Winter 2003.

[Car05a] Brian Carrier. *The Autopsy Forensic Browser*, 2005. Available at: `http://www.sleuthkit.org`.

[Car05b] Brian Carrier. *File System Forensic Analysis*. Addison Wesley, 2005.

[Car05c] Brian Carrier. *NTFS Autodetect Test Image 1*. Digital Forensic Tool Testing Images, 2005. Available at: `http://dftt.sourceforge.net/test10/index.html`.

[Car05d] Brian Carrier. *The Sleuth Kit*, 2005. Available at: `http://www.sleuthkit.org`.

[Cas00] Eoghan Casey. *Digital Evidence and Computer Crime*. Academic Press, first edition, 2000.

[Cas02] Eoghan Casey. Error, Uncertainty, and Loss in Digital Evidence. *International Journal of Digital Evidence (IJDE)*, 1(2), Summer 2002.

[Cas04] Eoghan Casey. *Digital Evidence and Computer Crime*. Academic Press, second edition, 2004.

[CAV04] Ana Castello, Mercedes Alvarez, and Fernando Verdu. DNA from a Computer Keyboard. *Forensic Science Communications*, 6(3), July 2004.

[CBM90] Wing Hong Cheung, James P. Black, and Eric Manning. A Framework for Distributed Debugging. *IEEE Software*, 7(1), January 1990.

[CG95] Cristina Cifuentes and John Gough. Decompilation of Binary Programs. *Software - Practice and Experience*, 25(7), 1995.

[CG04] Brian D. Carrier and Joe Grand. A Hardware-Based Memory Aquisition Procedure for Digital Investigations. *Journal of Digital Investigations*, 1(1), 2004.

[CI90] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, January 1990.

[Cif94] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[Com04] Committee on Scientific Assessment of Bullet Lead Elemental Composition Comparison. *Forensic Analysis: Weighing Bullet Lead Evidence*, 2004.

[Com05] Gerald Combs. *Ethereal*, 2005. Available at: `http://www.ethereal.com`.

[CR04] Megan Carney and Marc Rogers. The Trojan Made Me Do It: A First Step in Statistical Based Computer Forensics Event Reconstruction. *International Journal of Digital Evidence (IJDE)*, 2(4), Spring 2004.

[CS03] Brian Carrier and Eugene H. Spafford. Getting Physical with the Digital Investigation Process. *International Journal of Digital Evidence (IJDE)*, 2(2), Fall 2003.

[CS04a] Brian D. Carrier and Eugene H. Spafford. Defining Event Reconstruction of a Digital Crime Scene. *Journal of Forensic Sciences (JFS)*, 49(6), 2004.

[CS04b] Brian D. Carrier and Eugene H. Spafford. An Event-based Digital Forensic Investigation framework. In *Proceedings of the 2004 Digital Forensic Research Workshop (DFRWS)*, 2004.

[CS05] Brian D. Carrier and Eugene H. Spafford. Automated Digital Evidence Target Definition Using Outlier Analysis and Existing Evidence. In *Proceedings of the 2005 Digital Forensic Research Workshop (DFRWS)*, 2005.

[DCP05] Chris Davis, David Cowen, and Aaron Philipp. *Hacking Exposed: Computer Forensics Secrets & Solutions*. McGraw-Hill / Osborne, 2005.

[Den05] Peter J. Denning. Is Computer Science Science? *Communications of the ACM*, 48(4), April 2005.

[DKC+02] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[dV04] Olivier de Vel. File Classification Using Byte Sub-stream Kernels. *Journal of Digital Investigation*, 1(3), 2004.

[Eng73] Erwin Engeler. *Introduction to The Theory of Computation*. Academic Press, Inc., 1973.

[ET01] Christopher Elsaesser and Michael C. Tanner. Automated Diagnosis for Computer Forensics. Technical report, The MITRE Corporation, August 2001.

[Fed04] Federal Bureau of Investigation. *Statement on Brandon Mayfield Case*, May 24, 2004. Press Release.

[Fed05] Federal Bureau of Investigation. *FBI Laboratory Announces Discontinuation of Bullet Lead Examinations*, September 1, 2005. Press Release.

[Gar04] George Garner. *Forensic Acquisition Utilities*, 2004. Available at: `http://users.erols.com/gmgarner/forensics/`.

[Gar05a] Simson L. Garfinkel. Cross Drive Analysis and Forensics. (in submission), November 2005.

[Gar05b] George M. Garner Jr. *kntlist*, 2005. Available at: `http://www.dfrws.org/2005/challenge/kntlist.html`.

[Ger02] Zeno Geradts. *Content-Based Information Retrieval from Forensic Image Databases*. PhD thesis, University of Utrecht, 2002.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[Gla05] Pavel Gladyshev. Finite State Machine Analysis of a Blackmail Investigation. *International Journal of Digital Evidence (IJDE)*, 4(1), 2005.

[GN93] William G. Griswold and David Notkin. Automated Assistance for Program Restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, 1993.

[GNU05] GNU Coreutils. *dd*, 2005. Available at: `http://www.gnu.org/software/coreutils/`.

[Gop91] Rajiv Gopal. Dynamic Program Slicing Based on Dependence Relations. In *Proceedings of the Conference on Software Maintenance*, 1991.

[GP04] Pavel Gladyshev and Ahmed Patel. Finite State Machine Approach to Digital Event Reconstruction. *Journal of Digital Investigation*, 1(2), 2004.

[GP05] Pavel Gladyshev and Ahmed Patel. Formalising Event Time Bounding in Digital Investigations. *International Journal of Digital Evidence (IJDE)*, 4(2), 2005.

[Gui05] Guidance Software. *EnCase Forensic Edition*, 2005. Available at: www.encase.com.

[Gut96] Peter Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *Proceedings of the 1996 Usenix Security Symposium*, 1996.

[Gut01] Peter Gutmann. Data Remanence in Semiconductor Devices. In *Proceedings of the 2001 Usenix Security Symposium*, 2001.

[Har05] Nick Harbour. *dcfldd*, 2005. Available at: `http://dcfldd.sourceforge.net/`.

[HM04] Greg Hoglund and Gary McGraw. *Exploiting Software: How to Break Code*. Addison Wesley, 2004.

[Hou00] Houghton Mifflin Company. *The American Heritage Dictionary*, fourth edition, 2000.

[Int02] International Organization on Computer Evidence. *G8 Proposed Principles For The Procedures Relating To Digital Evidence*, 2002. Available at: `http://www.ioce.org`.

[Int05] Internal Revenue Service. *iLook Investigator*, 2005. Available at: `http://www.ilook-forensics.org`.

[IR01] Keith Inman and Norah Rudin. *Principles and Practice of Criminalistics: The Profession of Forensic Science*. CRC Press, 2001.

[JBR05] Keith J. Jones, Richard Bejtlich, and Curtis W. Rose. *Real Digital Forensics*. Addison Wesley, 2005.

[JLM05] Van Jacobson, Craig Leres, and Steven McCanne. *TCPDUMP*, 2005. Available at: `http://www.tcpdump.org`.

[JN03] Stuart James and Jon Nordby, editors. *Forensic Science: An Introduction to Scientific and Investigative Techniques*. CRC Press, 2003.

[KC03] Samuel T. King and Peter M. Chen. Backtracking Intrusions. In *Proceedings of the 2003 ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[KH01] Warren Kruse and Jay Heiser. *Computer Forensics: Incident Response Essentials*. Addison Wesley, 2001.

[KH03] Raghavan Komondoor and Susan Horwitz. Effective, Automatic Procedure Extraction. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 33, Washington, DC, USA, 2003. IEEE Computer Society.

[Kin04] Ketil Kintel. Using Hash Values to Identify Fragments of Evidence. Master's thesis, Gjvik University College, 2004.

[KL88] Bogdan Korel and Janusz Laski. Dynamic Program Slicing. *Information Processing Letters*, 29(3), 1988.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), July 1978.

[Lan00] Terran Lane. *Machine Learning Techniques for the Computer Security Domain of Anomaly Detection*. PhD thesis, Purdue University, August 2000.

[Lav05] Denise Lavoie. Mass. High Court to Hear Challenge to Fingerprint Evidence. *Boston Globe*, September 6, 2005.

[LK04] Ryan Leigland and Axel W. Krings. A Formalization of Digital Forensics. *International Journal of Digital Evidence (IJDE)*, 3(2), 2004.

[LPM01] Henry Lee, Timothy Palmbach, and Marilyn Miller. *Henry Lee's Crime Scene Handbook*. Academic Press, 2001.

[Lyl84] James R. Lyle. *Evaluating Variations in Program Slicing for Debugging*. PhD thesis, University of Maryland, 1984.

[Man05] ManTech International Corporation. *NetWitness*, 2005. Available at: `http://www.forensicsexplorers.com/`.

[Mas05] Stephen Mason. Trusted Computing and Forensic Investigations. *Digital Investigation*, 2(3), 2005.

[Mic00] Microsoft. *FAT32 File System Specification*, December 2000.

[Mor98] Bernard M. Moret. *The Theory of Computation*. Addison Wesley, 1998.

[MR05] Matthew Meyers and Marc Rogers. Computer Forensics: Meeting the Challenges of Scientific Evidence. *Research Advances in Digital Forensics*, 2005.

[Mur05] Nelson Murilo. *chkrootkit*, 2005. Available at: `http://www.chkrootkit.org/`.

[Nat05] National Security Agency. *SE Linux*, 2005. Available at: `http://www.nsa.gov/selinux/`.

[NCR02] Peng Ning, Yun Cui, and Douglas Reeves. Analyzing Intensive Intrusion Alerts Via Correlation. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, 2002.

[New05] New Technologies, Inc. *SafeBack*, 2005. Available at: `http://www.forensics-intl.com/`.

[NIK05] NIKSUN. *NetDetector*, 2005. Available at: `http://www.niksun.com/`.

[NPES04] Bill Nelson, Amelia Phillips, Frank Enfinger, and Chris Steuart. *Computer Forensics and Investigations*. Thomson Course Technology, 2004.

[OCH91] Ronald A. Olsson, Richard H. Crawford, and W. Wilson Ho. A Dataflow Approach to Event-based Debugging. *Software - Practice and Experience*, 21(2), 1991.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The Program Dependence Graph in a Software Development Evironment. In *SDE 1: Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. ACM Press, 1984.

[Pal01] Gary Palmer. A Road Map for Digital Forensic Research. Technical Report DTR-T001-01, DFRWS, November 2001. Report From the First Digital Forensic Research Workshop (DFRWS).

[Pop98] Sir Karl Popper. Science: Conjectures and Refutations. In E. D. Klemke, Robert Hollinger, David Wyss Rudge, and A. David Kline, editors, *Introductory Readings in the Philosophy of Science. Third Edition*, pages 38–47. Prometheus Books, 1998.

[PPBH91] Santanu Paul, Atul Prakash, Erich Buss, and John Henshaw. Theories and Techniques of Program Understanding. In *CASCON '91: Proceedings of the 1991 Conference of the Centre for Advanced Studies on Collaborative research*, pages 37–53. IBM Press, 1991.

[QU98] Willard V. Quine and Joseph S. Ullian. Hypothesis. In E. D. Klemke, Robert Hollinger, David Wyss Rudge, and A. David Kline, editors, *Introductory Readings in the Philosophy of Science. Third Edition*, pages 404–414. Prometheus Books, 1998.

[RCG02] Mark Reith, Clint Carr, and Gregg Gunsch. An Examination of Digital Forensics Models. *International Journal of Digital Evidence (IJDE)*, 1(3), Fall 2002.

[Rug95] Spencer Rugaber. Program Comprehension. Draft 2, May 1995.

[RW90] Charles Rich and Richard C. Waters. *The Programmer's Apprentice*. Addison Wesley, 1990.

[Ryn02] Joseph Rynearson. *Evidence and Crime Scene Reconstruction*. National Crime Investigation and Training, sixth edition, 2002.

[Saf00] Richard Saferstein. *Criminalistics: An Introduction to Forensic Science*. Pearson, seventh edition, 2000.

[San05] Sandstorm Enterprises. *NetIntercept*, 2005. Available at: `http://www.sandstorm.net`.

[SB03] Fred Smith and Rebecca Bace. *A Guide to Forensic Testimony*. Addison Wesley, 2003.

[Sch94] David A. Schum. *Evidential Foundations of Probabilistic Reasoning*. John Wiley and Sons, 1994.

[Sci05] Scientific Working Group on Digital Evidence. *ASCLD Glossary Definitions: Version 1.0*, 2005. Available at: `http://www.swgde.org`.

[SD04] Jonathan Saltzman and Mac Daniel. Man Freed in 1997 Shooting of Officer. *Boston Globe*, January 24, 2004.

[SE84] Elliot Soloway and Kate Ehrlich. Empircal Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, SE-10(5), 1984.

[SGM06] Shelly Seier, David Greer, and Gavin W. Manes. Linking individuals to digital information. In *Proceedings of the Second IFIP WG 11.9 International Conference on Digital Forensics*, 2006.

[SGS+00] John. D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A.N. Soules, and Gregory R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation*, 2000.

[SHJ+02] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. Automated Generation and Analysis of Attack Graphs. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2002.

[SKS02] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw Hill, fourth edition, 2002.

[SL03] Tye Stallard and Karl Levitt. Automated Analysis for Digital Forensic Science: Semantic Integrity Checking. In *Proceedings of the 2003 Annual Computer Security Applications Conference (ACSAC)*, 2003.

[SMW95] Margaret-Anne D. Storey, Hausi A. Muller, and Kenny Wong. Manipulating and Documenting Software Structures using SHrIMP Views. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*, page 275, Washington, DC, USA, 1995. IEEE Computer Society.

[Ste03] Peter Stephenson. Modeling of Post-Incident Root Cause Analysis. *International Journal of Digital Evidence (IJDE)*, 2(2), Fall 2003.

[Ste04a] Peter Stephenson. The Application of Formal Methods to Root Cause Analysis of Digital Incidents. *International Journal of Digital Evidence (IJDE)*, 3(1), Summer 2004.

[Ste04b] Peter Stephenson. A Comprehensive Approach to Digital Incident Investigation. *Information Security Technical Report*, 8(2), 2004.

[Sud97] Thomas A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison Wesley, second edition, 1997.

[Tec05] Technology Pathways. *ProDiscover DFT*, 2005. Available at: `http://wwww.techpathways.com`.

[Tho84] Ken Thompson. Reflections on Trusting Trust. *Communication of the ACM*, 27(8), August 1984.

[Tip95] Frank Tip. Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3, 1995.

[Tir05] Tire Global Information. *Tire Database 2005*, 2005. Available at: `http://www.tgi.co.at/`.

[TL00] Steven J. Templeton and Karl Levitt. A Requires / Provides Model for Computer Attacks. In *Proceedings of the 2000 Workshop on New Security Paradigms*, 2000.

[Tra05] Issa Traore. *BioTracker*, 2005. Available at: `http://www.isot.ece.uvic.ca/projects/biotracker/`.

[Tru04] Trusted Computing Group. *TCG Specification Architecture Overview*, April 2004.

[Tur36] Alan M. Turing. On Computable Numbers With an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, 2*, 42, 1936.

[Twi85] William Twining. *Theories of Evidence: Bentham and Wimore*. Standford University Press, 1985.

[Tys01] Mabry W. Tyson. DERBI: Diagnosis, Explanation and Recovery from Computer Break-ins. Technical report, SRI Artificial Intelligence Center, January 2001. Available from: `http://www.dougmoran.com/dmoran/PAPERS/DerbiFinal.pdf`.

[Uni93] United States Court of Appeals for the Ninth Circuit. *Daubert, et al. v. Merrell Dow Pharmaceuticals, Inc., 509 U.S. 579*, June 28, 1993.

[Uni01] United States National Institute of Justice Technical Working Group for Electronic Crime Scene Investigation. *Electronic Crime Scene Investigation: A Guide for First Responders*, July 2001.

[Uni02] United States National Institute of Justice. *Test Results for Disk Imaging Tools: dd GNU fileutils 4.0.36, Provided with Red Hat Linux 7.1*, 2002.

[Uni05a] United States Department of Justice. *HashKeeper*, 2005.

[Uni05b] United States National Institute of Standards and Technology. *Computer Forensic Tool Testing*, 2005. Available at: `http://www.cftt.nist.gov/`.

[Uni05c] United States National Institute of Standards and Technology. *National Software Reference Library*, 2005. Available at: `http://www.nsrl.nist.gov/`.

[War00] M. P. Ward. Reverse Engineering from Assembler to Formal Specifications via Program Understanding. In *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, page 11, Washington, DC, USA, 2000. IEEE Computer Society.

[Wei84] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4), 1984.

[X-W05] X-Ways Software Technology AG. *WinHex*, 2005. Available at: `http://www.x-ways.net`.

[ZC03] Ningning Zhu and Tzi-Cker Chiuch. Design, Implementation, and Evaluation of Repairable File Service. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN'03)*, 2003.

APPENDIX

A 4-BIT SIMULATOR RECONSTRUCTION

Section 6.3.1 outlined methods to calculate the certainty value of hypotheses, but actual numbers were not given. This appendix considers a simple computer with a limited number of registers and instructions so that we can calculate certainty numbers.

The system that we examine is a limited implementation of the Random Access Stored Procedure (RASP) machine [Eng73]. It has only four registers, which are labeled `RC`, `R1`, `R2`, and `R3`. Each register has 4 bits and can store numbers in the inclusive range of -8 to 7. There are a total of fifteen instructions:

- `ADD R1:  RC = RC + R1`
- `ADD R2:  RC = RC + R2`
- `ADD R3:  RC = RC + R3`
- `SUB R1:  RC = RC - R1`
- `SUB R2:  RC = RC - R2`
- `SUB R3:  RC = RC - R3`
- `MOD R1:  RC = RC % R1`
- `MOD R2:  RC = RC % R2`
- `MOD R3:  RC = RC % R3`
- `STO R1:  R1 = RC`
- `STO R2:  R2 = RC`
- `STO R3:  R3 = RC`
- `CLA R1:  RC = R1`
- `CLA R2:  RC = R2`

- `CLA R3:  RC = R3`

To calculate certainty values we consider a situation where the investigator finds the system in the following state:

- `RC`: 1
- `R1`: 2
- `R2`: -5
- `R3`: 1

Based on the observation of this state, the inferred primitive history is defined for time t ($h_{ps}(h1, t)$). We do not know the program that was executed before this state, but we want to perform reconstruction to determine the previous events and states.

To perform the reconstruction, a program was developed. The program constructed a hash table where the key was the state of the four registers and the content was the possible previous events and corresponding states. Using this program, the state at each time before the final state could be evaluated.

When the program was used to consider the previous events, the results in Table A.1 were determined. The table shows the event names and the probability that it could have been the event that occurred at time $t - \Delta t$. An event has a probability of 0% if there is no starting state that could have caused the final state using the event. If there are multiple starting states that an event could have had to create the final state, then it will have a higher probability.

These results show that if all events are equally likely than the most probable event at time $t - \Delta t$ was the `STO R3` or `CLA R3` event. This is because the `R3` and `RC` registers have the same value in the final state. The `STO R3` event copies the value in `RC` to `R3` and therefore `R3` can have any value in the starting state of the event and still result in the final state. Similarly with the `CLA R3` event, which copies the value in `R3` to `RC`. The starting state of the event can have any value in `RC`. To a smaller degree, the `MOD R1` event also has several starting states that can end with the needed final state. Specifically, `RC` can have any odd value in the starting state.

Table A.1
The possible events and their probability after reconstructing one time step from a known state.

Event	**Probability**	**Event**	**Probability**	**Event**	**Probability**
`ADD R1`	2%	`ADD R2`	2%	`ADD R3`	2%
`SUB R1`	2%	`SUB R2`	2%	`SUB R3`	2%
`MOD R1`	9%	`MOD R2`	5%	`MOD R3`	0%
`STO R1`	0%	`STO R2`	0%	`STO R3`	36%
`CLA R1`	0%	`CLA R2`	0%	`CLA R3`	36%

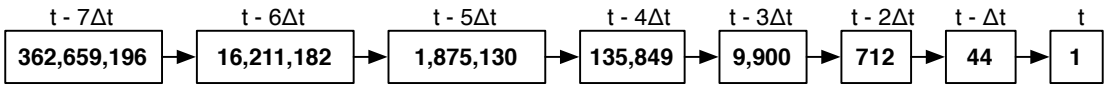

Figure A.1. This shows the number of possible histories that exist at each time when a single state is used for reconstruction. After one time step there were 44 possible histories and after seven time steps there were 362,659,196.

Because a single event was not identified, nine additional inferred histories are created, named $h2$ to $h10$. All of the histories have the same state for time t and each has a different event for $h_{pe}(h1, t - \Delta t)$. We could also apply a minimum certainty value threshold if we did not want to create a history for each possible event. At this point, the certainty value of each history is equal to the probability of the event.

For each of the ten inferred histories, we can next evaluate the state at time $t - \Delta t$. For the `ADD` and `SUB` events we can simply apply the inverse event to find the starting state. There are four possible states for the `MOD R1` event and the certainty value of each is based on the probability of each state existing.

Table A.2 shows the results from continuing the reconstruction to seven time steps. The second column contains the number of inferred histories with a non-zero certainty value. The number of unique histories are also given. A history is not unique if there exists another history that has the same starting and ending states, although the events in between could be different. Figure A.1 shows the number of possible histories at each time step and Figure A.2 shows a graph of both the total and unique number of possible histories at each time.

This example shows that the history of even a simple computer with only 15 instructions and four 4-bit registers can be difficult to determine when only the final state is known. In this simulation, each history would have a certainty value of $2.76 \times 10^{-7}\%$ if we assume that all events are equally likely. These numbers

Table A.2
A final state was used to reconstruct the possible histories that could result in this state. This table shows the results from seven time steps and includes time step, the total number of possible histories, and the number of unique histories.

Time	**Total Histories**	**Unique Histories**
$t - \Delta t$	44	31 (70.45%)
$t - 2\Delta t$	712	166 (23.31%)
$t - 3\Delta t$	9,900	489 (4.94%)
$t - 4\Delta t$	135,849	1,677 (1.23%)
$t - 5\Delta t$	1,875,130	5,748 (0.31%)
$t - 6\Delta t$	16,211,182	10,282 (0.06%)
$t - 7\Delta t$	362,659,196	34,541 (0.01%)

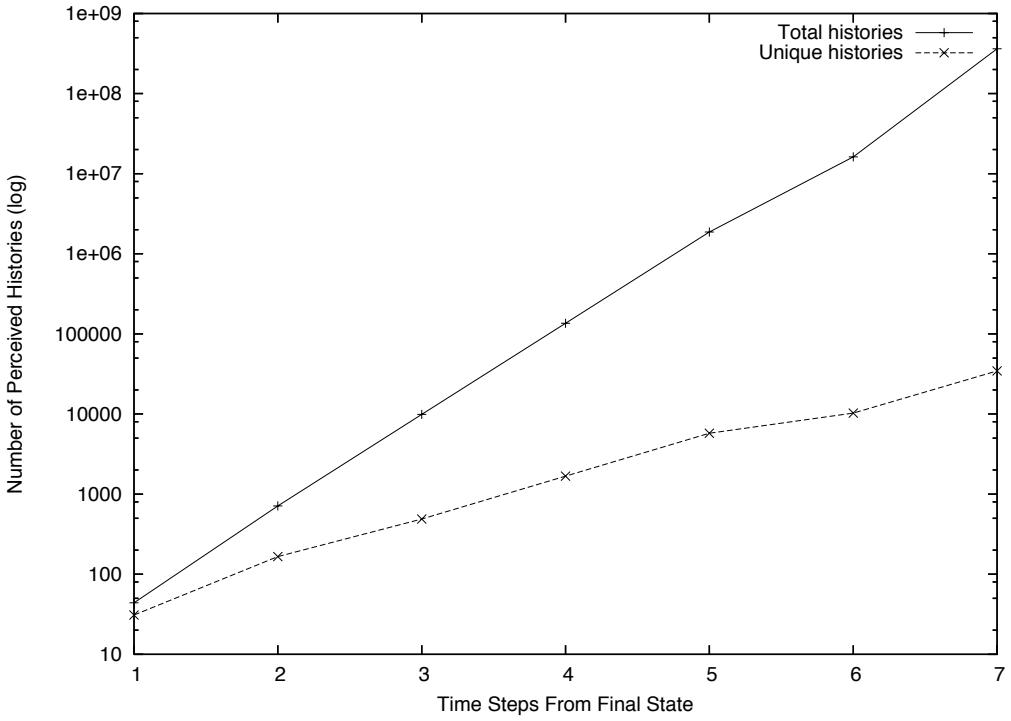

Figure A.2. A graph that shows the total possible and unique inferred histories at each time step during reconstruction.

will be much larger if we considered an actual computer with thousands of unique instructions and storage locations.

VITA

VITA

Brian Carrier received the Ph.D. degree in Computer Science from Purdue University in 2006 with Eugene H. Spafford as his major professor. He received the M.S. degree in Computer Science from Purdue University in 2001 and the B.S. degree in Computer and Electrical Engineering from Northeastern University in 1999. From 2001 to 2003, Brian was a Research Scientist at @stake, Inc.

Brian's research interests are in the areas of digital investigations and information assurance. He is the author of the *File System Forensic Analysis* book and several digital forensic analysis tools, including The Sleuth Kit and the Autopsy Forensic Browser. Brian has been involved with the European Commission's CTOSE project on Digital Evidence; is a member of the Honeynet Project; is a referee for the Journal of Digital Investigation; and on the committees of several conferences, workshops, and technical working groups.