**CERIAS Tech Report 2006-57**

# Categories of Digital Investigation Analysis Techniques Based on the Computer History Model

by Brian D. Carrier, Eugene H. Spafford

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

DIGITAL INVESTIGATION 3S (2006) S121–S130

available at www.sciencedirect.com

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

Digital Investigation

# Categories of digital investigation analysis techniques based on the computer history model

*Brian D. Carrier**, *Eugene H. Spafford*

*Center for Education and Research in Information Assurance and Security – CERIAS, Purdue University, West Lafayette, IN 47907, USA*

*Keywords:*
Digital investigations
Digital evidence
Digital investigation frameworks
Computer history model
Event reconstruction

ABSTRACT

Several digital forensic frameworks have been proposed, yet no conclusions have been reached about which are more appropriate. This is partly because each framework may work well for different types of investigations, but it hasn't been shown if any are sufficient for all types of investigations. To address this problem, this work uses a model based on the history of a computer to define categories and classes of analysis techniques. The model is more lower-level than existing frameworks and the categories and classes of analysis techniques that are defined support the existing higher-level frameworks. Therefore, they can be used to more clearly compare the frameworks. Proofs can be given to show the completeness of the analysis techniques and therefore the completeness of the frameworks can also be addressed.

© 2006 DFRWS. Published by Elsevier Ltd. All rights reserved.

## 1. Introduction

Digital investigations have been conducted in various forms for many years, with several different methodological frameworks proposed. Many frameworks have been presented at past Digital Forensic Research Workshops (DFRWS). The Research Roadmap (Palmer, 2001) from DFRWS 2001 outlined a five-phase framework and DFRWS 2004 had three presentations on the topic (Beebe and Clark, 2004; Baryamureeba and Tushabe, 2004; Carrier and Spafford, 2004). To date, no conclusions have been made about which framework is "best" or even most "correct."

To advance the field of digital investigations to one grounded in science requires the development of comprehensive models and theory that can be validated using scientific methods. We propose the following, first explicated in Carrier (2006), as one such model with a sound theoretical foundation and that supports existing frameworks. This should serve to bring a formal basis to further research, as well as satisfy external requirements for scientific rigor (e.g., Daubert requirements).

In the past, the framework proposals have been high-level process models that outline phases of the process. Unfortunately, there is typically not a single process that must be followed when conducting an investigation. Multiple investigators may do different things at different times and all may reach the same conclusions. Further, there are different processes for a criminal investigation versus a non-criminal investigation, and different processes for a contraband investigation versus a computer intrusion investigation. Therefore, it has been difficult to show that one of the previously proposed frameworks is incorrect because its phases could be appropriate for some circumstances.

It was the authors' belief that the proposed frameworks and process models had underlying concepts in common. The goal of this work was to identify the common concepts of a digital investigation by focusing on the subject being investigated, which is a computer. A computation model based

* *Corresponding author.*
E-mail addresses: carrier@digital-evidence.org (B.D. Carrier), spaf@cerias.purdue.edu (E.H. Spafford).
1742-2876/$ – see front matter © 2006 DFRWS. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2006.06.011

on a finite state machine (FSM) and the history of a computer was used to define digital evidence and digital investigation concepts. One result of this work is the ability to define categories and classes of analysis techniques. Proofs can be given to show that the categories are sufficient. It can also be shown that the previously proposed frameworks have these categories and classes in common and the differences among the frameworks are the phases in which the categories and classes exist.

Section 2 of this paper provides an overview of the computer history model and Section 3 describes the categories and classes of analysis techniques. Section 4 compares the existing frameworks using these categories and Section 5 concludes.

## 2. Computer history model

To define categories of analysis techniques based on something beyond only our investigation experiences and biases, a computer model was used. The *computer history model* is based on the concept that every computer has a history, which is the sequence of states and events that occurred between two times (Carrier, 2006). In practice, the history of a computer is not fully recorded or known.

This work uses the definition that a *digital investigation* is a process that formulates and tests hypotheses to answer questions about digital events or the state of digital data. Questions include "does file *X* exist?" and "did event *Y* occur?". If the investigator knew the complete history of a system, then she could use it to answer any question about the system. In practice, the investigator may know only a current or previous state and need to make inferences about unknown states and events.

We propose that the techniques that can be used to make inferences about the history of a computer are the same that could be used during a digital investigation. Each investigation will require different techniques based on what questions are being asked and what devices were used. The following two sections outline the computer history model in more detail.

### 2.1. *Primitive computer history model*

The computer history model makes the assumption that the computer being investigated can be reduced to a finite state machine (FSM) quintuple $(Q, \Sigma, \delta, s0, F)$. $Q$ is a finite set of states and $\Sigma$ is a finite alphabet of event symbols. The transition function $\delta : Q \times \Sigma \rightarrow Q$ is the mapping between states in $Q$ for each event symbol in $\Sigma$. The machine state changes only as a result of an event. The starting state of the machine is $s0 \in Q$ and the final states are $F \subseteq Q$ (Sudkamp, 1997). It should be stressed that this work does not assume that a computer will be reduced to an FSM during an investigation. The FSM is used only as a theoretical model so that this work is not tied to a specific platform or design.

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. A graphical representation of states and events can be seen in Fig. 1, which shows an event $E_1$ reading the values from storage locations R3 and R6 and writing to locations R3 and R4. In this work, network cables are treated as short-term storage devices and network devices are treated as computers.

If the system can be represented by an FSM, then it is trivial to formally represent the primitive history. We define a set $T$ that contains the times that the system has a history. The duration between each time in $T$, which we will call $\Delta t$, must be shorter than the fastest state change in the system. The primitive state history is represented by a function $h_{ps} : T \rightarrow Q$ that maps a time to the primitive state that existed at that time. The primitive event history is represented by a function $h_{pe} : T \rightarrow \Sigma$ that maps a time $t \in T$ to a primitive event if that event occurred between $t - \Delta t$ and $t + \Delta t$.

While it is trivial to represent the history of a theoretical computer as a mapping, it is not trivial to represent the history of a modern computer because the possible states and events of a modern computer can change at each time in $T$. For example, when a USB drive is plugged into a system, the total number of possible states increases, which means that the size of the FSM $Q$ set increases. To fully represent the primitive history of a system, the model must take the dynamic nature of the system into account.

The computer history model uses a dynamic FSM model with sets and functions to represent the changing system capabilities. In the dynamic FSM model, the $Q$, $\Sigma$, and $\delta$ sets and functions can change for each time in $T$. The set $Q$ is defined at each time $t \in T$ based on what storage devices are connected, the number of storage locations in each device, and the domain of each storage location. The set $\Sigma$ and function $\delta$ are defined at each time $t \in T$ based on what event devices are connected, the possible events that each supports, and the state change function for each event.

As shown in Carrier (2006), knowing when each device was connected and the capabilities of each device will allow $Q$, $\Sigma$, and $\delta$ to be defined for each time in $T$. Table 1 lists the names of the 11 sets and functions that are used in the primitive

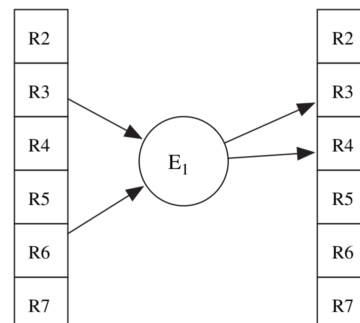


**Fig. 1 – Graphical representation of an event $E_1$.**

**Table 1 – Primitive computer history model variable names**

| Variable | Description |
|---|---|
| $T$ | Set of times that system existed |
| $D_{ps}$ | Set of primitive storage devices |
| $DAD_{ps}$ | Set of storage locations per device |
| $ADO_{ps}$ | Set of possible values per location |
| $c_{ps}$ | Maps a time to the set of storage devices that were connected |
| $h_{ps}$ | Maps a time to the state that existed |
| $D_{pe}$ | Set of primitive event devices |
| $DSY_{pe}$ | Set of event symbols for each event device |
| $DCG_{pe}$ | Set of state change functions per device |
| $c_{pe}$ | Maps a time to the set of event devices that were connected |
| $h_{pe}$ | Maps a time to the event that occurred |

computer history model. Using this information, we can define the primitive state and event histories as a mapping from a time in $T$ to a state and event in the dynamic $Q$ and $\Sigma$ sets. It can be shown that the 11 variables are necessary and sufficient to represent the primitive history of a computer.

### 2.2. *Complex computer history model*

The previous section outlined a model for low-level events and storage locations, but modern systems execute billions of instructions per second and store billions of bytes of data. Further, investigations are typically concerned with questions about complex events and states, such as the existence of a file or an event that downloaded a file. The primitive history model cannot directly represent complex states and events and therefore the complex computer history model builds upon the primitive model.

A *complex event* is a state transition that causes one or more lower-level complex or primitive events to occur. For example, the process of clicking on a button to save a file is a complex event because it is one event to the user and it causes hundreds or thousands of primitive events to actually occur. A complex event may have multiple representations. For example, we can also represent the complex event associated with clicking on a button as the sequence of programming language statements and functions that were used to implement the program and we can represent each programming language event with primitive-level events.

A *complex storage location* is a virtual storage location that is created by software. Complex storage locations are created by grouping together or dividing primitive-level storage locations. For example, a file is a complex storage location and the contents of a file are presented to the user and program as a single storage location, but it is really a group of sectors. Complex storage locations have one or more attributes, which are name and value pairs. For example, a file is a complex storage location and the name value attribute pairs include the file name, MAC times, permissions, and content. Complex storage locations exist only because software exists to create them. For example, a file can be read from and written to only if the operating system supports the file system type that was used to create the file.

A graphical representation of complex storage locations and events can be seen in Fig. 2. At a high-level, complex event $E1$ reads from complex storage locations $D1$ and $D2$ and writes a value to complex storage location $D1$. At a lower-level, complex event $E1$ is performed using lower-level events $E1$a and $E1$b. For example, these could be machine-level instructions. In addition, we can see that the complex storage locations use lower-level storage locations. The contents of $D1$ are stored in lower-level locations $D1$a and $D1$b and the contents of $D2$ are stored in lower-level locations $D2$a and $D2$b.

As previously mentioned, there are multiple levels of abstraction in a system and each can be represented by an FSM where $Q_X$ is the set of possible complex states at abstraction layer $X$, $\Sigma_X$ is the set of possible events at abstraction layer $X$, and $\delta_X$ is the state transition function at abstraction layer $X$. The possible complex states and events in a realistic system can change at each time, based on which applications and programs exist.

To define the dynamic FSM variables for the complex system at each time in $T$, we must enumerate the programs that existed and determine the complex events and storage locations they support. We must also define the transformation rules between the various abstraction layers. For example, there are transformation rules that define how lower-level data are abstracted to a higher-level, such as the rules that define which bytes of the FAT boot sector store the size of the file system and the location of the root directory. Transformation rules also exist to map between high- and low-level events.

The complex history maps a time in $T$ to a complex state in $Q_X$ and a complex event in $\Sigma_X$. Table 2 lists the names of the 17 sets and functions that exist in the complex computer history model.

## 3. Categories of analysis techniques

As described in the previous section, the primitive computer history model has 11 variables and the complex computer history model has 17 variables. For example, one variable in the primitive history model defines the number of addresses in each of the storage devices and another defines when the storage devices were connected.

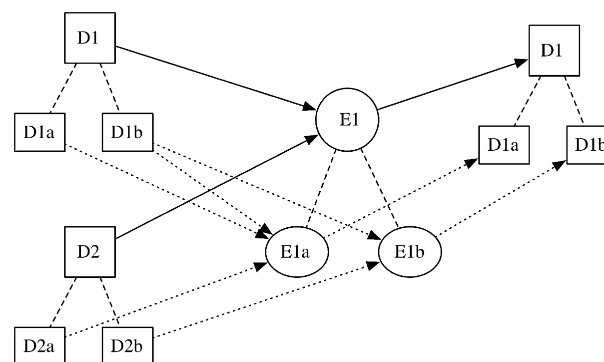


**Fig. 2 – A complex event E1 with two complex cause locations and one complex effect location.**

**Table 2 – Complex computer history model variables**

| Variable | Description |
|---|---|
| $T$ | Set of times that system existed |
| $L$ | Set of abstraction layers used by system |
| $D_{cs}$ | Set of complex storage types |
| $DAD_{cs}$ | Set of addresses per storage type |
| $DAT_{cs}$ | Set of attributes per storage type |
| $ADO_{cs}$ | Set of possible values for each attribute |
| $ABS_{cs}$ | Set of abstraction transformation functions |
| $MAT_{cs}$ | Set of materialization transformation functions |
| $c_{cs-X}$ | Maps a time to the complex storage types that existed at layer X |
| $h_{cs}$ | Maps a time to a complex state at that time |
| $D_{ce}$ | Set of programs on the system |
| $DSY_{ce-X}$ | Set of complex events per program |
| $DCG_{ce-X}$ | Set of state change functions per program |
| $ABS_{ce}$ | Set of abstraction transformation functions |
| $MAT_{ce}$ | Set of materialization transformation functions |
| $c_{ce}$ | Maps a time to the programs that existed |
| $h_{ce}$ | Maps a time to the slice of complex events that occurred |

Digital investigation questions about a state or event can be answered using the computer history and we use the history model variables to define the categories of analysis techniques. An investigation defines, at a minimum, the variables needed to answer the questions that started the investigation. For example, to answer questions about the existence of a file in the current state, we must define the primitive storage device capabilities and the current state. Next, the programs that exist in the current state must be identified and analyzed to determine the file system types that are supported. Lastly, the primitive state is transformed into a complex state to define the virtual storage locations. We refer to the history that is defined during an investigation as the *inferred history*.

The classifications of the analysis techniques in this work are subjective. We propose seven categories that are further organized into 31 unique classes, but this is not the only classification. It can be shown that the analysis techniques in the classification are capable of defining all variables in the model. The next section describes the general process to define a variable and the next seven sections describe the categories of analysis techniques.

### 3.1. *General investigation process*

The process of a digital investigation will depend on the questions being asked and therefore this work does not assume a specific sequence of high-level phases. However, this work proposes that the process of defining each of the variables in the computer history is a hypothesis formulation process and that the scientific method can be used to come to objective conclusions. The scientific method has four general phases: Observation, Hypothesis Formulation, Prediction, and Testing and Searching. This section provides a brief overview of the process and examples of how it would be used.

In the *Observation* phase, an investigator or program makes observations about states and events. Examples include observing the output of a tool that lists running processes or e-mail contents. The Observation phase in a digital investigation is equivalent to an investigator looking at a physical crime scene, except that in the digital world most observations are indirect. Indirect observations occur when something else is used to make the observation and observations of most digital data require hardware and software. In general, we trust direct observations and do not need to test them. Indirect observations will be trusted based on our trust and confidence in the things, such as tools, that were used to make the observation.

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 history model. Example hypotheses may be about the storage capabilities of the system, the primitive state of a hard disk, or the occurrence of a complex event.

Formulating hypotheses is a difficult part of the investigation process and the categories of analysis techniques that are presented in the following sections are based on the different types of hypotheses that are used. To be a scientific process, the hypotheses must be capable of being refuted (Popper, 1998).

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. Some common types of predictions are based on the existence of data, the results from running a simulation, and 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 from the Internet, 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.

The last phase of the process is the *Testing* phase. 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 include files whose SHA-1 hash is not in a hash database, files whose attributes are set to hidden, and 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 a hypothesis is neither

supported nor refuted, then an assumption can be made that it is true, but the investigator must be capable of justifying the assumption.

### 3.2. *History Duration*

We now describe the seven categories of analysis techniques proposed in this work. The first is the *History Duration* Category and it is used to define the contents of the set $T$, which contains the times that the system being investigated has a history.

There is a single class of post-mortem analysis techniques for in this category and it is *operational reconstruction*, which uses event reconstruction and temporal data from the storage devices to determine when events occurred and when the system was active. For example, a log file that records the OS installation date could be used to define when the system was first used and file times could also provide additional information. When time values from logs and files are used to formulate these hypotheses, the prediction and testing phases should attempt to refute the values by searching for signs of tampering. In practice, these techniques are used to show that the computer was used during the times being investigated.

### 3.3. *Primitive Storage System Configuration*

The second category is the *Primitive Storage System Configuration* Category and it contains analysis techniques to define the capabilities of the primitive storage system. The techniques in this category define the names of the storage devices ($D_{ps}$ set), the number of addresses for each storage device ($DAD_{ps}$ set), the domain of each address on each storage device ($ADO_{ps}$ set), and when each storage device was connected ($c_{ps}$ function). These sets and functions define the FSM set $Q$ of possible states.

We can organize this category into two types of hypotheses. The first defines the storage capabilities of each device and the second defines when the devices were connected.

To define the storage capabilities, there are two classes of techniques. *Storage Device Capability Observation* uses direct observations of the device labels and technical specifications to determine the device capabilities. The second class is *Storage Device Capability Query*, which uses a program to query the device for its capability information. For example, a program could query a hard disk to determine the number of sectors and size of each. In practice, these techniques are used when a copy of the hard disk is made. As with all of the values that are defined in the model, the results from the observation or query should be tested because the device could be incorrectly labeled or the device could have been modified to return false data.

There are also two classes of techniques to define when devices were connected. The *Storage Device Connection Observation* class of techniques uses direct observations of which devices were connected to define the current set of connections. The *Storage Device Connection Reconstruction* class of techniques uses log files and other data to reconstruct when each device was connected. For example, there could be a system log file that records when a removable storage device was inserted and removed. In practice, these techniques are performed for removable media, but not typically for hard disks.

### 3.4. *Primitive Event System Configuration*

The third category is the *Primitive Event System Configuration* Category and it defines the capabilities of the primitive event system. The techniques in this category are used to define the names of the event devices connected ($D_{pe}$ set), the event symbols for each event device ($DSY_{pe}$ set), the state change function for each event device ($DCG_{pe}$ set), and when each event device was connected ($c_{pe}$ function). These sets and functions are used to formally define the FSM set $\Sigma$ of event symbols and state change function $\delta$. In practice, because questions are not typically asked about primitive event devices, these analysis techniques are not performed.

We can organize this category of techniques into two types of hypotheses. One defines the name, event symbols, and state change function for each event device and the second defines when each event device was connected to the system.

There are two classes of techniques to formulate hypotheses about the capabilities of the event devices. *Event Device Capability Observation* uses direct observations about the make and model of event devices. *Event Device Capability Query* uses a program to query the event device and determine its capabilities.

There are also two classes of techniques to formulate hypotheses about when devices were connected. The *Event Device Connection Observation* class of techniques formulates and tests hypotheses based on direct observations of which devices were connected. The *Event Device Connection Reconstruction* class of techniques uses logs and other data to reconstruct when devices were connected to the system.

### 3.5. *Primitive state and event definition*

The previous two categories defined the capabilities of the primitive system and now we consider the techniques to define which events and states are believed to have occurred. The *Primitive State and Event Definition* Category defines the primitive state history ($h_{ps}$ function) and event history ($h_{es}$ function). There are five classes of analysis techniques that can be used to formulate and test this type of hypothesis and each class has a directional component, as can be seen in Fig. 3. 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.

The *Primitive State Observation* class of techniques uses a direct observation of an output device to define its state in the inferred history. Note that this technique cannot be used to define the state of most internal storage devices, such as a hard disk, because an investigator must use software or hardware to observe those devices. Direct observations can be used to define the state of a monitor controller or other output device controller.

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 may arbitrarily choose a possible state or

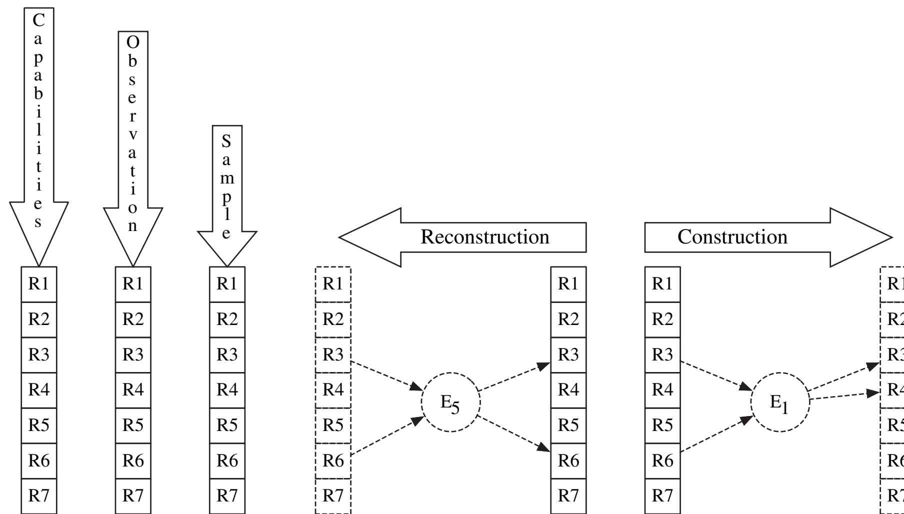


**Fig. 3 – The five classes of analysis techniques for defining primitive states and events.**

event. This approach is not efficient for real systems, which have a large number of possible states and events. Note that if a hypothesis is formulated with this technique, it will still need to be tested and evidence will need to exist to support the occurrence of the state or event. This class of techniques can also be used to formally test a hypothesis about states and events if the hypothesis has values that are beyond the known system capabilities.

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 include 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 randomly choosing a state). Testing will reveal if there is evidence to support the state or event. This class of techniques is not formally used in practice because sample data do not exist and their usefulness is not clear at the primitive level.

The *Primitive State and Event Reconstruction* class of techniques uses a known state to formulate and test hypotheses about the event and state that existed immediately prior to the known state. This is not formally performed in practice because questions are not typically asked about primitive events.

Informally, primitive-level reconstruction is performed in practice when the clone of a disk is analyzed. A clone of a disk is created to preserve its state and is created by copying the contents of each sector to 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, as shown in Fig. 4.

Now consider that at time $t'$ ($t' > t$) the investigator wants to use the clone to define the state of the system at time $t$. At time $t'$, the investigator knows only the current state of the clone. 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. In practice, this is typically achieved by calculating 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. A similar reconstruction process happens at the complex level when using disk image files.

The *Primitive State and Event Construction* class of techniques is the forwards technique and uses a known state to formulate and test hypotheses about the next event and state. In practice, this class of techniques would not be useful because we typically start with an ending state of the system and not a starting state. Further, hypotheses about events when the ending state is not known cannot be refuted and therefore any event and ending state is possible.

### 3.6. *Complex Storage System Configuration*

We now define three categories for the complex system. 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 techniques in this category define the names of the complex storage types ($D_{cs}$), the attribute names for each complex storage type ($DAT_{cs}$), the domain of each attribute ($ADO_{cs}$), the set of identifiers for the possible instances of each complex storage type ($DAD_{cs}$), the abstraction transformation functions for each complex storage type ($ABS_{cs}$), the materialization

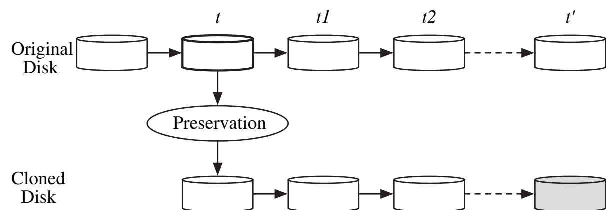


**Fig. 4 – A cloned disk has its own history and reconstruction must be performed to define the state of the original disk.**

transformation functions for each complex storage type ($MAT_{cs}$), and the complex storage types that existed at each time and at each abstraction layer $X \in L$ ($c_{cs-X}$).

There are two types of hypotheses that are formulated in this category. The first defines the names of the complex storage types and when they existed. The second defines the attributes, domains, and transformation functions for each complex storage type.

Complex storage locations exist only because programs exist to create them. Therefore, to enumerate the complex storage types that existed at a given time we must reconstruct the state of the computer to that time so that the programs can be identified and analyzed. There are two classes of techniques that are used to identify what programs existed. The first is *Program Identification* and it searches for programs on the system, which are then analyzed. The second class is *Data Type Observation* and it makes inferences about complex storage types that existed based on the data types that are found. This technique may identify complex storage types that were not supported by the system though, if for example the data were created on another computer and transferred to the computer being investigated.

The second type of hypothesis in this category defines the attributes, domains, and transformation functions for each complex storage type. Three classes of techniques are presented here. One is *Complex Storage Specification Observation* and it uses a specification to define the details about the complex storage types that the program supports. The second is *Complex Storage Reverse Engineering* and it uses design recovery reverse engineering techniques (Chikofsky and Cross II, 1990) to analyze the data that are transformed to define complex storage locations. The third is *Complex Storage Program Analysis* and it 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.

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 when the corresponding data type is identified. Therefore this category is typically performed by tool vendors prior to an investigation.

### 3.7. *Complex Event System Configuration*

The *Complex Event System Configuration* Category of analysis techniques defines the complex event system capabilities. The techniques in this category define the names of the programs that existed on the system ($D_{ce}$), the names of the abstraction layers ($L$), the symbols for the complex events in each program ($DSY_{ce-X}$), the state change functions for the complex events ($DCG_{ce-X}$), the abstraction transformation functions ($ABS_{ce}$), the materialization transformation functions ($MAT_{ce}$), and the set of programs that existed at each time ($c_{ce}$).

Making inferences about complex events can be even more challenging than about complex storage locations because complex storage locations are designed to be both 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 transformation rules can be more easily changed 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 in this category. The first defines which programs existed and when they existed. The second defines the abstraction layers, event symbols, and state change functions for each program. The third defines the materialization and abstraction transformation functions between the abstraction layers.

There are two classes of analysis techniques that can be used to determine which programs existed and when they existed. These are *Program Identification* and *Data Type Reconstruction*. Both of these were described in the previous section.

The complex events in each program can be determined using two classes of analysis techniques. The first is *Complex Event Specification Observation* and it uses a specification of the program to determine the complex events that it could cause. The second is *Complex Event Program Analysis* and it analyzes the program itself. At a basic level, this is performed by the investigator when he runs a program to determine its capabilities. At a more detailed level, this is a difficult task and requires extensive reverse engineering knowledge.

The third type of hypothesis in this category is about the abstraction and materialization transformation rules. These rules define the mappings between the higher-level events and the lower-level events. These rules are difficult to learn and only one class of techniques is proposed. The *Development Tool and Process Analysis* class of techniques analyzes the programming tools and development process to determine how complex events are defined. In practice, this is not performed.

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 how the software engineers decided to design and implement the program and therefore there may not be exact rules. The problem of abstracting events is not unique to digital investigations and the software engineering community has been researching program understanding techniques to help developers abstract legacy code. For example, Cifuentes and Gough (1995) and Ward (2000) describe approaches to abstract primitive-level events to programming language-level events using flow graphs and intermediate representations.

### 3.8. *Complex state and event definition*

The final category of analysis techniques defines the complex states that existed ($h_{cs}$) and the complex events that occurred ($h_{ce}$). There are eight classes of analysis techniques in this category and each has a directional component, as can be seen in Fig. 5. Two of the techniques can be used to formulate and test hypotheses about individual states and events. Two are forwards- and backwards-based and the final four are upwards- and downwards-based.

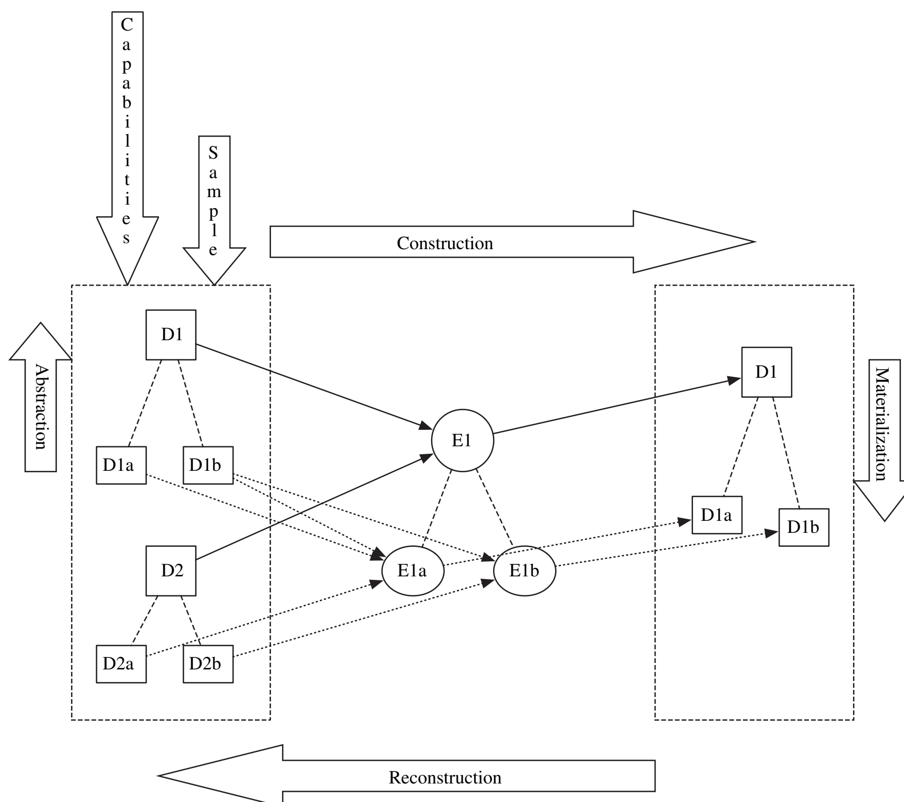


**Fig. 5 – The classes of analysis techniques for defining complex states and events have directional components to them.**

The *Complex State and Event System Capabilities* class of techniques uses the capabilities of the complex system to formulate and test state and event hypotheses based on what is possible. Therefore, a hypothesis may include a state or event at random. This class of techniques is not used in practice for formulating hypotheses, but can be used to show that another hypothesis is impossible because it is outside of the system's capabilities.

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 include metrics on the occurrence of events and states and would show which states and events are most likely. This class of techniques informally occurs in practice when investigators formulate hypotheses about the existence of files and data types based on the types of systems or the type of incident being investigated. For example, if a PC is found then hypotheses are formulated about the most common types of operating and file systems. If the investigation is for an intrusion, then hypothesis will be formulated about the existence of rootkits and attack tools.

The *Complex State and Event Reconstruction* class of techniques uses a state to formulate and test hypotheses about the previous complex event and state. In practice, event and state reconstruction often occurs, but it does not reconstruct the event and state immediately prior to a known state. More commonly, the hypothesis describes a general class of event instead of a specific event and the starting state of the event is not given. Common examples of event reconstruction include analyzing a web browser history file to determine which web sites were visited and analyzing the Most Recently Used (MRU) keys in the Windows registry to determine previous files and commands. Another 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.

A less obvious example of data and event reconstruction 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 listed 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 were hidden or introduced before they were was displayed on the monitor.

The *Complex State and Event Construction* class of techniques uses a known complex state to formulate and test hypotheses about the next complex event and state. As with primitive-level construction techniques, complex-level construction techniques are not frequently used in practice to define the event and state immediately following a known state. Instead, the construction concept is used to predict what events may have occurred. For example, if an e-mail is found that instructs a user to perform events or if a program is found, then the instructions in the e-mail and program could be used to predict what events occurred.

The final four classes are based on abstracting lower-level data and events to higher-levels and on materializing higher-level data and events to lower-levels. The *Data Abstraction* class of analysis techniques is a bottom-up approach to defining complex storage locations using lower-level data and data abstraction transformation rules. Because many current investigations start with only disk sectors or network packets, this class of techniques is one of the first steps in the investigation process so that the investigator can search the complex state for evidence. A common example of this is the abstraction of disk sectors into a file system and files.

The *Data Materialization* class of techniques is the reverse of data abstraction and it defines primitive and complex storage locations using higher-level complex storage locations and data materialization rules. This class of techniques is not used much in practice because the lower-level data are typically known and, by design, the materialization process will produce either multiple possibilities of lower-level data or the data will have undefined values.

The *Event Abstraction* class of analysis techniques is the bottom-up approach to defining complex events using a sequence of lower-level events and event abstraction rules. 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.

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 and event materialization rules 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.

## 4. Application to existing frameworks

There are several applications to the computer history model and categories of analysis techniques. For example, it can be used to show how the scientific method is used in a digital investigation and it can be used to support existing process models and frameworks. In this section, we will show how this work supports three of the existing frameworks and process models. Only the analysis related phases of the models are considered.

The first model that we consider is the process model in the NIJ Electronic Crime Scene Investigation Guide (United States National Institute of Justice, 2001). This model 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 electronic devices are collected or copied. This phase would therefore contain the analysis techniques to define the primitive storage and event capabilities of the system and a known state.

In the Examination phase, the electronic evidence is made "visible" and the contents of the system are documented. Data reduction is performed to identify the evidence. In the Analysis phase, the evidence from the Examination phase is analyzed to determine its "significance and probative value."

Presumably, the Examination phase includes the techniques to define the complex state, which includes the file abstractions. After establishing the primitive and complex state, the investigator can begin to formulate and test hypotheses about the incident, which may use Complex State and Event Sample Data techniques to determine which events and states are most likely to exist. For example, searching for all JPEG files if they are commonly associated with this type of incident. Basic event reconstruction of deleted files may also occur.

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 will vary depending on the type of investigation. For example, in an intrusion investigation, this phase may perform event reconstruction on files identified in the Examination phase. While many of the classes of analysis techniques that were described in the previous section are not specifically addressed in this model, the phases are general enough that any technique can be performed in them. In general, there is not a clear difference between the technical requirements of the Examination and Analysis phases and any of the analysis techniques could occur in either phase.

The Event-Based Digital Forensic Investigation Framework has three phases: Preservation, Search, and Reconstruction (Carrier and Spafford, 2004). The Preservation phase preserves the state of the system and is where the primitive and complex system capability definition techniques would be used. The Search phase is where the evidence is identified, which would require that the remaining system capabilities and complex storage locations be defined. This phase will also require the Complex State and Event Sample Data techniques to formulate hypotheses based on experiences with similar incidents. Searches would be conducted to answer the investigation questions. The Reconstruction phase uses event reconstruction techniques to formulate hypotheses about the events that occurred based on the evidence that was found.

This model is specific about the analysis techniques that must occur in each phase, but it is not clear if, for example, the event construction techniques are supported by this framework. Event construction techniques are not useful in practice though they formulate hypotheses about an event occurring when there is no evidence to support the event. Therefore, it is not unreasonable for a process model to not specifically include this class of techniques. 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.

Lastly, we consider the Hierarchical, Objectives-Based Framework (Beebe and Clark, 2004). This model has two tiers of phases. In the first tier, the relevant phases are Incident Response, Data Collection, and Data Analysis. The Incident Response phase is where the incident is detected and verified. The detection and verification processes are equivalent to using event reconstruction techniques to test the hypothesis that the incident occurred because they look for evidence of

the incident. The Data Collection phase defines states of the system and system capabilities.

The Data Analysis phase is further organized into the Survey, Extract, and Examine phases. The Survey phase performs the complex storage location abstraction techniques and the Extract phase begins to identify data based on objectives such as keywords and filters. This is a process of formulating hypotheses about states based on experience and sample data. The Examine phase may include event reconstruction on the extracted data to determine if the data are relevant to the incident. The exact techniques of this phase will likely depend on the investigation type.

## 5. Conclusions

There is no single high-level process that every digital investigation must follow. We have shown a model with theoretical foundations and that supports existing frameworks. This work allows a formal basis to exist for further research and shows where the scientific method can be used in a digital investigation.

This work has outlined analysis techniques that may apply to all types of investigations and the completeness of the analysis techniques can be shown. As new process models and frameworks are proposed, the results from this work can be used to show how the new proposals are different from existing ones.

The class and category definitions in this paper still have a level of subjectivity to them based on how they are grouped. The unique contribution of this paper is that the categories are based on how digital evidence is created and not based on how existing tools operate or how existing investigations are conducted. This work has identified analysis techniques that are not supported by existing tools, although the practical value of some of the techniques is not entirely clear.

Carrier (2003) previously defined analysis tools based on abstraction layers. This work expands upon that to show that there are more than simply abstraction techniques. There are also techniques to, for example, learn the abstraction transformation rules, to determine the system capabilities, and to reconstruct previous states and events.

## REFERENCES


Baryamureeba Venansius, Tushabe Florence. The enhanced digital investigation process model. In: Proceedings of the 2004 digital forensic research workshop (DFRWS); 2004.

Beebe Nicole Lang, Clark Jan Guynes. A hierarchical, objectives-based framework for the digital investigation process. In: Proceedings of the 2004 digital forensic research workshop (DFRWS); 2004.

Carrier Brian. Defining digital forensic examination and analysis tools using abstraction layers. International Journal of Digital Evidence (IJDE) Winter 2003;1(4).

Carrier Brian D. A hypothesis-based approach to digital forensic investigations. PhD thesis, Purdue University; 2006.

Carrier Brian D, Spafford Eugene H. An event-based digital forensic investigation framework. In: Proceedings of the 2004 digital forensic research workshop (DFRWS); 2004.

Chikofsky Elliot J, Cross II James H. Reverse engineering and design recovery: a taxonomy. IEEE Software; January 1990.

Cifuentes Cristina, Gough John. Decompilation of binary programs. Software – practice and experience 1995;25(7).

Palmer Gary. A road map for digital forensic research. Technical Report DTR-T001–01, DFRWS; November 2001. Report from the first digital forensic research workshop (DFRWS).

Popper Sir Karl. Science: conjectures and refutations. In: Klemke ED, Hollinger Robert, Wyss Rudge David, David Kline A, editors. Introductory readings in the philosophy of science. 3rd ed. Prometheus Books; 1998. p. 38–47.

Sudkamp Thomas A. Languages and machines: an introduction to the theory of computer science. 2nd ed. Addison Wesley; 1997.

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.

Ward MP. Reverse engineering from assembler to formal specifications via program understanding. In: WCRE'00: Proceedings of the seventh working conference on reverse engineering (WCRE'00). Washington DC, USA: IEEE Computer Society; 2000. p. 11.