Design and Implementation of Zeitline: a Forensic Timeline Editor

Florian Buchholz and Courtney Falk
{florian,court}@cerias.purdue.edu
CERIAS, Purdue University
656 Oval Drive, West Lafayette, IN, 47901, USA

Keywords: Event Reconstruction; Digital Forensics; Computer Forensics; Audit Data, Graphical User Interface

ABSTRACT
In this paper we describe the design and implementation of Zeitline. Zeitline is a graphical timeline editor that allows a forensic investigator to create a timeline of events that were gathered from different sources, such as host MAC times, system logs, and firewalls. We present some background information, discuss the design of the tool, describe its features, and give an overview of how to improve the existing prototype.

1. INTRODUCTION
Many tools exist that aid a computer forensic investigator in analyzing storage media and the overlying file systems. Among those are Encase [5], the Sleuth Kit [4], and the Forensic Toolkit [1]. They focus mainly on evidence recovery, i. e. recovering deleted or hidden data. Beyond simple searching and indexing functionality, these tools generally have limited abilities to further analyze the data that is recovered.

Searching for keywords, file types, or file hashes might be sufficient when trying to locate incriminating material on a system, but is insufficient when trying to reconstruct events that have taken place on a system. Zeitline allows a forensic investigator to construct and view a time line of events based on the information found the system under investigation. The timeline will consist of events that come from different sources such as file timestamps (MACs), system and application logs, IDS and firewall logs. When examining evidence from different sources of a computing system (e.g. system logs and MAC times), currently an investigator needs to analyze the output from those sources separately and make notes to correlate events. Some tools exist that can incorporate events from multiple sources into a single list of time-sorted entries (see Section 2), but only when special software was installed on the systems to be monitored. Zeitline is the only tool we are aware of that lets a user import events from arbitrary sources and order them according to time values without the need for setting up special monitoring. Furthermore, it is the only tool that lets a user generate a hierarchy of events: starting with events at discrete times that were generated from the sources, events that belong to the same "action" or type can be grouped together into event hierarchies. For example, the three events "access program gcc", "access file x" and "access library y" could be grouped together into a super event labeled "compile program x", which in turn could be part of another super event "install rootkit z".

Zeitline is an open-sourced graphical tool [3] written in Java that allows a forensic investigator to import various events from a computing system and then order and classify them into one or more timelines of events. Events may be grouped together into super-events, creating a hierarchy of events. The organization of events and timelines as tree views allows the investigator to display and hide specific events, which makes it easy to focus on the relevant aspects of the investigation one at a time. This is further supported by the ability to filter events based on keywords as well as start and ending times. Furthermore, events serve as a unified data structure to bring together forensic evidence from different data sources. It is now possible to combine data from those sources and analyze them within a single framework.

The design objectives for Zeitline are as follows:

- Generating events from arbitrary data sources
- Grouping events together into logical groups recursively
- Filtering of the data that is displayed
- Locating specific events
- Intuitive interface
- Platform-independent implementation

2. BACKGROUND AND RELATED WORK
The main functionality of the Encase tool [5] from Guidance Software lies in the retrieval of data from a system and providing the means to locate specific data easily. Encase provides the functionality to sort file information by various fields, which includes timestamps. It is also possible to

retrieve and search within log files from a system, such as system logs, log data from security software and application logs [6]. However, there is no opportunity to combine data from different sources and establish any kind of temporal or logical ordering among them.

Access Data's Forensic Toolkit (FTK) [1] focuses primarily in securing information off a computing system and then provides the ability to locate and examine specific files. Files can be sorted by their attributes, including the timestamps. There are extensive search capabilities as well as a large number of known file formats whose contents can be displayed and searched.

Brian Carrier's Sleuth Kit [4] also has the ability to view file system events. The focus, however, lies primarily in the recovery of the information as opposed to its analysis. Basic timelines of the file system events are generated with the `mactime` tool, which generates a sorted list of MAC timestamp events. One way an investigator is able to organize events with the Autopsy forensic browser is the ability to generate annotated bookmarks for events. However, it is not possible to group events in a hierarchy or perform detailed searches on them.

The FileList Pro tool from New Technologies [9] can create timelines of file activity on a system. A user can choose to sort the files and the information associated with them by various timestamps. Timelines can be created for file access, creation, modification, and deletion for DOS and Windows systems. However, a user is not able to use the tool to introduce events from other sources, or group events together.

The nFX open Security Platform from netForensics [8] provides a mechanism to gather event information from various sources of a network. The vents are normalized and synchronized and displayed real-time for intrusion detection purposes. Furthermore, statistical analysis techniques may be utilized for event correlation. The events are gathered via agents that need to be installed on the systems from which to gather data. There is a large number of devices supported directly by the product, and a "Quick Connect" feature provides the ability for custom agents for other data sources.

Given that agents need to be installed and active on all systems that are monitored, the usefulness for nFX for forensic purposes is limited. When the platform is deployed on a system that needs to be investigated it may well be used for a forensic investigation. However, it is not possible to group events into hierarchies to graphically build a timeline of events. Also, only those events are captured that are provided through some kind of logging or reporting mechanism by the system or application. Events from MAC times for example cannot be captured in this fashion.

3. THE DESIGN OF ZEITLINE

Zeitline presented several design challenges to the developers. Among those were fast and efficient data structures, intuitive user interface design, and compliance with the basic requirements of digital forensics tools.

3.1 The Event Data Structure

Choosing an appropriate data structure was the first hurdle faced by the developers. The Java Development Kit (JDK) in version 1.4 offers several data structure classes. As the anticipated size of the data set was in the hundreds of thousands of events, the ability for a rapid search and retrieval of data items is of utmost importance. Also, the project assumed that events could be added to the data structure at up to ten thousand or one hundred thousand at a time. This necessitated a quick build time of the data structure. Given that that events needed to be sorted, it was decided that any data structure would require a run time of $O(\log(n))$ for any of its operation (lookup, add, remove), which would result in an overall build-time of $O(n \log n)$.

JDK 1.4 offers a `TreeSet` class, which is an implementation of a balanced search tree. However, due to the mappings required for a custom `TreeModel` class for the `JTree` GUI component, one of the lookup operations needed could only be run in $O(n)$ time, which resulted in a build-time of $O(n^2)$. Thus we decided to implement a custom data structure that utilizes Adelson-Velskii and Landis (AVL) trees [2], a variant of a balanced binary search tree.

The main data structure of Zeitline is the *event*. There are two types of events in Zeitline: *atomic events* are the events that are directly imported from the system, i.e. file MAC times, logs, etc. *Complex events* are events that are comprised of atomic events or other complex events, that is they act as a container for other events from which it derives some of its properties. There is an abstract class `TimeEvent`, which defines common fields and methods of the two kinds of events, such as the start time, name, description, and parent event fields and methods to retrieve them. The class `AtomicEvent` further adds a reference to the source of the event. The source contains information about from where the event was imported and what the time granularity is. The class `ComplexEvent` adds fields for an end time and its child events. The start time of a complex if event is defined as the smallest start time among its children, while the end time is the largest end time of the children (for atomic events, the end time can be considered the same as the start time). That is, whenever child events are added or removed from the complex event its start and end times potentially change. Note that a complex event does not have a source associated with it. Instead, the sources of its children define its "source".

The child events of a complex event are organized in a balanced binary AVL search tree. The sorting key for the children is their start time combined with a unique identifier so that the events are sorted by their start time. Events that have the same start time and are siblings are sorted in an arbitrary fashion (actually, an event that was created earlier is considered "smaller" because its unique identifier will be smaller. Complex events that are children of a complex event in turn may contain children, which are also organized in AVL tree, and so on. Figure 1 depicts the structure of a complex event.

Thus a timeline is nothing but a complex event as a root that contains the hierarchy of events with its children. Zeitline uses a subclass of the `java.swingx.JTree` class to dis-

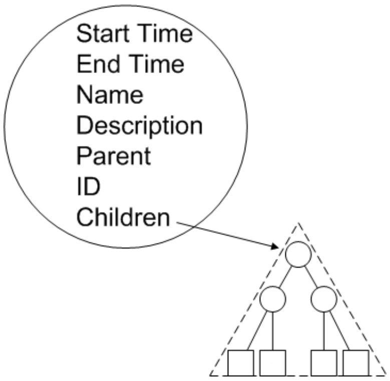

Figure 1: The structure of a complex event

play timelines in a tree view. Complex events are expandable/collapsible nodes, whereas atomic events are the leaf nodes. This is analogous to a file systems browser with directories and files.

A complex event must at least contain one child, which means that at the lowest level of the hierarchy there must be an atomic event. The exception is an empty timeline, where the complex event serves as the root node of the tree view.

3.2 Importing Events

One of the essential features of Zeitline is its ability to generate events from any kind of data source. This is similar to the "Quick Connect" feature of the nFX Open Security Platform, but given that we do not attempt to gather data in real-time, we have the ability to gather more types of data, such as file MAC times or other information that cannot be actively monitored by agents installed on a system. While currently only output from the the `fls` and `ils` tools are supported, Zeitline's import capabilities can be dynamically extended by supplying a Java class that implements our `InputFilter` interface and generates events from the desired data source. The fact that Zeitline dynamically loads the input filter classes at start-up means that a user can extend the functionality without having to re-compile the Zeitline classes. It is sufficient to compile the new input filter class and put it into a special directory.

Because the ability to create your own input filters to generate events is one of the most important features of Zeitline, we discuss the Java `InputFilter` class in great detail here and show through the example of the `FLSInputFilter` how to implement an input filter. The interface requires the following methods:

```
public abstract class InputFilter {
    public abstract Source init(String location,
                                Component parent);
    public abstract AtomicEvent getNextEvent();
    public abstract FileFilter getFileFilter();
    public abstract String getName();
    public abstract String getDescription();
    public abstract long getExactCount();
    public abstract long getTotalCount();
    public abstract long getProcessedCount();
}
```

Classes that implement the `InputFilter` interface must provide an implementation for the required methods. `init()` will initialize the data source (e.g. open a file and gather information about the source) and return an object describing the source. `getNextEvent()` will return the next event that comes from an initialized source. If there is no next event, then `null` is returned. This allows loops such as:

```
while ((event = filter.getNextEvent()) != null){
  /* process the event */
}
```

The `getFileFilter` method tells the tool how (and if) to filter files or file names when the file chooser dialog is opened (e.g. "*.txt" for text files). It may return `null` if no such filtering is desired.

The `getName()` and `getDescription` methods return a name and a description for the filter, respectively. The name will appear in the filter type selection when importing, while the description will be used in future versions to give the user more feedback when selecting a filter.

The three methods `getExactCount()`, `getTotalCount()`, and `getProcessedCount()` are used for progress bar updates. If the filter knows how many events will be generated, then it returns that number as the exact count. Otherwise, the exact count should return 0 and the total count can be returned, which represents the amount of data that is processed (number of lines or bytes, for example). The method `getProcessedCount()` will then return the amount of data (of the total count) that has already been processed by the filter during the import process.

In the following we explain how our `FLSInputFilter` class works, which may serve as a proof-of-concept for other filter classes. The `FLSInputFilter` class processes data that was output by the `fls` tool, which is part of the Sleuth Kit [4]. We require the `fls` output to be in machine format, i.e. with the `-m` flag enabled.

The `init` method is fairly straightforward. We attempt to open the input stream specified by the file name and on success we return a new `Source` object or `null` otherwise:

```
public Source init(String filename) {
    try {
        file_input = new BufferedReader(
          new FileReader(filename));
    }
    catch (IOException ioe) {
        return null;
    }
    return new Source("FLS filter", filename,
      Source.GRANULARITY_SEC);
}
```

Each line of the `fls` output can create between one and three separate events, depending on whether the timestamp are the same or differ. For this, we have a FIFO queue called `event_queue` in which we can place the extra events resulting from the processing of the line to be read in subsequent `getNextEvent()` calls. Thus the algorithm for processing lines and returning events is as follows:

1. If `event_queue` is empty, read the next line from the input. Else dequeue and return the next event.
2. Return `null` is end of file is reached.
3. Process the line and compare the timestamps.
4. If more than one event is created, queue all but one.
5. Return the remaining event.

The following is pseudo-Java code of the `getNextEvent()` method, glossing over unimportant parts:

```
public AtomicEvent getNextEvent() {

  if (event_queue.isEmpty()) {

    read the line from the input stream
    if (line == null) return null;

    fields = line.split("\\|");

    // get timestamps, we have second granularity
    // but need to convert to ms

    long mtime =
      Long.decode(fields[12]).intValue()*1000;
    long atime =
      Long.decode(fields[11]).intValue()*1000;
    long ctime =
      Long.decode(fields[13]).intValue()*1000;

    String name = fields[1];

    String description = "User: " + fields[7] +
      "\n" + "Group: " + fields[8] + "\n" +
      "Mode: " + fields[5];

    if ((mtime == atime) && (mtime == ctime))
      return new AtomicEvent("MAC " + name,
        description, new Timestamp(mtime));

    if (mtime == atime) {
      event_queue.add(
        new AtomicEvent("MA. " + name,
          description, new Timestamp(mtime)));
      return new AtomicEvent("..C " + name,
        description, new Timestamp(ctime));
    }

    /* and so on for all the MAC combinations ...*/

  }
  else
    return (AtomicEvent) event_queue.removeFirst();
}
```

The progress bar methods are implemented as follows: because we do not have an exact count of the number of events that will be generated when we initialize the filter, `getExactCount()` returns 0. The total count is simply the size of the FLS file in bytes, and the processed count the byte position of the open file handle:

```
    public long getExactCount() {
        return 0;
    }

    public long getTotalCount() {
        return file_input.length();
    }

    public long getProcessedCount() {
        return file_input.getFilePointer();
    }
```

3.3 The Graphical User Interface

Java offers several options for producing GUIs. Among them are the Abstract Windows Toolkit (AWT) and more recent Swing classes. Briefly considered was the Eclipse project, whose Simple Window Toolkit (SWT) allows Java GUI functionality to be handled by OS-native APIs. Eventually Eclipse was discontinued from consideration because of its poor performance when building a `Tree` object. Swing proved to be the optimum choice for GUI design because of its native GUI objects such as the tree list, an object ideally suited for the hierarchical display of events such as those dealt with in Zeitline.

A conscious thought while designing the GUI was to make it as easily understandable as possible. Because the general audience of the program may include law enforcement professionals with little prior background in computers, the program should be as simple as possible to approach an understand. With this thinking in mind the GUI was constructed to imitate the functionality of other pervasive applications, such as Microsoft's Windows Explorer, so as to take advantage of the user's innate understanding of user interface functions.

Figure 2 shows Zeitline with two timelines facing each other. The timeline on the right was imported from the `fls` tool output from the HoneyNet Scan of the Month 15 [7]. The timeline on the right contains select events from that source that involve the installation of a rootkit. Atomic events are displayed with a single bullet icon, and complex events have a triangle of three small, differently colored bullets for an icon. The selected event in the left tree can be moved to any of the complex events via drag & drop or cut & paste. Furthermore, a new complex event or even a new timeline can be created, containing the selected event. We can also "delete" the event (see Section 4.2).

4. FEATURES OF ZEITLINE

The initial prototype of Zeitline that we describe in this paper concentrates on a small set of basic features that are important when constructing a timeline of events. They can be divided into three categories: managing events via the GUI, maintaining the integrity of digital evidence, and being able to quickly locate events. We will also give a discussion of future features for Zeitline in Section 5.

4.1 Managing events

Being able to manage events efficiently is the most important feature of Zeitline. After events from one source have been imported into a single timeline, a user has several options to group them into complex events. New complex events can only be created from a selection of other events (atomic

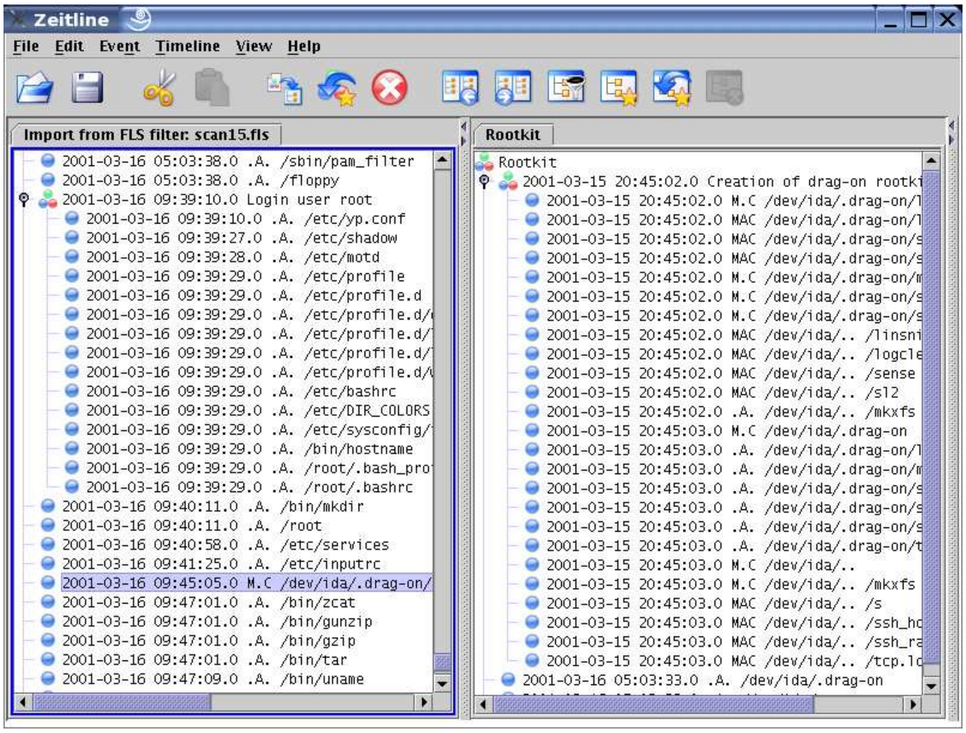

Figure 2: A screenshot of Zeitline

and/or complex): the selected events are moved into a newly created complex event, which, in turn, is inserted into the timeline at the parent of the node(s) highest in the hierarchy among the selected events. Once a complex event exists, events can be transfered to and from it via drag and drop. This can be within the same timeline or between two timelines. Furthermore, events may be transfered using cut and paste actions. Nodes in the tree that represent a complex event may be expanded or collapsed, allowing the user display and hide information as needed.

A user can also create empty timelines and then populate it with events, or he can create a new timeline from selected events of another one. This way, a user can make a hypothesis and then look for events supporting it while at the same time building the timeline for it. Timelines are displayed either within a single `JTabbedPane` or in two such panes next to each other. The arrangement of timelines in tabbed panes allows the user to easily switch view between timelines. The double-pane model lets the user view two timelines next to each other. This is especially beneficial when constructing a timeline of events from different sources: one side is used to construct the event hierarchy, and the other side to search for the relevant events supporting the timeline under construction. This mode is also ideal for moving events between timelines via drag and drop. The single-panel mode offers more viewing space for the tree view and may be helpful when grouping events within the same timeline.

4.2 Maintaining forensics integrity

The final challenge faced by the developers of Zeitline was that of conforming to the special needs of digital forensics software. Data utilized by forensic programs is often presented in courts of law. As a result of this usage the handling of data by the software is subject to certain restrictions to prevent manipulation, favoring one side or another. One specific example of this is how a single source of information such as output from the FLS tool provides hundreds or thousands of pieces of evidence. An unscrupulous investigator could remove from the data set any events that are not beneficial to his or her side's case. However, an investigator may also want to remove from view certain events that are distracting or cloud the overall picture.

An all-or-none approach towards sources and events was taken. Either the program would use every single event from a source file or none at all. An investigator would not be allowed to delete from the project any events. The solution to this was to create an unseen "orphan" timeline that contains any "deleted" events. The events themselves are not deleted from the project but merely shifted into the orphan timeline. In this way the events can be moved out of the field of view but without compromising the integrity of the source of evidence.

Cut and paste functionality also affects the source integrity of a data set. It is possible for an event to be cut and never pasted back into a timeline. If this were to happen then when the program closes the event stored in the cut buffer is lost. In order to avoid this, special care is taken when the program is saving to first dump the contents of the cut buffer into the orphan timeline before writing out the data.

4.3 Queries

Zeitline uses Query objects to match against events. Queries can then be used to filter and locate events. During filtering only those events are displayed that match the query, whereas when locating events the (first) event(s) that match the query are displayed in their current context.

Currently, queries only support a keyword search in combination with a time interval in which the events must occur. The keyword search allows the use of regular expressions as we use Java's `String.matches()` method to determine a match. The keyword is initially padded with wildcard (`.*`) matchers at the start and end, but the keyword itself can contain Java regular expression characters. This also means that certain characters have to be escaped for a literal match.

All the logic needed to perform query matches is contained in the `Query` class. More sophisticated types of queries may thus be easily added to Zeitline by either modifying the `Query` class, or by sub-classing it and overriding its `matches()` method. Some of the query features that are planned for future releases are: search by event source, end time, and type (once a typing system for events exists).

5. CONCLUSIONS AND FUTURE WORK

With the first release of Zeitline we have provided a tool that allows a forensic investigator to import events from different sources, and lets him group events together into complex events. This grouping may create a hierarchy of events from very detailed (individual file accesses, network logs, etc.) to general (system installation, system break-in, etc.). That way, an investigator may hypothesize about what events took place on a system, using the low-level events to support the hypothesis. At the same time, events are kept in chronological order, so that timelines of either the entire system or individual (complex) events can easily be created. Furthermore, events that can be identified to be part of the normal system behavior and moved to a proper complex event. This can drastically reduce the amount of information an investigator has to analyze repeatedly.

Zeitline is programmed in Java, which means that it will run on many platforms, including Windows, MacOS, and Linux. Adding new event import filter objects is a fairly simple task and independent from Zeitline's other functionality. Thus, it is simple to add support for more or future types of event inputs. Furthermore, Zeitline is open source, which means that it may be adjusted to individual needs and extended by third parties easily.

Zeitline performs reasonable fast, but the Java Swing components cause a significant overhead. Importing approximately 86,000 events on an AMD Athlon 900MHz processor machine with 1GB main memory takes 46.6 seconds, of which 27 seconds are used to draw the JTree widget from the complex event we created. Importing 14,630 events takes a total of 9.7s, of which 5.5s are used for the GUI operations. The responsiveness of the GUI also depends on the number of events that are present. Selecting 86,000 events within a single tree view takes about as much time as generating the events. The drag and drop performance is also affected by the number of events. For this reason we recommend that after importing a large number of events to break the large set down into smaller timelines that belong together either temporal or logical, as a first step. This will improve responsiveness of the GUI significantly. For future work we plan to introduce a feature that "swaps out" unimportant events to disk, reducing the amount of events that need to be kept in the GUI data structures and in main memory.

The first version of Zeitline has a limited set of what we consider its core functionality. We do see great potential for future features that may be very useful, as well. Some of these include:

- Performance improvements
  - Multi-threading for tasks such as filtering or importing events
  - Indexing various fields of events for faster searching and filtering
- Usability improvements
  - Right-click context menus for events and timeline tabs
  - Status bar and progress bars
  - Drag & drop functionality for timelines
- Have more information about the host available. This could include things such as user mappings and information about hard- and software.
- Support for events that do not have a time associated with them. For example, a user's command line history contains events that are ordered and can provide valuable information.
- Support for events from more than one host. Thing such as clock differences and clock drift and time zones will have to be considered for that. Zeitline can then be used to synchronize events from different sources.
- A type system for events. This can simplify the grouping of events and it enables new filter and search possibilities.
- The ability to export timelines, for example into some kind of XML format.

6. REFERENCES

[1] AccessData Corp. Forensic Toolit. `http://www.accessdata.com/Product04_Overview.htm`.

[2] G.M. Adelson-Velskii and E.M. Landis. An algorithm for the organization of information. *Dokladi Akademia Nauk SSSR*, 146(2):1259–1262, 1962.

[3] Florian Buchholz and Courtney Falk. Zeitline forensic timeline editor. `http://www.cerias.purdue.edu/homes/forensics/timeline.php`, April 2005.

[4] Brian Carrier. Sleuthkit and Autopsy forensic browser. `http://www.sleuthkit.org`.

[5] Guidance Software. Encase forensic software. `http://www.guidancesoftware.com`.

[6] Guidance Software. EnCase Enterprise Edition Detailed Product Description.
http://www.guidancesoftware.com/corporate/whitepapers/downloads/EEEDeta%iled.pdf, July 2004.

[7] HoneyNet Project. HoneyNet Scan 15.
http://www.honeynet.org/scans/scan15, May 2001.

[8] netForensics Inc. nFX Open Security Platform.
http://www.netforensics.com/nfxosp.asp.

[9] New Technologies Inc. FileList Pro Computer Timeline Software.
http://www.forensics-intl.com/filelist.html.