Automated Digital Evidence Target Definition Using Outlier Analysis and Existing Evidence

By

Brian Carrier and Eugene Spafford

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2005 USA

New Orleans, LA (Aug 17th - 19th)

Automated Digital Evidence Target Definition Using Outlier Analysis and Existing Evidence

Brian D. Carrier
carrier@cerias.purdue.edu

Eugene H. Spafford
spaf@cerias.purdue.edu

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

Abstract

Searching for digital evidence is a time consuming and error-prone process. In this paper, we introduce techniques to automate the searching process by suggesting what searches could be helpful. We also use data mining techniques to find files and directories created during the incident. The results from using these techniques on a compromised honeypot system are given and show that the data mining techniques detect a higher percentage of files than a random sampling would, but there are still many false positives. More research into the error rates of manual searches is needed to fully understand the impact of automated techniques.

Key Words: Digital Evidence, Digital Investigation, Digital Forensics, Data Mining, Spatial Outlier Analysis, Automated Target Definition

1. Introduction

One of the most time consuming tasks during a digital investigation is the process of searching for evidence. If evidence of an event does not exist, then the investigator can make only assumptions about what occurred. While it is common to hear people refer to current computer forensic analysis tools as being "automated," this paper introduces additional automation into the searching process by identifying for what should be searched.

This paper describes two approaches and four implementations of automated evidence searching. The first approach suggests new searches based on evidence that has been found and the second approach uses outlier analysis to find files and directories that were created or modified during the incident. These approaches can help to make investigations more thorough and accurate because the tool keeps track of what searches still need to be conducted. We implemented four analysis tools and one suggested additional searches based on existing evidence and the other three used different outlier analysis algorithms. The implementations were run on the file system data from a compromised Linux system.

Before we discuss how to search for digital evidence, we must define it. We define *digital evidence of an incident* as digital data that contain reliable information that support or refute a hypothesis about the incident being investigated. An object is evidence because it played a role in an event that supports or refutes an investigation hypothesis [CS04]. Only a subset of these objects will be considered legal evidence and presented in court.

The remainder of this paper is organized as follows. Section 2 provides a general process model for evidence searching and four phases are defined. These phases will be used to illustrate where automation can be incorporated. Section 3 describes our automated search technique that is based on evidence as it is found by an investigator. Sections 4 to 6 describe three approaches that use data mining techniques to find files and directories that are evidence and Section 7 concludes the paper.

2. The Search Process

The search phase of a digital investigation is where the digital crime scene is processed and evidence is recognized. The primary goal of this phase is to recognize objects that played a role in events related to either the incident or a hypothesis about the incident. The motivation for evidence searching could be to verify an incident report, to find evidence of a specific event, or to test a hypothesis.

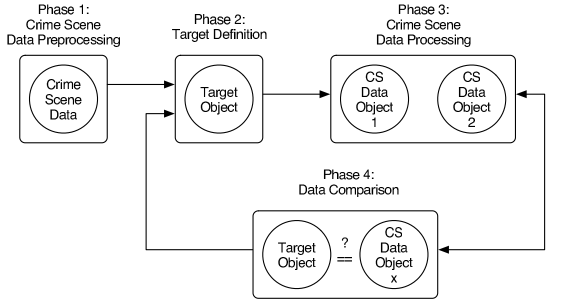

Figure 1. Flow of the four phases in the search process

The digital evidence searching process has not been formally described in the literature and this section defines the phases of the process. This is important because it will help to show where automation can occur. The general concept of the search process is to first define for what we are looking, called the target object. All searches begin with basic expectations of what kind of evidence will exist [IR01]. After the target has been defined, data from the digital crime scene are processed and compared to the target. If they match, then we may recognize the data as evidence and start a new search.

The search process used in this paper has four phases[1]: crime scene data preprocessing, target definition, crime scene data processing, and data comparison, as shown in Figure 1. The crime scene data preprocessing phase occurs a limited number of times and the other three phases are iterative and occur for each search. This assumes that the required crime scene preservation techniques have already been performed.

2.1. Crime Scene Data Preprocessing Phase

The Crime Scene Data Preprocessing Phase is where a tool processes the crime scene data to make individual searches more efficient. This is not a required phase in the search process, but is included so that accurate time and space metrics can be calculated for the entire search process. The goal of this phase is to organize the crime scene data so that the time required for the Crime Scene Data Processing Phase will be smaller.

As an example of preprocessing, consider a fictitious tool that extracts all of the ASCII words out of a file system image during the preprocessing phase and sorts them so that keyword searches are faster. This could require $O(nlog(n)m)$ time, where n is the size of the crime scene data and m is the length of the average ASCII word. With the sorted data, each keyword search can be done in $O(log(n)m)$ time. If the strings were not extracted, then each search would take $O(nm)$ time.

2.2. Target Definition Phase

The first phase of each search is the Target Definition Phase, where we determine for what we will look. After all, we need to know what characteristics evidence should have before we can recognize it. A *target object* is an abstract entity with attributes that define the characteristics we expect evidence to have. For example, its attributes may define a keyword that must exist in a file or network packet, the name or creation time of a file, or the destination port of a TCP packet.

With current investigation tools, the target object is frequently virtual and defined only in the mind of the investigator. Some tools require a partial digital representation of the target object and the user must enter the target's characteristics. For example, a user types in the keywords for a keyword search and therefore defines a digital representation of the target object.

Target objects are defined based on incident hypotheses and therefore it is one of the most challenging phases of the search process. The other three phases in the searching process are an engineering process where data are processed using algorithms, but the Target Definition Phase is more of an "art." The automated techniques presented later in this paper operate in this phase. An example of automation in this phase is the work from Stallard and Levitt, which uses audit data to define target objects that will show if there are inconsistencies in the audit data [SL03].

There are two main methods for defining target objects. One is from training and experience with the type of incident and the other is from evidence already found in this investigation.

2.3. Crime Scene Data Processing Phase

The next phase in the searching process is to process the crime scene data objects so that they are in the format needed to compare them with the target object characteristics. The amount of processing needed will depend on the type of data collected from the crime scene, the characteristics that have been defined in the target object, and the amount of crime scene data preprocessing that was performed. With older toolsets this phase required several tools, but newer toolsets can perform this phase with only one tool, which is why many refer to current tools as "automated."

During a search, data are typically processed in some pattern. At physical crime scenes, the physical objects are

[1] Note that these phases are not specific to digital investigations and can be applied to other areas, such as debugging

examined as they are seen by an investigator who is walking around in geometric patterns [LPM01]. For example, the line method involves investigators walking along a straight line looking for evidence and the grid method involves investigators who are walking in straight lines that are perpendicular to each other.

Digital crime scenes do not have a notion of geometry, but they do have a notion of structure imposed by the storage data structures, which create abstraction layers. We can use the abstraction layers to process the digital data and conduct searches [Car03]. Examples include processing a disk and viewing the sectors in a specific partition, processing a file system and viewing the names of every file and directory, and processing the bytes in a file as ASCII characters. The appropriate layer for a search will depend on what characteristics have been defined in the target object.

2.4. Data Comparison Phase

The final phase in the searching process compares the processed crime scene data object characteristics with the target object characteristics. If they match, then we can consider the crime scene data object to be evidence. In some tool implementations, this phase and the previous phase are conducted in parallel where each crime scene object is immediately compared with the target object.

Comparisons can be automated, manual, or a combination of both. Manual comparisons are typically performed using visual techniques where the data are displayed to the investigator and she compares them with the target object. For example, if an investigator used the file name search pattern, then she will be given a list of files and directories and will look for a specific name or extension. This is prone to errors because the investigator may misread data or become distracted by other data or other physical events around her.

Fully automated comparisons are rare, but they can be used for simple targets when the target object is fully represented in a digital format. For example, in a keyword search the search tool compares the bytes in each sector to the keywords. Other automated searches may find data that are within a given range or use fuzzy comparisons.

It is more common to have a combination of automated and manual comparisons. The target object will be partially defined in a digital form and the search tool will locate objects that match the digitally defined characteristics and list them. The investigator visually compares the listed objects to the target characteristics that were not digitally represented. An example is directory listings. When an investigator clicks on a directory to open it and look at its contents, he defines a digital target for the directory and then manually compares each file name.

2.5. Automation

When the four phases are examined, it is clear that some phases will benefit more from automation than others. The Crime Scene Data Preprocessing and Crime Scene Data Processing phases are already automated in most tools and require little user intervention. The Data Comparison Phase is easily automated if the target object is digitally defined. The Target Definition Phase has the least amount of automation and is therefore the focus of the techniques in this paper. Automating the Target Definition Phase also has the benefit of having a digital target object, which can help in the automation of the Data Comparison Phase.

3. Target Definition Based on Existing Evidence

3.1. Overview

In this section, we outline a technique for automated target definition based on existing evidence. The basic concept is that the investigator uses an analysis tool to identify files as evidence and the tool makes suggestions for additional searches. This is an intuitive step that all investigators perform when they find evidence, but it has been a manual step and not integrated into the analysis tool.

This approach was implemented using the Autopsy Forensic Browser [Car04a] and The Sleuth Kit (TSK) [Car04b]. TSK is a collection of Unix-based command line tools and Autopsy is a HTML-based graphical interface to the tools. They are designed to analyze a bitwise copy of a hard disk or partition, called an image file. There were two major features that needed to be added to Autopsy for the automated searching. The first was a method for storing the target objects so that they can be later used and the second was the process of suggesting new target objects. These two issues are described next.

3.1.1. Digital Storage of Target Objects The first new addition to Autopsy was the digital representation and storage of target objects. In the proof of concept implementation, only seven characteristics of the target objects were supported. The characteristics of the objects were saved in a comma delimited ASCII text file so that the investigator could save searches for a later time. A more scalable implementation would likely benefit from an XML format. The seven characteristics that can be defined are:

- Parent Directory Name
- File Name
- Last Modified (or Written) Time
- Last Accessed Time

- Last Changed (or Created) Time
- Application Type
- Content Keyword

After a target object has been defined, an investigator can conduct the search by selecting the target from a list. If the file or directory name fields were defined in the target, then a file name search pattern is used and the search hits are listed so that the investigator can view the contents and metadata information for each. If the file content field was used in the target, then a keyword search is conducted in each data unit. If the application type was defined in the target, then Autopsy will show a list of files that have the same application type. If the time field was defined in the target object, then the timeline view of the file system is used to show the files that had activity at that time.

3.1.2. Target Object Suggestions The second addition to Autopsy was a process to suggest new target definitions based on the characteristics of recently found evidence. The original version of Autopsy had a feature to create a note, similar to a bookmark, about data. In the updated version, the window that allows the investigator to make notes also provides a list of additional searches that could be relevant. The investigator can choose to save or ignore the suggestions.

The basic approach uses several static algorithms for suggesting new targets. The user has the choice of editing the target objects before saving them. If the new evidence was a file, then up to seven default target objects are suggested. If the suggestion has already been made, then it will not be shown again.

Parent Directory: The target object has the `Parent Directory Name` attribute defined with the same name as the parent directory of the evidence. This is useful when a file is found because of a keyword and the surrounding files have not yet been examined.

Similar Name: The target object has the `File Name` attribute defined with the same name as the evidence. The investigator can remove the extension or other parts of the name so that files with similar names or extensions can be found. This is useful to find configuration files that have a different extension or to find duplicate copies of the file.

Name in Content: The target object has the `Content Keyword` attribute defined with the name of the evidence file. This will find files that reference the evidence file and have not used any obfuscation techniques.

Temporal Data: Up to three target objects are created and each has one of the `Last Modified` or `Written`, `Last Accessed`, or `Last Changed` or `Created` values defined. This can be used to find other files that played a role in events in the same time frame as the evidence.

Application Type: The target object has the `Application Type` attribute defined with the same type as the evidence file. This can be used to find additional files of the same type.

The investigator is also given the ASCII strings from the file content and she can create additional target objects based on them. For example, a target object's `File Name` or `Content Keyword` characteristic can be defined using data in the evidence file's content. Future work will work on techniques to suggest searches based on the content.

While the suggested searches are all intuitive, the unique aspect is that the investigator is reminded of the additional searches that could be useful. These are saved so that the investigator does not forget to conduct them.

3.2. Case Study Results

To test the target object definition abilities of Autopsy, the file system images from the Honeynet Forensic Challenge [Hon01] were used. The images are from a Linux system that was placed on the Internet in November of 2000 and it was broken into shortly afterwards. Because the system is from 2000 the attack techniques and tools are outdated, but the automated evidence searching techniques are not specific to any attack tools or techniques.

The file system images were imported into the updated version of Autopsy and the case knowledge was collected. We know that it is a Linux system, its address was not advertised, and it may have been compromised. Through experience or training, we know that many intrusions of systems that are not advertised are from automated scanning tools and the attackers install rootkits.

To check for rootkits, we mentally define target objects for rootkit signatures, which we know from experience. Rootkits commonly create configuration files in the `/dev/` directory of UNIX systems. These can be detected because they are a text file in a directory that typically contains only special device files. Using our target object we find the `/dev/ptyp` file, which contains a list of names that look like application names. Based on this file, Autopsy suggested a keyword search for the string "/dev/ptyp" and other files that have the same change time. Autopsy suggested other searches as well, but we ignored them because we know the nature of the rootkit configuration files.

The recommended keyword search for "/dev/ptyp" has hits in four locations in the `/` partition, including the `/bin/ps` file, which was modified by the attacker to hide processes that had a name listed in `/dev/ptyp`. Therefore, we know that a user-level rootkit was installed and we use our experience to conclude that the other system executables should be searched for and analyzed. From the `/bin/ps` file, Autopsy suggested new searches, but these did not find additional evidence.

We return to the saved target objects and conduct the temporal search for other files that have a similar last changed time as the `/dev/ptyp` file. This finds 17 files and directories that were changed within five seconds of the `/dev/ptyp` file. All 17 of the files are related to the incident and we learn where the rootkit was stored.

4. Single Attribute File Outlier Detection

4.1. Overview

A second method of automatic target definition uses outlier analysis to find files that have been hidden or that are different from their surrounding files. *Outlier objects* are objects that "are grossly different from or inconsistent with the remaining set of data" [HK01] and outlier analysis tries to find the outlier objects. The motivation for this target definition technique is that it is common for new files to be created during an incident. The new files could be rootkits that will hide the attacker's activity, copyrighted material that will be illegally shared, or other attack tools. The attacker typically takes steps to hide these files from detection. Outlier analysis may be able to detect these hidden files.

Before we consider how to detect the hidden files, we must identify how files are typically hidden. One method is to place them in a directory that already has many files and the theory is that they will not be noticed by casual observation. As we previously saw, a common example of this is to hide files in the `/dev/` directory of a Unix system, which may have thousands of files with archaic names. The files to be hidden are given names similar to existing files and the temporal data are changed. A similar method is when the attacker replaces system files with trojan versions that will collect sensitive data or hide incident data. For example, the Unix `ps` command could be replaced with a version that hides the processes being run for the attacker. The files hidden using these methods will likely have different characteristics from their surrounding files.

For this searching technique, spatial outlier algorithms were used. A *spatial outlier* is an outlier with respect to its neighboring spatial values, even if it is not an outlier with respect to the entire set of values [StLZ01]. The attribute used to compare points is not spatial-based. In this case, the spatial location of a file or directory is its parent directory, as can be seen in Figure 2, where the location of the `/dir1/blue.bmp` file is `/dir1/`.

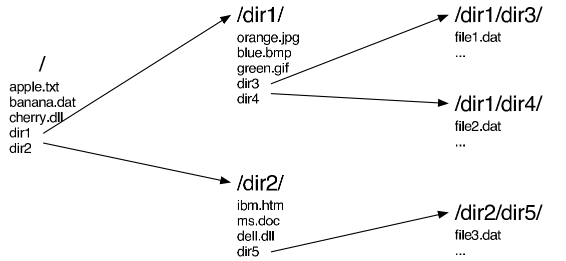

Figure 2. The directory structure can be viewed as a spatial layout.

To find hidden files and other evidence, the characteristics of files in a directory were compared with each other to find outliers. For example, in Figure 2 we would compare `/dir1/blue.bmp` with other files in `/dir1/` but not with files in `/dir2/`. The characteristics that were considered include:

- File Size
- Last Changed (or Created) Time
- Last Modified (or Written) Time
- Starting Block Address
- Application Type (i.e. JPEG, Word document).

4.2. Implementation

An analysis tool to detect hidden files was developed using Perl and the command line tools from TSK. TSK was used to extract the file system data and the *iterative z* algorithm [LCK03a] was used for spatial outlier detection. Let there be a set of spatial points $X = \{x_1, x_2, ..., x_n\}$, an attribute function $f(x_i)$ that represents the attribute value for point x_i, and neighborhood function $NN_k(x_i)$ that represents the k nearest neighbors to point x_i. In this implementation, the neighborhood was not restricted to only k members and instead it included all files in the directory. The function $g(x_i)$ returns a summary metric for the points in the neighborhood $NN_k(x_i)$. For example, the average attribute value for the directory. The basic concept of spatial outlier detection is to compare the attribute function $f(x_i)$ with the group summary $g(x_i)$ using a comparison function $h()$ to detect if the attribute of x_i is different from its neighbors.

There are many methods for choose the size of the neighborhood, the group summary function $g()$, and comparison function $h()$. The algorithm used for this analysis worked as follows:

1. Define each file as a spatial point x_i (for $i = 1, 2, ..n$), and define the neighborhood $NN(x_i)$ as the set of files in the same directory as file x_i.

2. Compute the neighborhood summary $g(x_i) = \frac{1}{|NN(x_i)|} \sum_{x \in NN(x_i)} f(x)$, and comparison $h_i = h(x_i) = f(x_i) - g(x_i)$ for each point x_i.
3. Given the set of differences $\{h_1, h_2, ...h_n\}$ calculate the mean, μ, and standard deviation, σ. Standardize the points using $y_i = \left|\frac{h_i - \mu}{\sigma}\right|$ for $i = 1, 2, ...n$.
4. Let y_q be the largest y_i. If it is larger than a threshold θ then x_q is an outlier.
5. Remove x_q from the data so we can examine the other points. To do this, set the attribute value of x_q to be the average of the neighborhood, $g(x_q)$. Return to step 1 (or update only the values effected by the change). If no outliers are found, then stop.

For a normal distribution, the value of θ can be 2 or 3. The replacement in the final step is done to remove the bias added to the average calculation with the outlier in it.

The iterative z algorithm works for most file attributes, but not for a file's application type. The standard algorithm assumes a single dimension attribute that can be subtracted to get the distance between two points (or files). This works for attributes such as last access time or file size, but not for the application type. The implementation of this analysis technique used the `sorter` tool from TSK and it organizes files into one of 13 categories of application types. Example application types include graphic images, executables, documents, and compressed data.

The naive approach is to assign the attribute value of each file to a number from 1 to 13 based on its category. The error with this approach is that when the average of, for example, an executable file with a category id of 3 and an image file with an id of 5 are calculated then we get the same value for an ASCII text file, whose category id is 4. Clearly this is not an accurate representation of the application types in a directory because the same average value can be achieved from ids of 2 and 6 from, for example, compressed data and HTML files.

The solution was to make the application type attribute be a 13-dimensional value. Dimension i is 1 if the file's application category id is i. For all other dimensions, the value was 0. The distance and average calculations were done using euclidean distance in the 13-d space. With this method, the application types can be more accurately compared and averaged.

4.3. Case Study Results

The implementation was used on the file system images from the Honeynet Forensic Challenge. The goal of the outlier analysis was to detect the `/dev/ptyp` rootkit file, the trojan executables, the `/usr/man/.Ci/` installation directory, and other system changes. In the case of this incident, the attacker patched vulnerable programs on the system, so several artifacts of the patching process may also be found. Example artifacts include the relevant executable, documentation, and configuration files.

We ran the file-based outlier detection routines on the `/` and `/usr/` partitions and the results with $\theta = 2$ and 3 can be found in Table 1. The `/` partition had 6,951 files and 50 (0.72%) of which were created or modified during the incident. The `/usr/` partition had 21,267 files and 323 (1.49%) were created or modified during the incident.

The first column lists the attribute that was used in the analysis. Columns two to four are for the `/` partition, five to seven are for the `/usr/` partition, and eight to ten combine the results. The second column shows the percentage of all files that were identified by the analysis and the third column shows the percentage that were actually involved in the incident. The fourth column gives the percentage of how comprehensive the search was, or how many of the incident related files were found by the analysis. Columns five to seven and eight to ten show the same categories of data.

When we look at the combined results we see that the last changed time (C-time) has a high success rate, but less than 10% of the files identified using other attributes were actually related to the incident. Using a θ value of 2 or 3 has little impact because some accuracy rates increase and others decrease.

In this system, the trojan executables had their temporal data set to the times of the original executables and they were therefore not identified. We had hoped that the starting block attribute would find the trojan executables because they were created after the other system executables and therefore their starting block would be much larger than the starting blocks of the other system executables. In the `/` partition this was not the case, but the starting block of the trojan executables in the `/usr/` partition were outliers and detected. The Ext2 file system allocation routines likely contributed to the trojans not being found in `/` because all files in a directory are allocated to the same block group, which makes them start close together [Car05]. We expected the application type and size attributes to find the `/dev/ptyp` file and other hidden rootkit files, and they did.

To improve the success rate, a modified version of the iterative z algorithm was used. Instead of calculating the mean, μ, and standard deviation, σ in step 3 over all n, they were calculated over only the files in the directory. In the original version, if some directories have a large difference in the attribute values and some do not, then files can be hidden in the directories with little difference. With the modified version, this would be more difficult. The results from running the modified version on the `/` and `/usr/` partitions can be found in Table 2, where the columns are the

$\theta = 2$									
	/ partition			`/usr/` partition			Combined		
Attribute	**Hits**	**Accuracy**	**Comp**	**Hits**	**Accuracy**	**Comp**	**Hits**	**Accuracy**	**Comp**
App Type	1.09%	9.21%	14.00%	12.03%	2.46%	19.50%	9.34%	2.66%	18.77%
Block	8.19%	5.80%	66.00%	6.54%	4.03%	17.34%	6.95%	4.54%	23.86%
C-time	2.75%	22.51%	86.00%	1.27%	54.98%	46.13%	1.64%	41.56%	51.47%
M-time	0.89%	16.13%	20.00%	10.39%	6.79%	46.44%	8.05%	7.04%	42.90%
Size	6.66%	2.81%	26.00%	8.72%	3.23%	18.58%	8.21%	3.15%	19.57%

$\theta = 3$									
	/ partition			`/usr/` partition			Combined		
Attribute	**Hits**	**Accuracy**	**Comp**	**Hits**	**Accuracy**	**Comp**	**Hits**	**Accuracy**	**Comp**
App Type	1.01%	10.00%	14.00%	9.15%	3.08%	18.58%	7.14%	3.33%	17.96%
Block	6.93%	6.22%	60.00%	3.96%	2.97%	7.74%	4.69%	4.15%	14.75%
C-time	2.24%	26.92%	84.00%	1.27%	54.44%	45.51%	1.51%	44.37%	50.67%
M-time	0.75%	5.77%	6.00%	8.26%	7.63%	41.49%	6.41%	7.58%	36.73%
Size	4.78%	2.41%	16.00%	5.84%	3.22%	12.38%	5.58%	3.05%	12.87%

Table 1. File-based single attribute outlier detection results.

same as in Table 1. This algorithm provides higher accuracy for the last modified time (M-time) and C-time, but less accuracy for the application type, starting block, and size attributes. The M-time has a large standard deviation across the full file system and therefore the original algorithm could not use it to identify many outliers.

To evaluate the accuracy of the algorithms, we must consider how many files were actually created or modified during the incident. When both partitions are considered, 1.32% of the files were created or modified and therefore we would expect that, on average, any random subset of files would have an accuracy rate of 1.32%. When we consider the original algorithm, the lowest accuracy rate is twice as large as the expected number.

Further, many of the false positives were obvious. For example, many of the false positives on the `/usr/` partition using the size attribute were from the README or CHANGELOG files that came with software packages. Therefore, a special user interface could allow the investigator to quickly view a random part of the file to verify it is a false positive.

One problem of using a honeypot system is that there is little user activity and therefore the incident data could be more obvious than it would be on a real system. In light of this, the analysis programs were run on a normal Microsoft Windows system. This system was not involved in any incident and has been used for at least 2 years. On average, with $\theta = 3$ there were 2.58% of the files identified using the original algorithm and 0.81% of the files identified using the modified algorithm. Obviously, there was a 100% false positive rate as there were no incident-related files. For comparison, 5.07% and 1.76% of the honeypot files were identified using the original and modified algorithms, respectively. This shows that a normal system may have fewer false positives because the attributes are more diverse and therefore fewer outliers. The accuracy of running the algorithms on a normal system remains to be seen though and is future work.

5. Multiple Attribute File Outlier Detection

5.1. Overview

To reduce the number of false positives that were found using single attribute outlier analysis, we also tried multiple attribute outlier analysis. Multiple attribute outlier analysis uses two or more attributes for each point to detect outliers. The motivation for using multiple attributes was that some of the false positives occurred because one of the file's attributes was very large or small. If we consider the file's other attributes, then it may no longer be an outlier and this would reduce the false positive value. On the other hand, it could also reduce the accuracy count because some of the incident files will have only one outlier attribute and therefore not be identified.

To test the multiple attribute outlier analysis techniques, a program similar to the single attribute outlier analysis was written using Perl and TSK but a different algorithm was used. One of the difficulties when considering multiple attributes of data is how to measure the distance between the points. We used the Mahalanobis distance as the metric [BL84] [LCK03b].

The algorithm uses a set of spatial points $X = \{x_1, x_2, ..., x_n\}$ and each point has q attributes. The attribute function $f(x_i)$ returns a vector of the q attribute values and attribute a can be referenced as $f_a(x_i)$ for $0 < a \leq q$. Let $g(x_i)$ be the neighborhood summary function, which returns a vector where each at-

$\theta = 2$									
	/ partition			`/usr/` partition			Combined		
Attribute	**Hits**	**Accuracy**	**Comp**	**Hits**	**Accuracy**	**Comp**	**Hits**	**Accuracy**	**Comp**
App Type	0.33%	30.43%	14.00%	1.69%	0.28%	0.31%	1.36%	2.09%	2.14%
Block	0.69%	4.17%	4.00%	4.74%	2.78%	8.67%	3.74%	2.84%	8.04%
C-time	1.14%	51.90%	82.00%	0.89%	53.44%	31.27%	0.95%	52.99%	38.07%
M-time	0.91%	20.63%	26.00%	3.49%	20.22%	46.44%	2.85%	20.25%	43.70%
Size	3.55%	3.64%	18.00%	19.76%	2.00%	26.01%	15.77%	2.09%	24.93%

$\theta = 3$									
	/ partition			`/usr/` partition			Combined		
Attribute	**Hits**	**Accuracy**	**Comp**	**Hits**	**Accuracy**	**Comp**	**Hits**	**Accuracy**	**Comp**
App Type	0.10%	14.29%	2.00%	0.59%	0.80%	0.31%	0.47%	1.52%	0.54%
Block	0.16%	18.18%	4.00%	0.81%	2.31%	1.24%	0.65%	3.26%	1.61%
C-time	0.55%	34.21%	26.00%	0.62%	47.33%	19.20%	0.60%	44.38%	20.11%
M-time	0.27%	31.58%	12.00%	1.38%	27.21%	24.77%	1.11%	27.48%	23.06%
Size	1.45%	3.96%	8.00%	7.48%	2.20%	10.84%	6.00%	2.30%	10.46%

Table 2. File-based single attribute outlier detection results with modified algorithm.

tribute has an element. The algorithm we used is as follows:

1. Standardize each attribute value, such as $f_j(x_i) = \frac{f_j(x_i)-\mu_{f_j}}{\sigma_{f_j}}$ for each attribute $0 < j \leq q$. Where μ_{f_j} is the mean of attribute j and σ_{f_j} is the standard deviation of attribute j.
2. Compute the average neighborhood function $g(x_i) = \frac{1}{|NN(x_i)|} \sum_{x \in NN(x_i)} f(x)$ and the comparison function $h(x_i) = f(x_i) - g(x_i)$.
3. Calculate the directory sample mean as $\mu_s = \frac{1}{|NN(x_i)|} \sum_{x \in NN(x_i)} h(x)$. Calculate the directory variance-covariance $\Sigma_s = \frac{1}{|NN(x_i)|-1} \sum_{x \in NN(x_i)} [h(x) - \mu_s][h(x) - \mu_s]^T$.
4. Compute the Mahalanobis distance from each file to the directory sample mean using the variance-covariance matrix. Let distance $a(x_i) = a_i = (h(x_i) - \mu_s)^T \Sigma (h(x_i) - \mu_s)$.
5. Identify the files with an outlier Mahalanobis distance by calculating the mean μ_m and standard deviation σ_m for the distances a_i in the directory. If $\frac{a_i - \mu_m}{\sigma_m} > \theta$ then x_i is an outlier. This last step is the same as the single attribute outliers and a theta value of 3 was used.

The last step is not included in the algorithm in [LCK03b], but was added because otherwise a threshold must be chosen for the distance value and this can change for each system.

5.2. Case Study Results

The analysis program was run on the `/` and `/usr/` partitions from the Forensic Challenge. Four attribute sets were considered, as shown in Table 3. One set of attributes was the last modified and last changed times. The next set added the file size and the third set added starting block address. The final set considered all five attributes that were previously used: last changed, last modified, size, starting block address, and application type.

We can see from these results that the accuracy when using multiple attributes is higher than when using only a single attribute. The high accuracy of the C-time alone likely contributes to this, but these results show that most of the files identified by the C-time attribute using the single attribute analysis were not outliers with with the M-time attribute. When the C-time was used alone, it was identifying, between 1% to 1.5% of the total files, but when combined with the M-time only 0.50% of the files were identified. Note that the accuracy rate when the M-time and C-time were combined was less than the C-time alone and greater than the M-time alone.

When the multiple attribute outlier detection was run on the clean Windows system, no files were identified when the C-time, M-time, size, and starting block combination or the C-time, M-time, size, starting block, and application type combination was used. When the C-time and M-time combination was used then only 0.63% of the files were identified and when the size was added then only 0.54% were identified. As with the single attribute analysis, the normal system had fewer hits, but the accuracy remains to be seen because there were no hidden or incident-related files on the system.

	/ partition			/usr/ partition			Combined		
Attribute	**Hits**	**Accuracy**	**Comp**	**Hits**	**Accuracy**	**Comp**	**Hits**	**Accuracy**	**Comp**
C- and M-time	0.30%	85.71%	36.00%	0.56%	23.33%	8.67%	0.50%	32.62%	12.33%
C-, M-time, and Size	0.22%	53.33%	16.00%	0.32%	43.28%	8.98%	0.29%	45.12%	9.92%
C-, M-time, Size, and Block	0.22%	53.33%	16.00%	0.22%	31.91%	4.64%	0.22%	37.10%	6.17%
C-, M-time, Size, Block and App Type	0.10%	57.14%	8.00%	0.15%	38.71%	3.72%	0.13%	42.11%	4.29%

Table 3. File-based multiple attribute outlier detection results with $\theta = 3$.

6. Single Attribute Directory Outlier Detection

6.1. Overview

In addition to files, an attacker may also create directories that contain her files. The directory is typically named similar to the surrounding directories or with special characters. A common Unix example is to start the directory with a `.` so that it is not shown by default. Other examples add spaces and special control characters into the name, such as `.. ` (two periods followed by a space). Outlier analysis was used to detect hidden directories.

To find suspicious directories, the characteristics of directories at the same level were compared. For example, if directory `/a/x/` is the hidden directory, then we will compare it to directories `/a/m/`, `/a/n/`, and `/a/o/`, but not `/a/m/t/` or `/b/c/`. The theory is that the average characteristic value for the hidden directory will be an outlier with respect to the average characteristic value for the other directories.

To implement this analysis technique, we wrote a Perl script and used tools from TSK. The single attribute iterative z algorithm was used to analyze the extracted data.

6.2. Case Study Results

The `/` and `/usr/` partitions from the Forensic Challenge were analyzed using the directory outlier analysis script and the results can be found in Table 4. There is only one directory in this system that contains hidden files and it is the `/usr/man/.Ci/` directory. The `/` partition had 89 directories and none of which were hidden. The `/usr/` partition had 1,088 directories and 1 (0.09%) of which was hidden. As can be seen, when run on the `/` partition with a threshold of 2 over 10% of the directories were, on average, identified as being hidden, all of which were false positives. When run on the `/usr/` partition, the directory was found, but there were hundreds of other directories that were falsely identified. When the program was run again with a threshold of 3, much fewer directories are falsely identified, but the hidden directory was also not found.

When the analysis tool was run on the clean Windows system with a threshold of 2 then 3.44% directories were, on average, identified as being hidden. Using a threshold of 3, the algorithm identified 0.78%, on average, directories. This is less than we saw with the Linux system, which could be because the system is used more and therefore the directory attributes are more diverse and therefore outliers are less common. A decrease in hits from the clean system was previously seen with the file outlier analysis as well.

7. Conclusion

In this paper, we have outlined the basic phases of the searching process and proposed several techniques that can help automate the Target Definition Phase. The other phases in the searching process already have some level of automation. The purpose of automating the Target Definition Phase is to help identify the characteristics of files that could be evidence. It is unreasonable to expect automated searching techniques to identify all evidence files because a file is evidence with respect to a given question or hypothesis and no algorithm can give answers to all questions.

The Autopsy tool was updated to suggest additional searches based on existing evidence. This will help an investigator organize the list of searches that he needs to conduct and therefore decrease the number of searches about which he forgets. This process does not necessarily suggest searches that an experienced investigator would not otherwise perform, but it helps to formalize the process.

This paper also applied several data mining techniques to detect outlier files and directories, which an attacker could have created or modified. When run on the honeypot system, the last changed and last modified times had a high accuracy rate, but, overall, the results had a high false positive rate. Using multiple attributes helps to reduce false positives, but it also misses some of the hidden files that were found using only a single attribute. The test data had only

$\theta = 2$									
	/ partition			`/usr/` partition			Combined		
Attribute	**Hits**	**Accuracy**	**Comp**	**Hits**	**Accuracy**	**Comp**	**Hits**	**Accuracy**	**Comp**
App Type	12.36%	0.00%	n/a%	9.19%	1.00%	100.00%	9.43%	0.90%	100.00%
C-time	8.99%	0.00%	n/a%	5.24%	1.75%	100.00%	5.52%	1.54%	100.00%
M-time	12.36%	0.00%	n/a%	12.87%	0.71%	100.00%	12.83%	0.66%	100.00%
Size	11.24%	0.00%	n/a%	10.94%	0.84%	100.00%	10.96%	0.78%	100.00%

$\theta = 3$									
	/ partition			`/usr/` partition			Combined		
Attribute	**Hits**	**Accuracy**	**Comp**	**Hits**	**Accuracy**	**Comp**	**Hits**	**Accuracy**	**Comp**
App Type	0.00%	0.00%	n/a%	7.44%	0.00%	0.00%	6.88%	0.00%	0.00%
C-time	2.25%	0.00%	n/a%	1.65%	0.00%	0.00%	1.70%	0.00%	0.00%
M-time	1.12%	0.00%	n/a%	1.38%	0.00%	0.00%	1.36%	0.00%	0.00%
Size	1.12%	0.00%	n/a%	3.77%	0.00%	0.00%	3.57%	0.00%	0.00%

Table 4. Directory-based outlier detection results.

one hidden directory and therefore it is difficult to draw conclusions from its results.

Based on this work, there are many areas of future work. First, it is difficult to evaluate how effective an automated search technique is when the error rate and time complexity of manual searching is unknown. Second, the outlier techniques will likely have different results on different types of systems and different types of incidents. For example, some systems may have only one file type in each directory and others may have all file types in each directory. More exhaustive testing should be conducted to identify in which conditions this is most effective. Also, the time between the incident and the investigation may have a large effect on the effectiveness of the outlier techniques.

In the future, these analysis techniques could be combined with others to identify the evidence. For example, a data mining algorithm could be used to identify a subset of suspect data and keyword searching or hash analysis is then conducted. Similarly, backup tapes could be automatically used to estimate the age of a suspicious file. When investigating recent incidents, older files in the outlier analysis results can, initially, be hidden from the investigator. As storage sizes grow larger, data mining techniques may help with the evidence searching process [BC05]. Outlier analysis is a technique that can help to detect evidence.

References

[BC05] Nicole Lang Beebe and Jan Guynes Clark. Approaching the Terabyte Dataset Problem in Digital Investigations. *Research Advances in Digital Forensics*, 2005.

[BL84] Vic Barnett and Toby Lewis. *Outliers in Statistical Data*. John Wiley and Sons, 2 edition, 1984.

[Car03] Brian Carrier. Defining Digital Forensic Examination and Analysis Tools Using Abstraction Layers. *International Journal of Digital Evidence*, Winter 2003.

[Car04a] Brian Carrier. *The Autopsy Forensic Browser*, 2004. Available at: http://www.sleuthkit.org.

[Car04b] Brian Carrier. *The Sleuth Kit*, 2004. Available at: http://www.sleuthkit.org.

[Car05] Brian Carrier. *File System Forensic Analysis*. Addison Wesley, 2005.

[CS04] Brian D. Carrier and Eugene H. Spafford. Defining Event Reconstruction of a Digital Crime Scene. *Journal of Forensic Sciences*, 49(6), 2004.

[HK01] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Mogan Kaufmann, 2001.

[Hon01] Honeynet Project. *Forensic Challenge*, 2001. Available at: http://www.honeynet.org/.

[IR01] Keith Inman and Norah Rudin. *Principles and Practice of Criminalistics: The Profession of Forensic Science*. CRC PRess, 2001.

[LCK03a] Chang-Tien Lu, Dechang Chen, and Yufeng Kou. Algorithms for Spatial Outlier Detection. In *Proceedings of the 3rd IEEE International Conference on Data Mining*, 2003.

[LCK03b] Chang-Tien Lu, Dechang Chen, and Yufeng Kou. Detection Spatial Outliers with Multiple Attributes. In *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, 2003.

[LPM01] Henry Lee, Timothy Palmbach, and Marilyn Miller. *Henry Lee's Crime Scene Handbook*. Academic Press, 2001.

[SL03] Tye Stallard and Karl Levitt. Automated Analysis for Digital Forensic Science: Semantic Integrity Checking. In *Proceedings of the ACSAC 2003*, 2003.

[StLZ01] Shashi Shekhar, Chang tien Lu, and Pusheng Zhang. Detecting Graph-based Spatial Outliers: Algorithms and Applications. Technical report, University of Minnesota, 2001.