NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AN AUTOMATED ACQUISITION SYSTEM FOR MEDIA EXPLOITATION

by

Steven D. Bassi Jr.

June 2008

Thesis Advisor:	Simson L. Garfinkel, Ph.D.
Second Reader:	Chris S. Eagle

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704–0188*

1. REPORT DATE *(DD–MM–YYYY)* 10-06-2008	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* 20-09-2006—10-06-2008

4. TITLE AND SUBTITLE An Automated Acquisition System for Media Exploitation	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Steven D. Bassi Jr.	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of the Navy	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This thesis explores the requirements for building a highly usable acquisition system for an automated document and media exploitation system and presents work to date on AcqMan, an acquisition manager written in Java which uses DBUS and the Hardware Abstraction Layer (HAL) to automatically detect device insertion and start forensic imaging.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			
Unclassified	Unclassified	Unclassified	UU	73	**19b. TELEPHONE NUMBER** *(include area code)*

Standard Form 298 (Rev. 8–98)
Prescribed by ANSI Std. Z39.18

THIS PAGE INTENTIONALLY LEFT BLANK

AN AUTOMATED ACQUISITION SYSTEM FOR MEDIA EXPLOITATION

Steven D. Bassi Jr.
Civilian, SFS Cybercorps
B.S., Santa Clara University, 2006

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2008

Author:	Steven D. Bassi Jr.
Approved by:	Simson L. Garfinkel, Ph.D. Thesis Advisor
	Chris S. Eagle Second Reader
	Peter J. Denning Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis explores the requirements for building a highly usable acquisition system for an automated document and media exploitation system and presents work to date on AcqMan, an acquisition manager written in Java which uses DBUS and the Hardware Abstraction Layer (HAL) to automatically detect device insertion and start forensic imaging.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgements

I would like to thank my parents, Steve & Leslie Bassi, for their love and support over the course of my life. This thesis would not have been possible without all their guidance throughout the years.

Next, I would like to express my thanks to Chris Eagle not only for agreeing to be my thesis reader but also for taking the time to help improve my skillset. The ad-hoc Friday meetings will be sorely missed.

Also, I want to thank Cynthia Irvine for creating and managing the Scholarship for Service program at NPS. The whole NPS experience would not have been possible without her.

Valerie Linhoff also deserves a great deal of my gratitude for getting me through all the phases of the NPS education. Her open door policy and willingness to help was key to my successful time at NPS.

Last, I'd like to thank Simson Garfinkel for his enthusiasm and help in all aspects of this thesis. My work was significantly enhanced through his insights and experiences.

This material is based on work supported by the National Science Foundation under Grant DUE-0414102.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: Introduction

More and more, intelligence gained from digital sources is on the forefront of providing solutions to crimes, military conflicts, and tough diplomatic problems. These digital sources are gathered through a variety of means and their contents are processed by yet another variety of means. Together, the acquisition and analysis is known as Document and Media Exploitation, or DOMEX, to the U.S. Government[14].

The DOMEX process may differ from organization to organization but generally boils down to “turn[ing] digital bits into actionable intelligence[15].” Turning this information into actionable intelligence involves many steps in the DOMEX process, including information processing, translation, analysis, and dissemination.

“Digital bits” is a broad term, and in the DOMEX environment these bits can be found on any media from cellular phone memory to hard disks. This broad spectrum of media inherently makes the DOMEX process a diffcult, and often manual, process as the analyst is forced to adapt to different media formats in order to extract actionable intelligence.

For the purposes of this thesis, it is helpful to understand what each step in the DOMEX process entails so that the reader can understand the area in which this thesis is concerned.

- **Processing**
  The processing stage of DOMEX is a critical step where all the information is placed in a form that is consumable by the analysis step. In a digital DOMEX environment this means that all information on any seized device must be available for analysis by a human or computer. Furthermore, this information must be accurate and perhaps put in a common format to reduce the complexity of the analysis step. This common format is often referred to as an image which in this context means a bit-for-bit copy of all the information on a device.

- **Translation**
  After processing, the information on a piece of media may contain information in a language that is often not native to an analyst. In order for the consumers an analysis to understand this information it must be translated into their native language.

- **Analysis**
  The analysis phase of DOMEX can consist of anything from simple statistics about a single captured device to a complex social network analysis encompassing many seized devices. As more information is made available in the analysis phase, new types of analysis become possible. These analysis can be done by a human or a machine.

- **Dissemination**
  The dissemination step of the DOMEX process involves distributing the results of the analysis back to humans in a form that they consider "actionable intelligence." Basically, it involves getting the right information to the right people at the right time.

Due to the complexity of the analysis, the volume of data, and the need for translation DOMEX is ripe for automation at every step in the process.

1.1 Purpose of Study

This study seeks to bring a high level of automation to the first phase of DOMEX: information processing. Information processing is a loose term that can be defined as many things in a DOMEX environment. We envision information processing as the stage where information is acquired from physical media and stored in a common base format and location from which all further steps in the process can use.

The purpose of this study is to explore a method for collecting and aggregating the contents of seized digital media in a form and location that supports the rest of the DOMEX process. Specifically, we wish to make the digital media acquisition and aggregation portion of the DOMEX process as automated, quick, and error-free as possible. In doing this, we hope to enable other parts of the DOMEX system to rapidly analyze the collected information and disseminate the results back to those who need it.

As such, this thesis attempts to answer the following questions:

1. What is a workable design for an media acquisition system that supports DOMEX operations?

2. What can be done to make this design as automated, quick, and error-free as possible so that it can be used by trained and untrained users?

1.2 Thesis Organization

We believe the requirements for a DOMEX acquisition system are much different from other media acquisition systems. We explore prior work in DOMEX like systems in Chapter 2. Next, we describe our envisioned requirements for a DOMEX acquisition tool in Chapter 3. In Chapter 4 we compare existing acquisition tools against our requirements in an attempt to show how current tools might be adapted into a DOMEX acquisition tool. Chapter 5 will expound on our implementation of a DOMEX acquisition system by describing its features and how they might meet some of the requirements set forth in Chapter 3. Finally, Chapter 6 will summarize our findings and suggest future directions for our work.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2: Prior Work in DOMEX

2.1 DOMEX Processes

We were unable to find open publications which stated the DOMEX capabilities and activities of US military and government organizations. Furthermore, we were unable to find references to tools and technologies that organizations such as FBI, DC3, and NMEC might be using for their DOMEX operations. This is expected in that publishing these capabilities would allow an adversary to form DOMEX countermeasures. However, we can make a reasonable assumption that DOMEX activities are occurring within some of these agencies through both an intelligence directive discussing DOMEX and a description of a portable DOMEX environment for the military[14][22].

However, Simson Garfinkel has published an article detailing his vision for automating the DOMEX process[15]. In the article he defines DOMEX, lays out its problems, and describes how the process can be automated. His plans for automation break down roughly along the lines of information acquisition and information analysis. This thesis focuses on automated information acquisition for the purposes of analysis which we discuss in the next chapter.

On the information analysis side of DOMEX, Garfinkel lists several steps which should be taken on acquired media to extract the maximum amount of useful information in an automated fashion. These steps consist of device imaging, file system analysis, file analysis and feature extraction, anomaly detection and social network analysis, and reporting. Tools exist to perform one or more of these functions but there are no tools which implement all steps as described in Garfinkel's article.

2.2 Automated DOMEX Tools

There are two publicly available tools which handle, to some degree, the kind of automated multiple image analysis and processing that would be useful in automated DOMEX. These tools stand out from other forensics tools in that they seek to automate the image analysis process on a large number of drives at once. This is different from the single drive, single examiner approach that is seen in most forensics tools.

2.2.1 The Open Computer Forensics Architecture

The OCFA is a toolset developed by the Dutch National Police to "automate the digital forensics process to speed up the investigation[10]." This toolset does not handle the acquisition of physical media and therefore is not directly applicable to this thesis. However, we did find the tool's aim of handling large numbers of images interesting.

The tool is designed to ingest and process large numbers of drive images. These images can be logically grouped together into the same case thus enabling an investigator to simultaneously view processing results from the many images in the same case.

OCFA is designed around the concept of a series of modules, each of which process and extract meaning from a given image. These modules are connected together by a virtual network which is able to dictate the order in which modules receive information as well as forward modular output to other modules. This approach is novel because it allows large volumes of evidence to be processed automatically by a preset number of modules.

We thought this idea was a good approach for the unskilled user. However, in attempting to setup and use OCFA, we found that a skilled user is definitely required. The installation process has several dependencies that are not well documented. Once the product is installed, an administrator needs to setup a different web site for each case. Each case may have many different images entered as evidence. To enter these images an administrator must copy the files to the machine on which OCFA is running, and run an ingestion script from the command line. Only after the image is processed, is the data mode available to the investigator via a web interface.

By involving an administrator or other skilled user, OCFA made the process much harder to deal with and therefore much less user friendly. Furthermore, entering evidence into a case does not import any metadata relating to the physical media making it difficult to determine if two of the same pieces of media have been entered twice.

In short, OCFA processes an image and makes the results available to the user. While this is valuable in the context of DOMEX, OCFA does not take any extra steps to analyze the contents of the extracted files and report their significance to the user.

2.2.2 Access Data's FTK 2.0

The Forensics Toolkit (FTK) is a forensics investigation tool released by Access Data[6]. It ships as a Microsoft Windows application that allows an investigator to acquire and browse images of media. As with many other forensics tools, it has a litany of features that range from data recovery to distributed cracking of password protected documents.

The difference between FTK and other forensics tools is that FTK performs an extensive automatic analysis on an image upon its introduction to the system. When an image is introduced into the system, FTK performs a background analysis and creates a list of all encrypted and deleted files. Additionally, an investigator can set FTK to look through the media's free space and attempt to put together files that were once written to disk (known as file carving). File carving is different from recovering deleted files in that it looks to known file signatures to reconstruct files rather than looking to file system data structures that are marked as deleted. This allows an examiner to recover files which have been deleted and for which no file system data structures exsist. Additionally, FTK is able to farm some of these processor intensive tasks out to worker nodes.

These features are not enabled by default. However, once the investigator enables them FTK automates the processes without further user input. Automatically processing an image to extract useful data is one of the key concepts in automated DOMEX and is explored heavily in Garfinkel's vision of DOMEX[15].

However, FTK falls short of fully realizing Garfinkel's vision in several areas. First, FTK requires a skilled operator to both extract and make sense of the data on the hard disk. FTK goes a long way to automate the extraction of the data, but the examiner must still be skilled enough to kick off the process. Even after FTK has finished extracting data from the image, it does not point out files with relevant content to the investigator or attempt to link files with data from other cases.

Both of these points contradict Garfinkel's vision of DOMEX in that he believes "automated software should be able to perform analysis that's at least as thorough as an analysis created by one or more humans[15]." FTK does implement some functionality that moves towards this vision, but its reliance on an investigator stops it short of becoming a full-fledged automated DOMEX system.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3: Our Vision for a Media Acquisition System

3.1 The DOMEX Environment

Today's DOMEX environment is one of many different digital sources spread across multiple physical locations. These digital sources are encapsulated on many different types of digital media which may be found, bought, seized, or surreptitiously borrowed by a variety of personnel. We would like our DOMEX acquisition system to be as automated, quick, and error-free under most if not all of these aforementioned operating conditions. We believe that in doing this, we will not only enable a greater depth of analysis in the DOMEX process, but we will also encourage use of the acquisition system in more environments.

3.2 General Architecture

We envision a distributed and automated DOMEX system, call it ADOMEX, that has a large number of networked field media acquisition nodes that can be used to acquire media contents as close to the point of origin as possible. In this case, we take "acquiring media contents" to mean the acquisition of a forensic quality image[20] of the device and its applicable metadata.

We feel that these distributed field acquisition nodes best match the distributed nature of the information that we seek to exploit. In being distributed, we can allow the organizations making use of the devices to decide the best way to acquire information for ADOMEX. For example, if a military commander feels it is important to be able to introduce information into a ADOMEX system as devices are being seized, she might place field acquisition nodes with forward deployed units. Conversely, an intelligence agency concerned with secrecy might decide to place acquisition units at secure locations and have media introduced into the system there. In any case, a system with distributed field acquisition units will allow the most flexiblity in terms of how an organization wishes to deploy our system.

These field acquisition nodes would then feed into a larger central repository where device images would be further processed with the results being stored for analysis. These stored results would be used both directly and indirectly in the analysis process. Direct analysis would be defined as analyzing information about or directly related to the piece of media. This would consist of things like recovering and analyzing deleted files on the media, collecting information

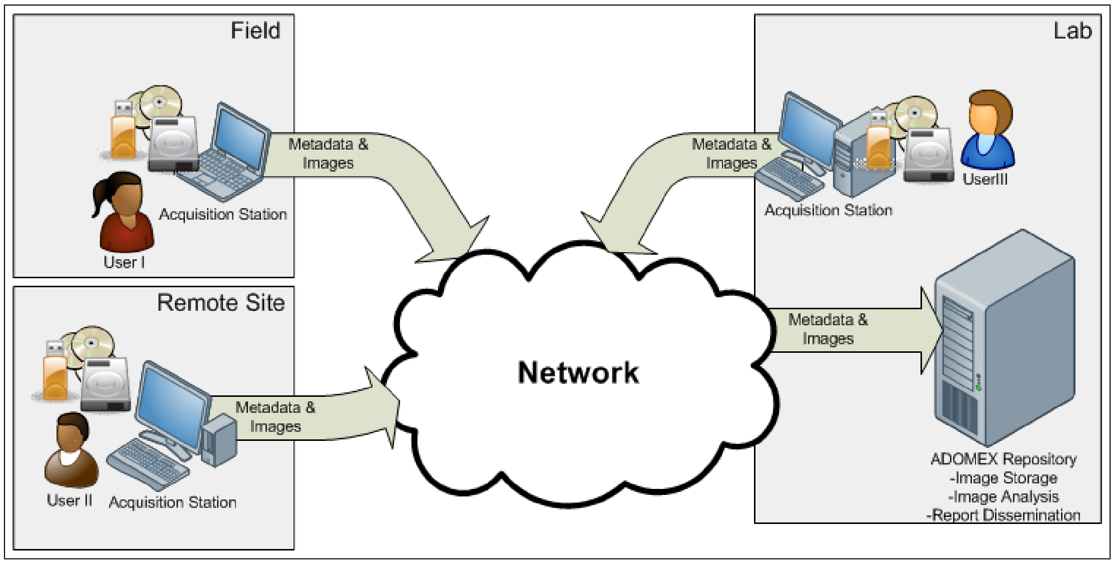

Figure 3.1: A sample ADOMEX architecture with several acquisition stations and a repository.

about the media, and comparing the media to other media in other repositories. Indirect analysis would consist of all analysis where the media in question is not the primary focus.

While the central repository would hold the results of all the analysis, some mechanism would need to disseminate the results of the analysis back to those who could benefit from its information. The aforementioned dissemination mechanism is out of the scope of this thesis.

For the purposes of this thesis, each field node is responsible for collecting device images and its accompanying metadata and pushing it to a ADOMEX repository. As such, we lay out our requirements for a ADOMEX media acquisition system that is automated, quick, and error-free. The requirements are as follows:

1. The acquisition system must be operable by a person with minimal technical training.
2. The acquisition system must automatically acquire images from as many common media types as possible.
3. The acquisition system must keep track of which devices have been imaged (globally) and refuse to create images of the same information on the same device.
4. The acquisition system must allow the user to insert notes and data relating to the circumstances surrounding the device.

5. The acquisition system must intelligently upload a resulting image to a central repository for processing. This upload must be able to handle poor network connectivity.

3.3 Operation by Unskilled Users

One of the main thrusts of DOMEX is to quickly extract actionable intelligence from a given piece of captured media[15]. As such, we would like to make our acquisition system accessible to all front line personnel who may come across digital media in the course of their operations.

We can make very few assumptions about the users on the front lines due to their diverse roles and talents. In fact, their expertise may lie in fields that are not associated with a deep knowledge of computing let alone digital DOMEX techniques. Making the DOMEX environment accessible to these people is a key goal for us because we believe they are often the ones left holding a captured USB flash drive and wondering "what kind of intelligence can I get out of this thing?" Additionally, we believe these front line people are in the best position to add valuable context and metadata to seized media.

We purpose an interface which requires very little user interaction. In our minds, a user should simply have to find the right connection to plug a device into and the system should handle the rest. The only other thing the system should ask of the user is any information which it can not possibly extract from the device itself such as the circumstances under which the device was obtained.

When we say the system should handle the rest we mean that the system should perform some sequence of operations that end with informing the user of all the intelligence on the captured device. That being said, this thesis focuses entirely on extracting an image from a captured device so that it can be sent off and mined for intelligence. We want to make the acquisition process as low interaction as possible. In doing this, we believe that we can increase automation of the DOMEX process in general, thus the ADOMEX moniker. We believe that increased automation of the DOMEX process will allow users to spend more time consuming accurate information instead of processing, translating, and analyzing it.

We will discuss how we went about designing a low interaction imaging process in Chapter 5 which describes our implementation of a media acquisition system for ADOMEX.

3.4 Acquisition from Diverse Media Types

An ADOMEX acquisition system may be operated all over the world. We must assume that the system's users are as likely to see a 3.5 inch floppy in Afghanistan as they are a USB flash drive in Paris. Obscure media may require rare hardware and even then may not be supported by several parts of a ADOMEX system's software. If an acquisition system is unable to acquire information from a piece of media, that information is essientially lost to the ADOMEX system.

We aim to create a system to which media support is modular. This means that given a new type of media, we would be able to add a module to the ADOMEX acquisition system that allows us to support that new media with minimal effort. With this modular approach, we can focus on implementing support for the most common media types first.

"Implementing support" in this case means that the system should be able to read all of the information off of the device in question. This information includes not only the contents of the device but the devices internal diagnostics and identification information (e.g. serial number, model, manufacturer).

3.5 Intelligent Allowance of Duplicate Images

ADOMEX operations could range in size from a few localized users who seldom acquire device images to a large group of users who often acquire images with less than perfect communication. In a potentially large effort such as this, avoidance of duplication of efforts is often desirable. However, if a device is captured on different occasions it may also be useful for the ADOMEX system to note changes since the device was last seen.

As such, the acquisition piece of the ADOMEX system should only allow images to be made and uploaded to its central repository if they do not exist already. The acquisition system's duplication detection must be intelligent enough to know the difference between like devices containing slightly different data and like devices containing exactly the same data. In the former case, the acuquisition system must allow several images of the device to exist. In the latter case, the acquisition system should inform the user that the data already exists in the ADOMEX system and an image of the device is unessacary.

As discussed earlier, ADOMEX acquisition systems may be widely distributed or centralized in a lab. Because of this, we can not make any assumptions about constant network connectivity to the central repository. This complicates the problem of the aforementioned duplication detection in that acuquisition nodes may not always be able to connect to each other or a central respository to determine if a device has been imaged.

3.6 User Generated Metadata

In a ADOMEX environment the time, place, and circumstances in which the media was captured could have as much significance as the data on the media.

The ADOMEX acquisition system should give the user the opportunity to annotate the image's metadata with as as many details as possible regarding the media's provenance. The acquisition system should save this information, with the image of the device to minimize the chances that they will become separated or mismatched.

This raises some issues. First, the system should require as little interaction as possible. Asking a user to input data is in conflict with this goal. Next, there is the problem of making sure the user input is useful without being restrictive. We do not want the input prompts of our acquisition system to goad a user into entering dummy values.

There are several design decisions to be made here:

1. Should the input be structured or unstructured?
2. If it is structured, what kind of values should be required?

3.7 Intelligent Upload to a Processing Repository

One guiding principle of the NPS ADOMEX research project is that the information on a given piece of media is better understood by considering it alongside other information in the ADOMEX repository. Much of our vision of a ADOMEX architecture depends on having access to all data in one place. In this way, an analysis may attempt to link contents found on different images with the image in question.

One way to enable this sort of analysis is to make the acquisition tool efficiently transmit media images or information extracted from those images to the central repository. We say efficiently here because we can not make assumptions about the network connections of our acquisition stations. Therefore, the system needs to be able to make decisions about whether a network is available and suitable for transferring large device images. Once the network conditions allow for an image to be uploaded, the system should verify that a duplicate image isn't present and proceed to upload the image to the server.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4: Evaluation of Device Acquisition Tools

4.1 Tool Evaluation Rubric

There are many tools out there which exist to acquire images of all sorts of digital media. These acquisition utilities fall into two broad categories: general purpose copying tools and forensics device acquisition tools. Forensics acquisition tools are usually created for the purpose of capturing digital device images in a way that preserves their evidentiary integrity. On the other hand, general purpose copying tools are created to make a working copy of the device and have no special focus on preserving every bit present on the device.

In this chapter, we discuss some of these tools and how they might be used in an ADOMEX environment.The tools were chosen for their ability to fufill all or some of the requirements mentioned in Chapter 3. Additionally, we only review open source tools because we later wish to adapt a tool that meets our requirements into a full fledged acquisition system. We do not feel that we will be able to accomplish this without access to the tool's source code.

Specifically, we ask several questions of every tool:

1. Can the tool be used by a person with minimal training?
2. Does the tool allow one to acquire images from many types of media?
3. How would the tool handle imaging and tracking many pieces of media with very few repetitive steps?

In this context, a person with "minimal training" means a person who is familiar with GUI navigation and connecting various types of media to a computer system, but has no specific training in computer forensics.

The tools were selected for evaluation based on our vision of how they might be used in the aforementioned architecture for an ADOMEX acquisition system. We tried to select tools that would be usable as a small or large part of the ADOMEX acquisition system. Using the aforementioned questions we dicuss each tool. These questions are meant to see how well a tool might address some of the requirements we have for our acquisition system.

Requirement	dd	DF[a]	aimage	EA[b]	AIR	pE[c]	GFI[d]	Guymager
Operation By Unskilled User	No	No	No	No	No	No	Yes	Yes
Diverse Media Acquisition	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Duplicate Image Handling	No	No	Partial	Partial	No	No	Partial	Unknown
User Generated Metadata	No	No	Yes	Partial	Yes	No	Partial	Partial
Intelligent Image Upload	No	No	No	No	No	No	No	No

[a]DCFLDD
[b]EWFAcquire
[c]pyEWF
[d]Gnome Forensic Imager

Table 4.1: Features of disk imagers

4.2 Tools

In this section we will review multiple device acquisition tools that have the potential to be of use in the ADOMEX environment. These tools are split into two main types: command line and GUI tools. The command line tools can be defined as ones that to not have an interactive system of graphical menus by which they are operated. The graphical tools are operated through graphical menus and are, in this case, usually frontends to certain command line acquisition tools.

Table 4.1 summarizes our finding as to how the tools stood up to our requirements in Chapter 3:

4.2.1 Feature Summary of Image Formats

All of the tools we review support at least one of the following image formats: raw, Encase Expert Witness Format, or the Advanced Forensics Format (AFF). Each of these formats supports some combination of the following features:

- **Image Compression**
  The ability of the format to natively seek and store compressed image files.
- **Image Encryption**
  The format supports contents encrypted with either symmetric or asymetric key cryptography.
- **Image Hashing**
  The formats ability to store cryptograhic hashes of the image contents in the image file itself.

Feature	raw	EWF	AFF
Image Compression	No	Yes	Yes
Image Encryption	No	No	Yes
Image Hashing	No	Yes	Yes
Metadata Storage	No	Yes	Yes
Arbitrary Metadata Storage	No	No	Yes

Table 4.2: Feature support in image formats

Feature	dd	DCFLDD	EWFAcquire	Aimage
Formats Supported	raw	raw	EWF, raw	Raw, AFF
Image Compression	No	No	Yes	Yes
Image Encryption	No	No	No	Yes
Image Hashing	No	Yes	Yes	Yes
Metadata Storage	No	No	No	Yes
Arbitrary Metadata Storage	No	No	No	Yes

Table 4.3: Feature summary of command line acquisition tools.

Feature	Air	pyEWF	Gnome Forensics Imager	Guymager
Formats Supported	raw	EWF, raw	EWF, raw	EWF, Raw
Image Compression	Yes	Yes	No	Yes
Image Encryption	No	No	No	No
Image Hashing	Yes	Yes	Yes	Yes
Metadata Storage	No	Yes	Yes	Yes
Arbitrary Metadata Storage	No	No	No	Yes

Table 4.4: Feature summary of GUI acquisition tools.

- **Metadata Storage**
  The ability of the format to store case details, image hashes, and other metadata inside of a device's image file.

- **Arbitrary Metadata Storage**
  A feature which allows the format to store arbitraily defined pieces of metadata in the device's image file.

While each format supports the features summarized in Figure 4.3, it is up to the tool to implement them. Figures 4.3 and 4.4 detail the tools that support each imaging format as well as what features of the format the tools implement.

```
dd if=/dev/sdx of=/tmp/image
Where:
 /dev/sdx      the file/device to be imaged
 /tmp/image    the file to write the image to
```

Figure 4.1: A typical dd command line that will not work for imaging.

4.2.2 Command Line Tools

dd

dd is a tool that allows one to copy data at the lowest levels. It was originally meant as a utility for making sector for sector copies of files and was not designed for computer forensics usage. However, the tool is often used by forensic examiners to create duplicate copies of seized media for analysis.

dd has several options and features but in its simplest form it needs a source from which to copy and a destination which to write to. In cases where one wishes to make an image of a device, the source is the block file that represents the device. The corresponding destination is usually a file where the user wishes to have the device contents written. In more complex runs, dd can be made to continue on in the face of errors or convert data into different encodings. A full manual page is available on DD[21] but for the purposes of this thesis we are concerned with using dd to create images of devices in an ADOMEX environment.

It is helpful to walk through a typical dd usage scenario to illustrate how dd might be used in an ADOMEX environment.

Let us consider a common usage scenario where an examiner has just received a seized USB flash drive. As he wishes to use dd to image the data for analysis he must do several things. First, he must have a system with dd and enough disk space to hold an uncompressed image of the USB flash drive. If he does not have enough free space he must either delete files or pipe dd's output through a compression utility such as bzip. Next, he must insert the flash drive and determine where the computer has mapped it to. After he has determined what the computer has mapped the device as, he must then determine the appropriate parameters to pass to dd on the command line. Such a command line appears in Figure 4.1.

Assuming this command completes without errors, the examiner will then have an unverified sector for sector image of the media. However, if the command errors out because of a bad

```
dd if=/dev/sdx of=/tmp/image conv=noerror,sync
```

Where:

`/dev/sdx`	is the file/device to be imaged
`/tmp/image`	is the file to write the image to
`noerror,sync`	Ignore device errors and pad the unreadable blocks in the image with nulls

Figure 4.2: A dd command line that will properly handle a device with errors.

piece of media, the examiner would have to make a few adjustments. He would have to tell `dd` to ignore errors and instead copy null bytes into the image where the unreadable bytes would be. The command is shown in Figure 4.2

The `dd` command structure requires a user to be aware of which block file the target device is mapped to. Additonally, `dd` requires the user to specify and make sure that the destination file is not already in use. Because of these requirements, `dd` is not the most accessible tool to the unskilled user. It is uncommon for unskilled users to have the knowledge to identify a device mapping, have them be able to build the appropriate command, and handle any errors that they may encounter. A skilled user could create some sort of wrapper for `dd` which insulates others from being burdened with the complexities of `dd`. We will see some such tools later in this chapter as we review the GUI acquisition tools.

While `dd` is not the most accessible tool, it is extremely versatile. `dd` can be used to copy any piece of media that can be mapped to a block device. It is often used in this manner to provide a quick and dirty copy of media which more specialized tools may not be able to provide.

Even though it is versatile, `dd` lacks several functional components that make it suitable for imaging devices in an ADOMEX environment. First, it does not record metadata of the imaged device. This metadata consists of both user generated facts about the device's seizure and facts about the device's make, manufacturer, and serial number. This information is almost indispensable in relating physical media to a captured image. Next, `dd` does not compress, encrypt, or calculate the hash of the captured image. These features are important in a large-scale operation in that they allow one to verify that images have been stored correctly (hash values) and safely (encryption). Also, hash values and metadata would allow one to quickly compare images already in the DOMEX system to the device being imaged. This information would allow for duplicate images to be avoided as well as provide some information as to where a device has been seen before.

The lack of metadata collection features puts the onus for collecting useful tracking information on the user of the tool. Not only does this generate more repetitive steps per device, but it also makes the tool less suitable for large scale use and error free use by a minimally trained user.

Also, `dd` does not support the entry or storage of user generated metadata about the device. If one wished to store notes and information about the device's seizure, they would have to create their own system for doing so. In our requirements section we detailed the importance of user generated metadata and we feel asking the user to create a system for doing so is unreasonable.

DCFLDD

`DCFLDD` is an extended version of `dd` written by Nicholas Harbour in 2002 and aimed specifically at computer forensics examiners and security practicioners[3]. In addition to all the functionality of `dd`, hashing and verification of the image is integrated in the program. These hashes can be printed to standard error or they can be saved to a file.

While the integrated hashing relieves the user from performing another operation against the image, `DCFLDD` still has a few of the limitations of `dd`. First, `DCFLDD` must be operated from the command line and therefore requires a user who is comfortable with managing the imaging process via the command line. As previously discussed, the command line is not the ideal interaction device for untrained user. Next, `DCFLDD` calculates hashes of the collected image but leaves their storage up to the user. Again, this places the responsbility for storing this data on the user which is not an ideal practice in a large scale ADOMEX operation.

Like `dd`, `DCFLDD` does not provide facilities for the storage of user generated metadata and therfore has the same short coming as `dd` with respect to user generated metadata. Additonally, `DCFLDD` lacks facilities for compression, encryption, and device identification.

EWFAcquire

`EWFAcquire` is a a tool written by Joachim Metz that allows one to acquire device images into the Expert Witness Format or EWF[9]. `EWFAcquire` uses the LibEWF library to write EWF formatted files. The EWF format is largely associated with the Encase forensics tool which is used primarily in law enforcement.

The tool allows one to image devices much in the same manner as `dd`. Unlike dd, `EWFAcquire` can both compress and calculate the hash of an image as it is being acquired. It also has the ability to store both the hash and the compressed image in the EWF format. The EWF format includes provisions for storing case related metadata within the file. Unfortunately, this functionality is curiously absent from `EWFAcquire` even though the functionality is provided by LibEWF.

`EWFAcquire` has the same command line complications that `dd` has. Expecting an untrained user to be able to read manual pages, build a command line, and image several devices using `LibEWF` without error is simply unreasonable. Additonally, `EWFAcquire's` inability to write metadata to the image file requires that the user both obtain a tool that is able to write to the EWF format and input metadata separetly from the imaging process.

Aimage

`Aimage` is an imaging tool written by Simson Garfinkel which is built around AFF or the Advanced Forensics Format[4]. AFF is a file format which stores both a device image and its metadata in a single file. `Aimage` is a command line utility that allows one to specify one or more input devices to be imaged to one or more AFF files. In addition to creating AFF files, `Aimage` can also compress and encrypt the contents of the device on the fly.

There are many features to `Aimage` which make it useful in many forensic imaging situations. However, this thesis is concerned with how well `Aimage` fits into our envisioned acquisition system and how well it can address the questions of this chapter.

First off, `Aimage` provides slightly more ease of use to the untrained user than `dd`. Since `Aimage` was designed with imaging devices in mind, it is designed to be fault tolerant[16] and therefore should not require user intervention as often as `dd`. However, `Aimage` still requires that an untrained user make a foray into the somtimes daunting command line. Like `dd`, `Aimage` requires that the user know what block device his target media is mapped to. From there, the user has to build a command line that slightly resembles that of `dd`: "aimage (1)/dev/sdb (2)image.aff"

Where 1 is the block file of the device to be imaged and 2 is the destination image file.

4.2.3 GUI Acquisition Tools

AIR - Automated Image and Restore

The dd family is entirely command line based. Realizing this and the problems it presents to both the unskilled and batch users, Steve Gibson created the AIR GUI[2]. The GUI attempts to provide a wrapper around dd/DCFLDD extending it with network support and automatic media detection.

In using DCFLDD as its workhorse, AIR is able to include many of its important features such as device imaging, image hash calculation, and graceful error handling. This makes AIR able to acquire images from anything that can be seen as a file under a unix based operating system giving the tool all the device versatility of DCFLDD.

The automatic media detection is perhaps the most interesting feature of AIR. When the program starts up it goes through the /dev directory, a listing of all the devices, and determines which block devices are on the system. This list of valid block devices is shown in the main GUI window along with an icon representing the type of device (hard disk, cdrom, etc). Clicking on one of these devices allows one to get more information about the device or set it as the source or destination of the image. This feature, along with the ability to graphically control the parameters to DCFLDD, gives AIR an advantage in ease of use over the command line interface of DCFLDD. This advantage is significant in that one of our requirements for an acquisition system is that it be operable by an untrained user. AIR seems to be the first effort to bring the dd family of tools from the command line to the GUI thus making them accessible to a wider range of untrained users. This accessibility comes from the fact that many of DCFLDD's options are presented as a series of visible checkboxes and are not hidden in a man page or command help.

AIR does not record the serial number, model, manufacturer, or other metadata about the device being imaged. Furthermore, AIR requires an operator to image one device at a time. A user has to wait for the first device to finish imaging before he may try another. This requirement is not conducive to imaging the large amount of media often seen in the ADOMEX environment because an examiner can not queue several connected devices for imaging and walk away.

pyEWF

pyEWF is a console interface to EWFAcquire that attempts to take the command line difficulties of LibEWF's utilities and put them into a menu-driven format. Although this interface may not be considered a GUI, it is menu driven and does not require command line parameters. For these reasons, we have placed it in the GUI section. The interface is simple and provides the added benefit of being able to detect which devices are plugged into the system. However, pyEWF will not update this list without an manual request by the user. This becomes a repetitive step in that whenever a user inserts a new device he must tell the interface to scan for new devices. pyEWF's simple interface provides access to every feature that LibEWF's command line tools do. In this regard, pyEWF is a step in the right direction in satisfying our requirements.

While simple, the interface has several issues which make it unsuitable for usage in an ADOMEX environment given our stated usage. We counted 12 repetitive steps to begin the imaging process. Additionally, the interface does not allow an examiner to start or queue several connected devices for imaging. This makes the interface unsuitable for an ADOMEX environment in which and acquisition system needs to be able to handle the imaging of a large number of devices with little human interaction.

As mentioned, the EWF format has provisions to save certain pieces of metadata into the image file. pyEWF provides the user with an easy way to do this as it prompts them for their name, case notes, case number, evidence number, and other notes upon a request to image a device. Unlike EWFAcquire, pyEWF provides the facilities for a user to enter metadata and then passes the data to LibEWF which is responsible for writing it to the image file. Unfortunately, pyEWF does not offer to autofill some of these fields and therefore creates repetitive steps for a user. Additionally, it is unclear as to what metadata pyEWF collects and stores about the device itself. pyEWF's insistent metadata is a step in the right direction to be sure, but it seems so repetitive and insistent that a user might end up ignoring it.

Gnome Forensic Imager

The Gnome Forensic Imager is written by Joachim Metz and it offers yet another GUI for LibEWF. Gnome Forensic Imager is still under development and has not been released to the general public yet. However, the author provided us with several screenshots which detail the interface and its functionality.

The Gnome Forensic Imager uses `LibEWF` for imaging devices but it also uses the Hardware Abstraction Layer under Linux to discover block devices. It is unclear whether it scans for these devices on user request or detects the presence of the devices as they are inserted and removed. The interface offers several improvements on `pyEWF`; it allows a user to queue up several devices for imaging; it is less insistent about metadata entry and allows a user to begin imaging a device in as little as three clicks; and the tool records the model, manufacturer, and serial number of devices selected for imaging. `pyEWF` only reports the block device name and the size of the device to be imaged.

The Gnome Forensic Imager covers some of the requirements in Chapter 3 such as metadata collection, imaging from many devices, and accessiblilty to untrained users. However, the provided screenshots give us limited ability to assess and verify its true capabilities.

Guymager

`Guymager` is a forensic imaging tool authored by Guy Voncken[7] which allows one to acquire images in raw (`dd`) format or Encase's EWF format. According to the documentation, it strives to be a user friendly interface that takes advantage of multi-core hardware to parallelize tasks such as image checksumming, compression, and actual image acquisition.

The imager seems to succeed in its goals of being a user friendly imager. Unlike most of the graphical imaging utilities, `Guymager` does not require very much technical information from the user. The main window contains a list of devices from which the user can choose to image. This device list is not automatically updated on device insertion. However, the list contains identifying information for each device such as the serial number and device model. When the examiner wishes to image a device, he must right click on it and hit the acquire button. This leads to another dialog which allows the examiner to choose between output formats for the image. After the user enters this information, he only needs to click acquire. Unlike `AIR` and other frontends there are no options reflecting special parameters to `LibEWF` or `dd`. While this may limit the investigator in specialized circumstances the lack of options make the task a easier for the untrained investigator.

Based on the output format chosen the investigator is allowed to input some metadata to be stored with the file. In addition to the limited metadata stored in the image file, `Guymager` stores additional summary reports about the image in a file with a different extension.

Additonally, `Guymager` attempts to automatically name the image file with the model and serial number of the device. This is useful for two reasons. First, it allows the investigator to image more devices with fewer clicks. Second, it provides a rudimentary tracking system for which devices have and have not been imaged. This feature starts to address the problem of tracking which media has been imaged, but it is not a complete solution. If multiple acquisition stations were being used, the filename method used by Guyamger would not be robust enough to protect against multiple examiners imaging the same media. Additionally, `Guymager` does not protect (or warn) against overwriting existing files which would allow the same device to be imaged over its existing file. While there is nothing technically wrong with imaging the same device multiple times, it seems like a waste of resource and therefore is something that should be avoided unless the information on a device has changed.

While the tool can handle imaging a device with very little user interaction (three clicks minimum), it is not our ideal batch imaging tool because it lacks the ability to track multiple images.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5: Implementation of a Large Scale Media Acquisition System

5.1 Introduction

Chapter 3 presented the requirements for a large scale media acquisition system. In this chapter we detail our efforts to satisfy these requirements through a frontend we called AcqMan. We were not able to implement all of the functionality we believe is required for a complete ADOMEX acquisition system. However, we describe what we have implemented and detail designs for the features we would like to implement.

5.2 A Piece of the ADOMEX Architecture

The acquisition system we have designed and implemented is the piece of the ADOMEX architecture that captures media, images it, and uploads it to the ADOMEX repository for analysis.

The following steps and accompanying diagram detail a typical device acquisition process with AcqMan and serve to give a sense of where AcqMan fits in the larger ADOMEX picture.

A typical device acquisition might resemble the following:

1. A user connects a media device to the acquisition system while AcqMan is running.
2. AcqMan recognizes the new media and gathers metadata including model, manufacturer, and serial number.
3. Once this metadata is gathered, AcqMan uses it to query an ADOMEX repository for devices matching the inserted device with existing images.
4. If the device has not been imaged or the ADOMEX repository can not be contacted, the acquisition station begins the imaging process. If the device has been previously imaged, the ADOMEX repository informs the acquisition station another images is needed to determine if the contents of the device have changed.
5. While the device is imaging, the user is prompted for information about the device's origins and circumstances.

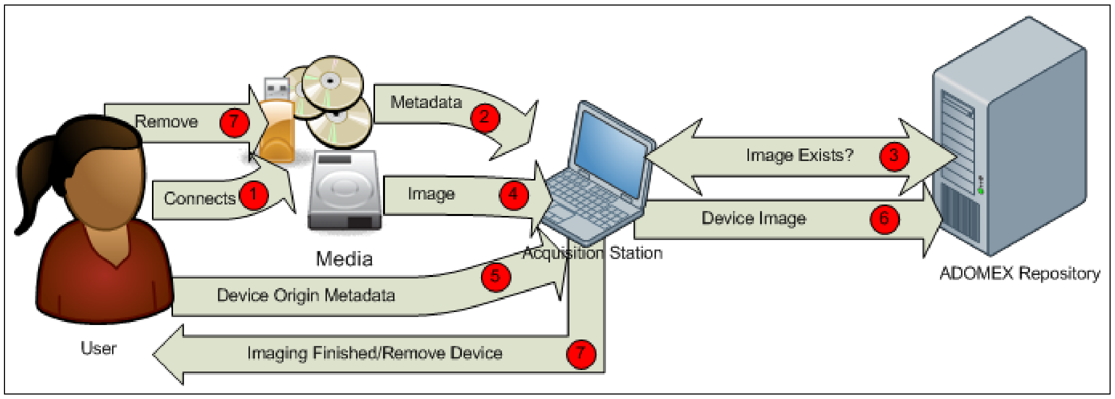

Figure 5.1: Steps of a typical device acquisition under AcqMan.

6. Upon completion of the imaging process, the image is uploaded providing the device had not been previously imaged. If the device was previously imaged, the acquisition station queries the ADOMEX repository to determine if the existing and current images are different. If they are, it uploads the local image to the repository which keeps both images.

7. The front end informs the user that the device has finished imaging and can be removed. If necessary, the device is wiped before removal.

8. The user removes the device and repeats the process as necessary.

Figure 5.1 shows how the system should progress through each step. In the next section we detail the design and implementation of each step in the acquisition process. Additionally, we give a detailed description of the mechanisms that tie the steps together.

5.3 Technologies Used

AcqMan integrates several existing technologies to take advantage of the work that has been done in device insertion detection, remote procedure calls, and forensic imaging.

5.3.1 Free and Opensource Unix Based Operating Systems

We developed our acquisition frontend primarily for Unix based operating systems. These operating systems offer us many advantages that are not available in Windows such as fine grained control over device mounting, immunity to a majority of viruses (which are designed for Windows), and easier development. The frontend should be able to run on any Unix-like OS that supports Java 1.6, D-Bus, HAL, and `Aimage`. Operating system distributions that support these dependencies include but are not limited to FreeBSD, Slackware Linux, Fedora Linux, and others.

Acquisition Appliance

We envision AcqMan to be a frontend which is run as the dominant application on purpose built acquisition hardware. An opensource Unix based operating system give us the ability to easily tailor both the operating system and its boot sequence to facilitate the building of these purpose built systems. This appliance would be tailored around an organizations needs an may take the form of a rugged device for the military or a small device for discreet operations. Designing and deploying the physical acquisition station and its operating system is out of the scope of this thesis. However, it is important to develop AcqMan so that it can be used in this environment.

5.3.2 Java 1.6

Java was picked as the language of choice for this frontend primarily for its portability. Writing the frontend in Java allows it to be run on any machine providing it has a Java 1.6 JVM and all the other dependencies listed below. Also, Java gives us the ability to have a portable graphical interface that looks the same under any operating system and window manager. Additionally, Java offers us the ability to distribute our application in a single package without the need for an install process. This allows us to distribute the package for use on systems which users may not have administrative access to. With this package, they are able to run the acquisition frontend as a non-privileged user providing that they are able to read the block files representing the device.

Finally, Java has packages available to interface with all of the other technologies we mention below. This is crucial in our frontend because it saves us time and effort and allows us to leverage existing software for imaging, device detection, and remote procedure calls.

5.3.3 D-Bus

D-Bus is a system for sending messages within the context of a single machine and it runs as a system daemon. It gives programs a way to talk to one another and is therefore an inter-process communication mechanism. The application, specification, and documentation are all maintained and distributed by Freedesktop.org; D-Bus is available for many Unix like operating systems including Linux, FreeBSD, and Mac OS X. A Windows port of D-Bus is in the works, but it does not conform to the freedesktop.org D-Bus specification at the time of this writing[12].

Applications can attach to a D-Bus through a socket located in the file system. The administrator has the ability to limit access to the system wide bus by restricting the file permission on the socket or by defining a security policy in the D-Bus daemon configuration files. The security policy can specify who is allowed to connect, what they are allowed to do on the D-Bus, and define what buses they are allowed to access[19]. D-Bus supports authentication upon a connection to a bus to map a security policy to a specific user or class of users. However, by default, this authentication mechanism is only used if D-Bus is made available over TCP.

There are two types of D-Bus buses: a single system wide bus and many per-login buses. The system wide D-Bus is started as the machine boots and provides a way for system wide daemons and the kernel to get messages to applications. Messages on the system wide bus would include things like device insertion notifications or network availability notifications[11].

The per-login instances of D-Bus are usually used for inter-process communications between a user's desktop applications. For example, an VoIP client might send a message about an incoming call to a music playing application so that the music can be paused.

The D-Bus protocol[19] and API[5] have been standardized with the promise that all future versions "will either be compatible or versioned appropriately[11]". Using this mechanism allows for us focus on the content of the D-Bus messages we are interested in rather than how to receive and process these messages.

We leveraged a package called dbus-java[17] by Matthew Johnson. This package allows us to attach to a system wide D-Bus from our Java Frontend. In doing this we are able to receive messages from anything on that system bus which include hardware insertion messages as well as messages about network connectivity. Being able to receive and act on these messages is a key part of our acquisition system.

5.3.4 Freedesktop.org's Hardware Abstraction Layer

The Hardware Abstraction Layer, or HAL, is a product put out by freedesktop.org which is available for most Unix-like platforms. For the most part HAL seeks to "make hardware just work"[8] by providing a standard API on top of the kernel which reports information on all system devices. This allows for one to gather information on installed hardware, detect the insertion/removal of new hardware, and ask the kernel to eject removable hardware in a fashion independent from the underlying kernel. HAL should work the same regardless of which kernel it sits atop. This thesis is concerned mostly with the device information and the events that HAL is responsible for generating.

The information that HAL reports on can be broken up into two categories: device information and device events. Device events are actions that occur on a device (e.g. a network cable is plugged in) or the insertion/removal of a device.These events are broadcast onto the system-wide D-Bus that we mentioned earlier. Any program attached to the system-wide D-Bus can receive and process these event messages. This structure provides us a mechanism to use in our frontend for detecting and handling the insertion of new media in a kernel independent manner.

Device information can also be accessed via HAL. Device information includes such metadata as model, serial number, bus location, and block device. This information allows us to retrieve information about a newly inserted device with zero user interaction, and without being concerned with the underlying systems kernel.

HAL has several limitations:

- **Dependency on D-Bus**
  HAL requires D-Bus to work which constitutes as another dependency for our application. D-Bus became available in 2004, and as such can not be found in OS distributions before that time. Even if D-Bus is supported by an operating system distribution, it may not be installed by default which is the case in server focused OS distributions such as FreeBSD, Ubuntu Server, and Gentoo.

- **Dependency on the Kernel for Hot-pluggable Device Support**
  HAL requires support within the kernel for detecting new devices and automatically adding the kernel modules which support the device. In Linux, this requires a kernel version greater than 2.6.17. HAL kernel support is also available in FreeBSD 6.2 or later.

- **Dependency on the Kernel for Basic Device Support**
  HAL is entirely dependent on the kernel for device support. A device that the kernel does not recognize will not be reported to HAL. If the device is not reported to HAL there is no way HAL can generate an insertion event which our frontend relies on. As far as our acquisition system is concerned, the lack of an insertion event means the device does not exist. While there may be other ways to get at the device, they are highly device specific

and often require user intervention. Our acquisition system is meant to be fully automated with respect to device detection and imaging. Therefore, devices that do require this sort of manual intervention fall outside of the scope of this thesis because they usually require an operator who is trained in computer forensics.

- **Varying Device Metadata Support Between Kernels**
  Supposing the kernel is able to detect and support a device, it may not be able to gather the device's serial number, model, manufacturer, or other critical metadata. This is a sign of incomplete device driver within the kernel. We saw examples of these limitations during our development when we attempted to collect metadata from USB hard disks, SD cards, CF cards, and Sony memory sticks. These difficulties will be discussed later in this chapter.

- **Varying Device Insertion Detection**
  Incomplete device drivers also affect HAL's ability to report device insertion. In our development, we encountered several situations where certain USB media card readers were not reporting device insertion to the kernel. We will also detail these situations later in this chapter.

5.3.5 XML-RPC

XML-RPC[1] is a specification for performing remote procedure calls over a network using the HTTP protocol and XML coding for arguments and results[13]. Our frontend needs to interact with the ADOMEX repository in several situations. We chose to use XML-RPC as a mechanism to query information in the ADOMEX repository. XML-RPC allows us to expose several functions of the ADOMEX repository to the acquisition stations while hiding their implementation details. This level of abstraction gives us the ability to change a function's implementation within the ADOMEX repository without having to make sure implementation is mirrored in the client.

Additionally, XML-RPC implementations are available in languages such as Perl, Java, C, C++, and Python. XML-RPC's availability in these languages frees us from having to write all ADOMEX acquisition and analysis tools in a single language. For example, AcqMan is written in Java for the sake of portability between different environments while ADOMEX analysis tools that are focused on speed may be written in C. AcqMan depends on the ws-xmlrpc package for its interactions with the ADOMEX repository.

5.3.6 AFFLIB and Aimage

While `Aimage` can not used as a complete ADOMEX acquisition system, it has many features that make it well suited for use in one. These features allow us to fulfill the requirements laid out in Chapter 3 without recreating the functionality in AcqMan. `Aimage` has the ability to create an image, compress it on the fly, and store the related metadata in it. Essentially, these features allow our frontend to direct the creation and storage of images and in a space efficient format.

When Aimage images a device, the image is broken into several pieces called *pages*. By default Aimage uses 16 megabyte pages and compresses each one with the ZLib compression algorithm. Breaking the image up into compressed pages simplifies the upload process in several ways. First, the image is pre-divided which greatly simplifies resuming failed transfers. After a failed transfer, the system does not need to search the entire file to determine where it left off, it only need to compare the number of pages in the local and remote file and resume the transfer at the number of segments in the remote file. Next, the pages are pre-compressed which saves us the complexity of having to compress the image as it is uploaded. Uploading compressed pages saves bandwidth in an environment where bandwidth may be extremely limited. Last, the pre-divided image gives us the ability to easily upload only the changed pages in cases where a device has already been imaged once. This saves us from having to upload the entire image, which is crucial in a bandwidth sensitive environment.

Next, `Aimage` can image any device which appears as a file under the operating system. This directly contributes to our requirement of being able to image a large amount of different device types.

Last, `Aimage` has built-in mechanisms for error recovery and correction. This contributes directly to our goal of making our entire acquisition system accessible to users with minimal training. In case of an error on the device, `Aimage` will attempt to recover and get as much information off of the device as possible. This allows us to create a “best-effort” system where user intervention is not required.

5.4 AcqMan Architecture Details

The GUI frontend that we have written ties together several of the above technologies to form an acquisition system which fits the requirements set forth in Chapter 3. In short, this frontend

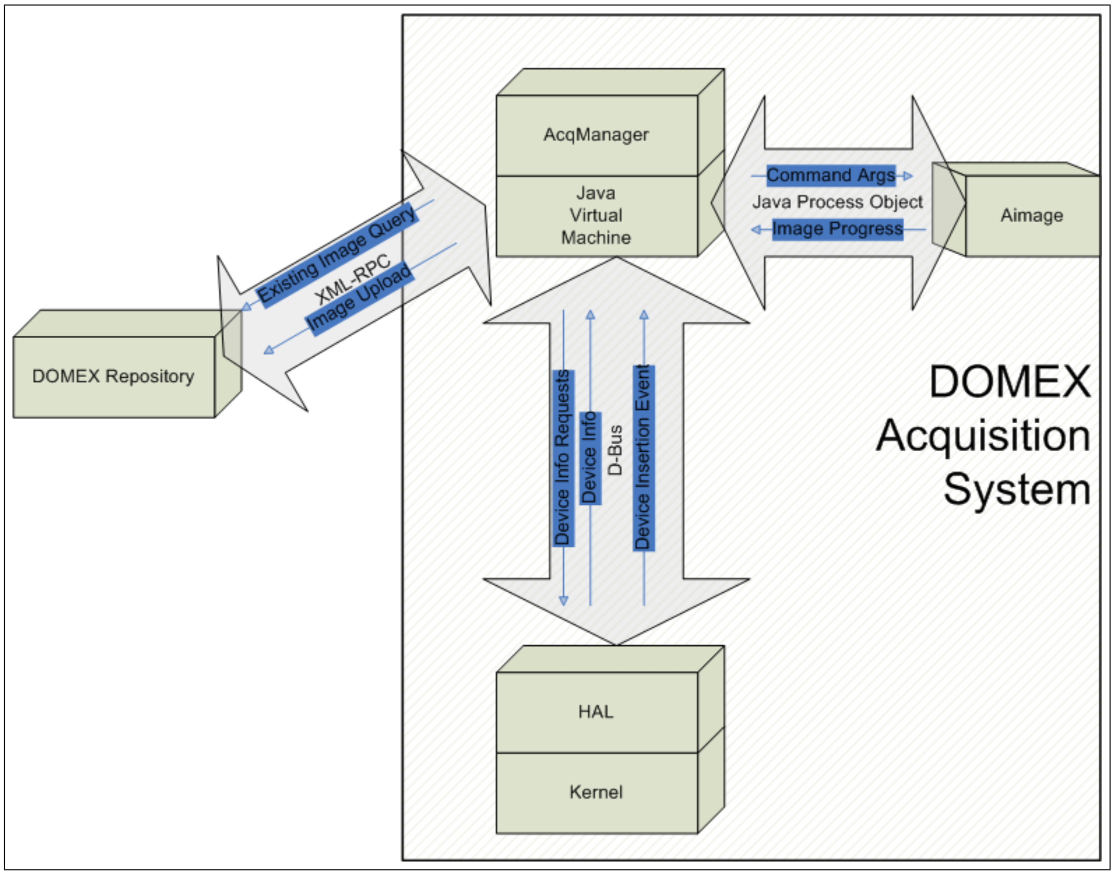

Figure 5.2: AcqMan's Interaction Diagram

should facilitate the image acquisition of many devices into the ADOMEX system. Figure 5.2 shows how AcqMan integrates with HAL, XMLRPC, the operating system, and Java.

AcqMan runs atop of the Java Virtual Machine. Through the JVM, AcqMan is able to use several communication mechanisms to tie into the acquisition station's HAL, the ADOMEX repository, and `Aimage`.

AcqMan is linked to HAL via D-Bus so that it can receive uniform device insertion and removal events as the kernel sees them. This link to HAL allows AcqMan to query any device on the system in order to gather its location and device metadata. All of this information allows AcqMan to detect an inserted device, query the ADOMEX repository for an existing image of the device, and start imaging the device if necessary.

AcqMan is responsible for deciding whether a device should be imaged as device insertion events come in from HAL (see section 5.4.4). If AcqMan decides an image of the newly-inserted device is needed, it invokes AFFLIB's `Aimage` in another process and monitors the progress. If `Aimage` is successful, AcqMan notifies the user that the imaging process is complete and that the device can be removed.

If multiple devices are inserted at once, AcqMan starts imaging the first two devices detected and queues the others. Upon completely imaging a device, AcqMan removes a waiting device from the queue and begins to image it. If no more devices are on the queue, AcqMan sits idle and waits for device insertion events.

5.4.1 AcqMan

AcqMan is a multi-threaded Java backend with a single threaded GUI frontend. There are three types of backend threads which split the responsibilities for:

- Waiting for and handling device insertion events.
- Checking if a device should be imaged.
- Imaging the devices.
- Uploading the completed images.
- Updating the status and providing feedback through the GUI to the user.

These threads will be detailed along with their responsibilities and communication mechanisms in the coming sections.

AcqMan's Internal Design

AcqMan classifies connected media in three distinct categories: queued for imaging, imaging in progress, and imaging complete or unnecessary. There are several threads within the application that are responsible for moving connected devices between these three categories and performing the requisite operations in each category.

Upon start up, the application connects to the system's D-Bus and attempts to find the HAL API. It does this through the "halHandler" class which is instantiated right after the GUI is rendered. During instantiation, "halHandler" sets up a D-Bus message handler through the "addSigHandler" method in java-dbus. This invokes the "halHandler.DevAddedHandler" method in its own thread whenever HAL sends a “DeviceAdded” message on the system bus. Having each “DeviceAdded” message spawn its own thread allows AcqMan to handle multiple device insertions at once. Every thread spawned by a “DeviceAdded” message is called an insertion thread within the context of AcqMan, the number of insertion threads is not limited by AcqMan.

Each insertion thread is responsible for collecting information for each inserted device, including its type (hard disk, memory stick, etc), model, manufacturer, serial number, and block file (i.e. /dev/sda). The insertion thread places all of this information in a “DeviceImageObject” which contains all the device metadata as well as the device’s status as it moves through AcqMan. Storing all of this information in a “DeviceImageObject” eliminates the need for other threads inside of AcqMan deal with the complexities of querying HAL again.

From there, the insertion thread queries the ADOMEX repository via XML-RPC to decide if the device should be imaged. If the insertion thread decides (detailed in section 5.4.4) that a device should be imaged, it adds the filled in “DeviceImageObject” to the “DeviceImageQueue”. The “DeviceImageQueue” is a thread safe, FIFO, Java queue data structure which represents all the devices within AcqMan that need to be partially or completely imaged.

If the device should not be imaged, it is placed in a Java Vector called “finishedList” and marked as an unnecessary image within its “DeviceImageObject”. At this point the job of the insertion thread is complete and it terminates.

When the frontend starts, it also starts two threads of the “DeviceImageWorkers” class. These threads pull awaiting devices out of the “DeviceImageQueue” in FIFO order and attempt to create images of their contents. For each device pulled out of the queue, the image worker thread attempts to start a new instance of `Aimage`. The image worker thread monitors the `Aimage` process for progress information (percent complete, errors) and then updates the GUI progress displays accordingly (see Section 5.4.2).

When `Aimage` finishes, the device’s imaging status is marked with an error or as complete and safe for removal (see Section 5.4.5).

5.4.2 AcqMan’s GUI Design

AcqMan uses the Swing toolkit to render all the GUI components. The GUI has a few main elements including:

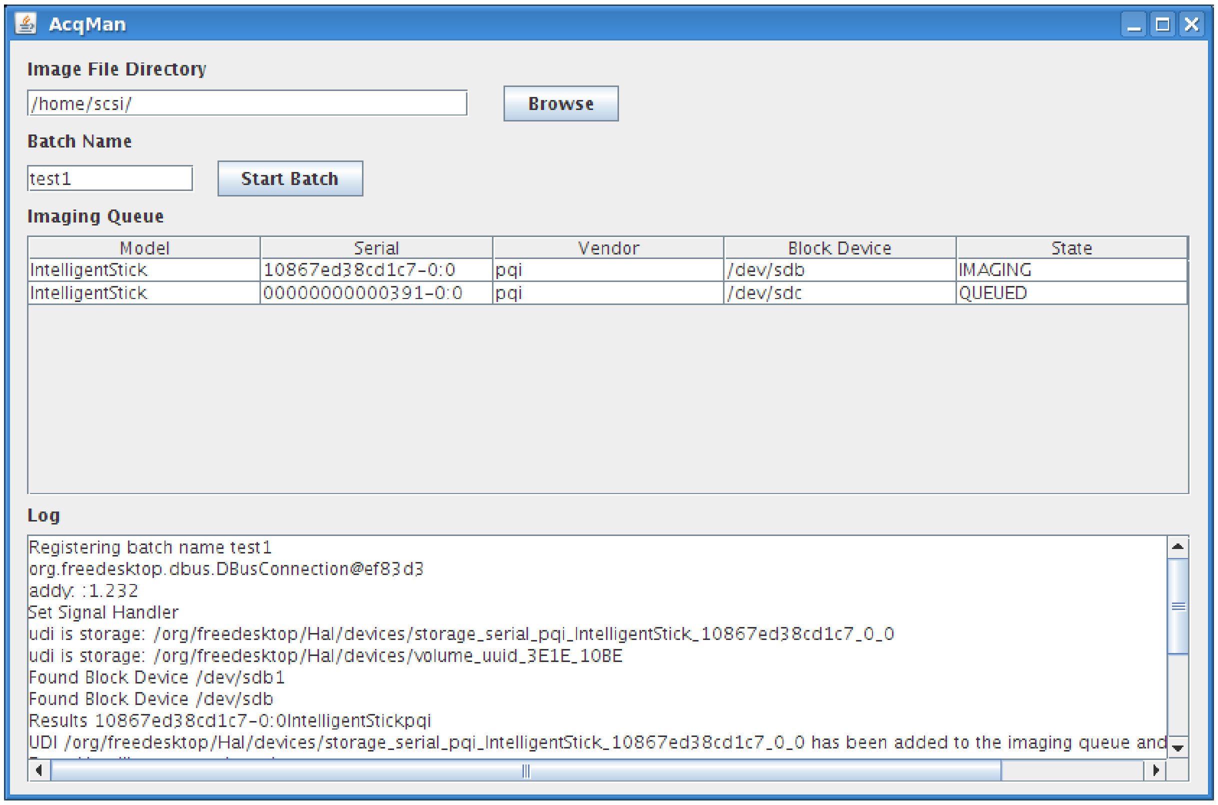

Image File Directory

/home/scsi/

Batch Name

test1

Imaging Queue

Model	Serial	Vendor	Block Device	State
IntelligentStick	10867ed38cd1c7-0:0	pqi	/dev/sdb	IMAGING
IntelligentStick	00000000000391-0:0	pqi	/dev/sdc	QUEUED

Log

```
Registering batch name test1
org.freedesktop.dbus.DBusConnection@ef83d3
addy: :1.232
Set Signal Handler
udi is storage: /org/freedesktop/Hal/devices/storage_serial_pqi_IntelligentStick_10867ed38cd1c7_0_0
udi is storage: /org/freedesktop/Hal/devices/volume_uuid_3E1E_10BE
Found Block Device /dev/sdb1
Found Block Device /dev/sdb
Results 10867ed38cd1c7-0:0IntelligentStickpqi
UDI /org/freedesktop/Hal/devices/storage_serial_pqi_IntelligentStick_10867ed38cd1c7_0_0 has been added to the imaging queue and
```

Figure 5.3: A screen shot of AcqMan in action with one device imaging and another queued for imaging.

1. An area to choose the destination directory of the image files.
2. A batch name field which allows the user to choose the first part of an image file name.
3. A status pane where the model, serial number, manufacturer, block device, and state (imaging, finished, error, etc) is displayed
4. A log pane where the frontend can detail errors and status updates.

5.4.3 AcqMan's Interactions with HAL

Device Insertion Detection as an Impetus for the Frontend

The first step to getting HAL to work in AcqMan was to attach to the system wide D-Bus. At the time of this writing, there is only a single implementation of a D-Bus framework in Java.

In AcqMan the "halHandler" class is responsible for connecting to the system D-Bus and registering a signal handler for device insertions. The connection is made by simply specifying which type of bus is required (system or per-login). After the connection is made, AcqMan resisters a signal handler by specifying the full name of the D-Bus message (Hal.Manager.DeviceAdded) and the method which will be spawned in a thread to handle the incoming message (halHandler.DevAddedHandler.class.handle). These messages serve as a starting point for device imaging in the AcqMan interface.

There were several challenges that we had to overcome in integrating dbus-java with AcqMan. First, the documentation and examples were centered around the idea of using dbus-java to expose API services for an application. There was very little guidance on how to use the libraries to get at other APIs that were exposed on the system bus. It took a great deal of effort to get our frontend to the point where it was able to connect to the system wide D-Bus and access the API functions in HAL.

Limitations in HAL and the operating system kernel prevent us from automatically detecting all device insertions. We found this to be problematic during our tests with the Sony MRW62E, Sandisk SDDR-89, and Lexar RW108B2 USB media card readers. We plugged in each of the readers and then tested if HAL would generate a device insertion event as an SD card was plugged in. We also tested if the card reader would generate an insertion event as media cards were plugged into the remaining slots.

The Sony MRW62E card reader is a multi-form reader with slots for SD, CF, Memory Stick, and Smart Media flash memory systems. We found through testing that this reader would report SD card insertions under FreeBSD but would fail to do so under Linux. Under FreeBSD this reader would also report multiple device insertions; under Linux it would only report insertions and allow one to read the media if no other slots were in use.

The Sandisk card reader reported device insertions under both FreeBSD and Linux. The reader was also able to handle the detection of several media cards at once under FreeBSD. However, under Linux the Sandisk device would refuse to recognize and report a second media card insertion. The Lexar device behaved in the same way as the Sandisk device.

Upon further investigation, we found that HAL has a “media check” feature associated with removable media devices. This media check feature tells the HAL daemon to poll the removable media device for inserted media. Under Linux, all of the card reading devices were seen as a single block device by the kernel. This is the reason why the card readers could not handle detection of multiple pieces of media under Linux: the single block device was already polled

Card Reader	Sony MRW62E	Sandisk SDDR-89	Lexar RW108B2
Insertion Event Generation	FreeBSD	Both	Both
Insertion Event for Second Card	FreeBSD	FreeBSD	FreeBSD
Read from Multiple Cards	FreeBSD	FreeBSD	FreeBSD

Table 5.1: USB media card reader support for device insertion under FreeBSD and Linux

and enumerated by HAL leaving no other block devices for the other media to be read from.

All three of these card readers were able to detect insertion and read cards from multiple slots at once in Windows and Mac OS X. These observations were made on machines which D-Bus and HAL were not running. FreeBSD matches the capabilities of Windows and OS X as far as device insertion and reading multiple card slots is concerned. Linux was not able to detect and read multiple cards in multiple slots.

Currently, AcqMan does not have a mechanism to allow imaging of devices which have not been detected through a “DeviceAdded” message from HAL. Support could be added to force HAL to rescan the system for all block devices. The rescan could be manually invoked or on a timer. If block devices are found, AcqMan could give the user the option of imaging the device. However, this does not work towards the goal of providing a hands off system to the unskilled user. Furthermore, this solution will not do anything to provide support to devices that the OS kernel does not support itself, it just enables AcqMan to detect non-hotpluggable devices. A better solution would be to catalog and report these errors to the HAL developers so that they may be fixed in future versions.

Device Metadata Collection

Device metadata is central to enabling AcqMan to achieve intelligent image upload, duplicate image detection, and accessibility to the untrained user. In collecting this metadata we are able to free the user from having to obtain and enter data such as device model, serial number, and manufacturer. Also, automated metadata collection enables less error-prone operation in that we are not relying on a human to transcribe device identifiers.

AcqMan collects device metadata after a device insertion message is received from HAL. The following is a list of metadata collected from the device and its usage in AcqMan:

- **Model, Manufacturer, and Serial Number**
  These three pieces of data are used in querying the ADOMEX repository for similar

devices that may have already been imaged. They are also passed to `Aimage` so that they may be stored with the image of the device. Lastly, they are displayed to the user by the frontend as a mechanism to help with correlating the drive's status with the physical device as it stands plugged into the computer.

- **Device Type (i.e. removable SD card, fixed hard disk)** This piece of information is not currently used by AcqMan but it will be used later to tell the user to remove that class of device.
- **Size (in bytes)**
  This information is being used inside of AcqMan to report the size of the device to the user; Aimage separately determines this information as well.
- **Block Device Path**
  This path is passed to `Aimage` to begin the device imaging process.

`Aimage` is also able to collect some device metadata on a FreeBSD system. It does this by querying the system's MIB thourgh the sysctl interface. Aimage has os-specific code for FreeBSD and several versions of Linux. AcqMan's metadata collection is slightly different for a few reasons. First, it uses HAL which is provides a standard API for collecting metadata across Unix like operating systems. Next, AcqMan uses the metadata to track device images in the ADOMEX repository, `Aimage` does not support checking against a repository for like images. Last, `Aimage` does not support device insertion detection. In short, AcqMan gathers an inserted device's metadata so that it can decide if a device needs to be imaged and if so direct Aimage to the proper device.

In the course of our development, we encountered several problems with obtaining and using the metadata from HAL:

- **Unavailable Device Metadata**
  In our attempts to collect metadata from USB flash drives, various memory card formats, and hard disks we discovered that device metadata is often unavailable to HAL. This is mainly due to lack of support in either the adapter which connects the device or in the driver for the device itself.

  In one case we encountered a situation where we were unable to read the device metadata off of a portable Seagate hard drive when it was plugged into a USB to ATA adapter from

Vantec (NexStar I). When we removed the device from the adapter and plugged it directly into a ATA port, we were able to collect the metadata. In this case, we believe the Vantec adapter was manufacturered without support for reading device metadata off of connected hard disks.

In another case, we were unable to read the serial number, make, and model of several SD card which were plugged into a variety of USB media card readers from Sandisk, Lexar, and Sony. We discovered two sources of the problem. First, the SD specification was not fully implemented under the Linux kernel we were using and therefore the device metadata would not be propgated to HAL. Additionally, when we plugged the adapters into a Windows machine, we were still not able to read the serial numbers on the SD cards. This was a case where both the USB media card readers and the Linux drivers did not support reading device metadata. From this, we determined that the serial was not actually being passed through the chipset responsible for bridging the SD card onto a generic USB mass storage device.

- **Poor Device Manufacturing Practices**
  Many of our test devices included inexpensive and second-hand USB flash drives which we found to have poor or non-existent metadata on the device's manufacturer, serial number, and make. For example, we had several unbranded USB flash drives of Chinese origin that had serial numbers of "`^_`" or "`0`". Not only was this information unhelpful in identifying a single device, but it was also present in several devices of the same physical appearance making identification of an imaged device difficult.

- **Non-field Specific Data**

  Device quality and driver support aside, HAL reports device metadata poorly, with a single device identifier that it places in the model, manufacturer, and serial number fields of a device's information. This device identifier includes all four pieces of information concatenated together to form: "`DEVICETYPE_MANUFACTURER_MODEL_SERIAL`". We were not able to find any information on why the HAL developers chose do this.

AcqMan uses this metadata in several critical steps such as duplicate image detection and intelligent image uploading. We have developed strategies for handling metadata inconsistencies and we will discuss them in the following sections.

Targeted Device Diagnostics and Removal

HAL maintains a tree of all the system devices and how they are connected to each other. In order to provide more accessibility to the untrained user, we want to use HAL to direct the user to devices that need physical interaction. This direction would be done by showing them a photo of the physical port on the actual system which they are supposed to remove the device from. We did not have the time implement this functionality in AcqMan but we have included a discussion on how such a mechanism would be designed.

In normal operation, AcqMan should detect and image any inserted device and then prompt the user for removal. We want a user to be able to recognize and remove a device which has finished imaging without disturbing devices which may be in the middle of imaging. Functionality that informs the user when the device is done and gives specific information on where the device is connected and how to remove it is both desirable from a usability standpoint and necessary for error-free operation.

To fulfill these needs, we would like to use HAL to determine where a specific device is connected and then have AcqMan present the user with photographic instructions detailing where the device is plugged in and how it should be removed. This involves adding a whole subsystem to AcqMan. This system would involve two distinct pieces: system bus mapping with photo entry and physical interaction notices with photo display.

System bus mapping would involve a skilled user who would create a profile for the system on which our AcqMan interface was to be deployed. In this mode, AcqMan would walk the user through all of the buses on the system by having the user plug a test device into each available port. In this process the user would be asked to take a digital photo of each physical port and its location on the system as devices are plugged in and detected. This mapping of ports to photos of their physical locations would be saved for later display to a user when removal of a device is required.

The physical interaction notices would include an action required (e.g. remove device, check operation), a description on the device (e.g. Seagate Harddrive ST3870002), and a photo of the port where the device is connected. We believe that pointing this information out in an explicit and graphical fashion will be an improvement in the usability and reliability of AcqMan.

5.4.4 AcqMan’s Interactions with the ADOMEX Repository

Duplicate Image Detection

Using XML-RPC, AcqMan can query the ADOMEX repository and determine if a device has already been imaged. If the device has already been imaged, AcqMan must decide if the data has changed and the device has to be re-imaged.

To begin the process, AcqMan will take metadata from the connected device before it is imaged. As we discussed earlier, this metadata is not always available or accurate. We do not plan to remedy this within AcqMan, instead we send whatever metadata AcqMan is able to gather on the device to the ADOMEX repository for an imaging determination. It is not optimal to first image the device and they query the repository with the hash of the image. Using device metadata allows us to determine if a device is in the repository without imaging it first. This saves both time and space on the acquisition station in the majority of the cases.

Upon insertion, the device’s model, manufacturer, and serial number are sent to the ADOMEX repository via XML-RPC for a determination on if the device should be imaged. The ADOMEX repository looks at the received metadata and tries to determine if the values are both sane and meaningful. By sane we mean that things such as the serial number are actually a number of significant length. Meaningful in this case means that the metadata uniquely identifies the device amongst other devices in the same category.

We believe that doing these checks on the ADOMEX repository is optimal because the repository inherently has access to more images than any single acquisition station. Furthermore, the repository gives us a central place to make updates to the imaging decision algorithm. This allows an organization to shape their device imaging policy in a central location and easily make all their acquisition stations follow suit.

Given a sane and meaningful serial number, model, and manufacturer of a device it is possible to determine if a device is present in the ADOMEX repository. But if a device is present in the repository it is not possible to decide (with the aforementioned information) if data on the device has changed and if that change necessitates the creation of another device image. Also, it is not possible to decide if a device is in the repository with incorrect or incomplete device identification metadata.

More information is needed for these types of decisions. In these cases, we use `Aimage` to image the device and then compare its cryptographic hash against the images in the repository. If the local image is found to have significant differences from the ones in the repository it is kept and marked for upload. Otherwise, the image is deleted and the operator is notified of an attempt to duplicate an image.

Intelligent Image Upload

There are two types of images that need to be uploaded to an ADOMEX repository: images of devices not already in the repository and images of devices that are in the repository but have changed data. These uploads must often be performed in an environment with intermittent and bandwidth limited network connectivity. We were not able to implement any upload functionality for AcqMan but we do have a design for handling uploads of the two different types of images.

AFF files are split into segments each of which has a cryptographic hash associated with it. Once network connectivity is available, AcqMan should take stock of its local images and prepare to upload their contents segment by segment, verifying each segment. To achieve this AcqMan would take the following steps once network connectivity was established:

1. **Inventory Local Images for Upload**
   For each image on the acquisition station, AcqMan will pull out the SHA256 image hash, and the device's model, serial number, and manufacturer.

2. **Query the ADOMEX Repository**
   Using the data collected above, AcqMan will query the ADOMEX repository. The ADOMEX repository will respond to the query with the SHA256's of the images in three separate categories. The first category will represent the images that the repository needs in their entirety. The second category is a list of images that the ADOMEX repository needs a partial copy of. This list will be based on devices that are present in the repository but whose SHA256s do not match. The last category will be a list of images that the ADOMEX repository already has and that the acquisition station can delete.

3. **Upload the Partial Images**
   AcqMan would first attempt to upload the images that the repository needs a partial copy of. To accomplish this, AcqMan would open the images that the ADOMEX repository requested a partial copy of and upload a list of every segments SHA256. The ADOMEX

repository will then request all of the segments that do not match the image in the repository which AcqMan will upload. Upon successful upload of every segment, the repository returns an acknowledgment. Upon completion of all segments of an image, AcqMan would move on to the next image.

4. **Upload Complete Images**
   Next, AcqMan would start uploading images that are not in the repository. This would be done, segment by segment in the same manner as the partial image upload.

5. **Verify All Images are Uploaded** After completing all of its upload tasks, AcqMan would query the ADOMEX repository one more time with the SHA256 of all the images it believes were uploaded. The ADOMEX repository checks for these images and responds accordingly. If all images are in the repository, AcqMan goes on to the next step. If the repository is unable to locate one or more of the images, the process is repeated starting with step 3.

6. **Remove Local Copies of All Uploaded Images** Once the uploads are verified as having been safely stored, and depending on organizational policy, AcqMan will remove all of the device images on the local machine.

These steps achieve our requirement for intelligent image upload that we set forth in Chapter 3. The uploading of partial images phase ensures that we retransmit parts of images in the face of network connection issues. Also, partial image uploading gives us the ability to only upload data that has changed in an image which is already in the ADOMEX repository allowing savings in both bandwidth and disk space.

Device Wiping

Devices collected in a DOMEX environment often contain sensitive information. AcqMan should have the ability to wipe the contents of these drives once they are imaged to the local acquisition station. While this facility may not be useful or desired for every organization, it is useful to make it avalible for the organizations that need it. Again, we have not implemented this functionality but we do have a workable design.

For a device to be wiped, AcqMan should verify that all of the following conditions are met:

1. An image of the device has been made.

2. `Aimage` did not return any major fault codes as the image was made.

3. The device does not need to be returned to operation.

The first two conditions do not need user interaction to evaluate, however, the last condition does. In terms of AcqMan, the best time to wipe a device is right before the user is signaled to remove it. As AcqMan is notifying the user that a device is done imaging and ready to be removed it could also ask if the user would like to wipe the device. If the user agrees, the device should be wiped pursuant to the guidelines in the NIST special publication 800-88[18].

However, one of our goals for this thesis is ease of use by an untrained user. While we are not asking the user to actually perform the device wiping, we are leaving them with a decision that they might not feel comfortable making without training. As an alternative to putting the decision in the user's hands, AcqMan could be configured with a per-organization wiping policy. This would take the decision out of the user's hands and leave it up to the organization.

Once it is decided that the device should be wiped, any utility that runs on the command line of a Unix like system and conforms to NIST special publication 800-88 could be used. The command line complexities of this utility would be hidden by the AcqMan frontend and the wiping would be done automatically. While a full survey of command line wiping utilities has not been done for this thesis, the BC Wipe, Killdisk, and Wipe applications could be well suited for integration into AcqMan.

5.4.5 AcqMan's Use of Aimage

Creating and Storing the Device Image

As previously explained, `Aimage` plays a large role in AcqMan's overall operation as it is responsible for creating and storing device images along with all related metadata. AcqMan interacts with `Aimage` solely through the use of Java's built-in Process class. This class allows us to spawn a process and map its standard in, out, and error back into our program.

Earlier we discussed AcqMan's different threads and structures. In AcqMan the image worker threads are responsible for all the interactions with `Aimage`. These threads take devices that are waiting to be imaged from a queue. Once the device is taken off of the queue, AcqMan begins to build the command line that it will use to spawn an instance of `Aimage`. This command line needs the `Aimage` command, the block file of the device, and a to write the image.

After this command line is built, AcqMan attempts to open an `Aimage` process in a new terminal window so that the user can watch the imaging process progress. The imaging thread blocks while `Aimage` is working. In this version of AcqMan we only read the exit code from `Aimage` to determine if the imaging failed or succeeded. This is not the best way to tie `Aimage` into our frontend as we are not able to fully insulate `Aimage`'s operations from the user and make the frontend look like its own simple entity.

Given more time, we would like to move `Aimage` behind the scenes and only present a few details of its operation to our users. We would do this using Aimage's "batch mode" which provides a text parsing friendly way to update AcqMan's imaging progress monitors. Also, we would explore all of `Aimage`'s return codes and give more accurate diagnostic information to the user within the AcqMan interface.

Storing Metadata Collected in the Image

Another reason that we use the AFF file format is its ability to store arbitrary name value pairs with the image of the device. This capability allows us to both store and transmit metadata with the image. It frees us from having to write functionality within AcqMan to make sure that the device metadata stays bound to the device image at all times.

AcqMan contains rudimentary support for storing device metadata in the AFF image. To do this, it uses a program called `afsegment`. `Afsegment` has the ability to add, edit, and delete arbitrary name value pairs within an AFF file. AcqMan writes device metadata to the image after Aimage has completed imaging the device. AcqMan does this in the same way it invokes Aimage, by building a command line with the `afsegment` command, the name of the image file, and the name value pairs of the device metadata.

While AcqMan handles retrieving device metadata through HAL, it does not yet gather situational device metadata from the user. Collecting this metadata presents two problems: how to collect it and when to collect it. User generated metadata can include such things as the location the device was captured from or the person found in possession of the device. In fact, the specific metadata to collect can depend upon how the device was obtained. For example, in some cases the device could be collected off of a person with no identification. This varying availability of metadata does not lend itself to a traditional structured input request where the user is asked to fill in well-defined fields such as name, case number, and location.

To enable user generated metadata collection in AcqMan, we plan to give the user the ability

Feature	Designed	Implemented	Tested
Frontend Design	Yes	Yes	Partially
Internal Frontend Design	Yes	Yes	Partially
Device Insertion	Yes	Yes	Partially
Device Metadata Collection	Yes	Yes	Not heavily
Targeted Device Removal	No	No	No
Duplicate Image Detection	Yes	Partially	No
Intelligent Image Upload	Partially	No	No
Device Wiping	Yes	No	No
Creating and Storing Images	Yes	Yes[a]	Partially
Storing Metadata in Image	Yes	No	No

[a]AcqMan does not yet implement Aimage's image encryption features

Table 5.2: Status of AcqMan's features as discussed in this thesis.

to input metadata as the device is imaging or just before it is to be removed. The user would be able to input metadata as the device is imaging by clicking on an "Add Metadata" button. This would spawn another window which would have a large text input area in addition to an informational pane. The informational pane would consist of a list of "tags" that the user could prefix their information with. These tags would help the ADOMEX repository delineate between different pieces of metadata. For example, a device captured from Alice might result in having the user to enter "owner:Alice" into the device metadata window.

Upon completion of the image, AcqMan would prompt the user to remove the device. In this prompt AcqMan would give the user a last chance to input or update metadata for a device. If metadata is entered, `afsegment` would be used to write the data into the device's corresponding AFF file. If no metadata is entered, AcqMan could refuse to let the device be removed if the organization desired metadata entry for every device. However, this refusal is not a full proof mechanism for forcing metadata entry as the user could remove the device anyway. The organization may need to develop some strict policy to combat this.

5.4.6 Status of AcqMan's Features

In the previous sections, we discussed designs and implementations for different pieces of AcqMan. We did not have time to provide complete designs, implementations, and tests for each of the features. Table 5.2 shows the progress to date of the various necessary parts.

CHAPTER 6: Conclusion and Future Work

6.1 Findings

6.1.1 ADOMEX Acquisition Tool

This thesis sought to determine a workable design for an ADOMEX acquisition tool. We reviewed existing tools that could serve as an ADOMEX acquisition tools and determined that none met all of our requirements. However, the `Aimage` acquisition tool met our requirements for image creation and metadata storage. Using `Aimage` and other technologies such as HAL, Java, XML-RPC, and D-Bus we created a program named AcqMan to handle our ADOMEX acquisition requirements. In developing AcqMan we discovered and documented many limitations in HAL and its ability to support device insertion for differt types of media.

We determined that with more development, AcqMan could be used as a workable design for an ADOMEX acquisition tool. Also, we laid out plans for this development by designing the intelligent upload, user metadata collection, duplicated image detection, and device removal features for AcqMan. We discussed the technical limitations that came about in desiging these features and tailored the designs to avoid the limitations where possible.

6.1.2 Acquisition Tool Automation and Usability

This thesis also sought to create an acquisition utility that was as automated and accessible to the untrained user as possible. We determined that complex user interactions with the acquisition system would be the biggest deteriment to both automation and untrained users. As a result, we designed AcqMan to be as automated as possible without the need for user interaction. This resulted in the AcqMan implementation requiring zero user interaction to detect and image an inserted device.

For the pieces of AcqMan we did not implement, we designed low interaction processes for duplicate image detection, image upload, and metadata storage. We documented several issues in devloping these low interaction processes. These issues were centered around the inability to retreive accurate and useful metadata from certain devices. We found the causes of these issues to range from poor device manufacturing to lack of device metadata support amongst different operating systems. We engineered AcqMan to remain as automated as possible automated even in the face of these failures and metadata anomalies.

6.2 Future Work

This thesis established the framework and baseline for a workable ADOMEX acquisition system, AcqMan. In establishing this baseline, we were not able to implement all of the functions that we had planned for AcqMan. Additionally, we were not able to perform a complete set of tests on AcqMan to work out all of the anomalies that may be seen on different operating systems and hardware configurations. We would like to see work continue on AcqMan, primarily in adding and testing features.

Also, we did not have a full implementation of a ADOMEX repository to test AcqMan against. We would like to see the ADOMEX repository built out to a point where we are able to test all of AcqMan's capabilities against it. We detail the directions we would like this work to take in the following sections.

6.2.1 AcqMan Feature Completion

We would like to see future work implementing the following features as designed in Chapter 5:

- Targeted Device Diagnostics and Removal
- Intelligent Image Upload
- Device Wiping
- Collecting and storing user generated metadata

The implementation of these features would yield a fully functional version of AcqMan that would meet all of our requirements for an ADOMEX acquisition system. Furthermore, these features would leave the completed version of AcqMan requiring very little user interaction and thus it would be suitable for untrained users.

6.2.2 AcqMan Test and Evaluation

This thesis did not place much emphasis on testing AcqMan in an ADOMEX environment. Because of AcqMan's incomplete nature, extensive testing was not possible and therefore future work testing AcqMan is needed. We have listed several general areas where AcqMan and its features should be tested for speed, correctness, and ease of use.

- **Feature Correctness**
  To make sure that AcqMan will operate correctly in a deployed environment its functionality should be thoroughly tested under normal conditions. These tests should include many different types of devices as well as many different types of network environments to simulate a real DOMEX environment.
- **Fault Tolerance**
  Device conditions, metadata, and types are not always optimal for imaging and exploitation in an ADOMEX system. AcqMan should be tested against many diverse devices in differing conditions to determine how well it handles sub-par devices.
- **System Performance**
  ADOMEX environments can often be very high volume in terms of the media to be collected and analyzed. AcqMan should be tested to see how well it performs with regards to usability, image creation, and image upload.

These tests should be done with an aim towards using the results to improve AcqMan.

6.2.3 Acquisition Appliance Development

This thesis did not deal with building a system to support field device acquisition. AcqMan was designed to run as the dominant application on an acquisition station handling all the nessacary DOMEX operations. We would like to see an acquisition appliance that is built to run AcqMan in a manner which maintains the requirements laid forth in the previous chapters. Additionally, we would like to see this acquisition appliance built to the needs of a few different organizations so that AcqMan may be further refined.

6.2.4 ADOMEX Repository Design and Implementation

Much of AcqMan's unimplemented features rely on interaction with the ADOMEX repository. A prototype of the ADOMEX repository exists but does not fully implement all of the features nessacary for AcqMan to interact with it. Specifically, the ADOMEX repository does not have the following facilities:

- **Image Upload via RPC**
  We described an image upload design in Chapter 5.
- **Duplicate Image Detection**
  The ADOMEX repository can be queried with device metadata to find out if an image exists in the repository. However, the repository does not handle the anomalies in device metadata that we discussed in Chapter 5. Additionally, the repository is not able to make an duplicate image decision based off of the heuristics we described earlier.

The aforementioned facilities would need to be implemented and tested before AcqMan could interact properly with the ADOMEX repository.

Bibliography

[1] Xml-rpc specification, 1999. URL http://www.xmlrpc.com/spec. [Online; accessed 5-June-2008].

[2] Air - automated image and restore, 2005. URL http://air-imager.sourceforge.net/. [Online; accessed 11-June-2008].

[3] dcfldd - latest version 1.3.4-1, December 19, 2006. URL http://dcfldd.sourceforge.net/. [Online; accessed 6-June-2008].

[4] Aff: The advanced forensics format, 2008. URL http://www.afflib.org. [Online; accessed 6-June-2008].

[5] D-bus documentation, 2008. URL http://dbus.freedesktop.org/doc/dbus/api/html/index.html. [Online; accessed 5-June-2008] API documentation only.

[6] Forensic toolkit 2.0, 2008. URL http://www.accessdata.com/Products/ftk2test.aspx. [Online; accessed 11-June-2008].

[7] Guymager, 2008. URL http://sourceforge.net/projects/guymager/. [Online; accessed 11-June-2008].

[8] Hal - hardware abstraction layer, 2008. URL http://freedesktop.org/wiki/Software/hal. [Online; accessed 5-June-2008].

[9] libewf: Project info, May 2008. URL http://www.uitwisselplatform.nl/projects/libewf/. [Online; accessed 6-June-2008].

[10] The open computer forensics architecture (ocfa), 2008. URL http://ocfa.sourceforge.net/. [Online; accessed 11-June-2008].

[11] What is d-bus?, 2008. URL http://freedesktop.org/wiki/Software/dbus. [Online; accessed 5-June-2008].

[12] windbus, 2008. URL http://sourceforge.net/projects/windbus. [Online; accessed 8-June-2008].

[13] Xml-rpc home page, 2008. URL `http://www.xmlrpc.com`. [Online; accessed 5-June-2008].

[14] US Intelligence Community Directive 302. Document and media exploitation, July 2007.

[15] Simson L. Garfinkel. Document & media exploitation. *Queue*, 5(7):22–30, 2007. ISSN 1542-7730.

[16] Simson L. Garfinkel, David Malan, Karl-Alexander Dubec, Christopher Stevens, and Cecile Pham. Disk imaging with the advanced forensic format, library and tools. In *Research Advances in Digital Forensics (Second Annual IFIP WG 11.9 International Conference on Digital Forensics)*. Springer, January 2006.

[17] Matthew Johnson. Java d-bus implementation documentation, 2008. URL `http://dbus.freedesktop.org/doc/dbus-java/`. [Online; accessed 5-June-2008].

[18] Richard Kissel, Matthew Scholl, Steven Skolochenko, and Xing Li. Guidelines for media sanitization. Technical Report 800-88, NIST, September 2006. URL `http://csrc.nist.gov/publications/nistpubs/800-88/NISTSP800-88_rev1.pdf`.

[19] Havoc Pennington. D-bus specification, 2008. URL `http://dbus.freedesktop.org/doc/dbus-specification.html`. [Online; accessed 5-June-2008].

[20] Chris Prosise. *Incident Response & Computer Forensics*. McGraw-Hill/Osborne, New York, 2003. ISBN 007222696x. 26 pp.

[21] Paul Rubin, David MacKenzie, and Stuart Kemp. *dd Manual Page*. Free Software Foundation, March 2007.

[22] Virginia Contracting Activity MDA908-97-R-0040. *Document Exploitation (DOCEX) Transportable Support System (DTSS)*, July 1997.

Referenced Authors

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudly Knox Library
   Naval Postgraduate School
   Monterey, California