Final report: Automatic Detection of steganographic content

STTR Phase 1 Project FA9550-04-C-0110
CLIN: 0001AC
June 30, 2005

Summary

In this document, we will describe our Anti-Steganography project for Phase 1. First, we will present the Software Architecture in Section 1. In Section 2, we will describe in details the Application Programming Interfaces (APIs) and the Software Implementation. In Section 3, we will show the organization of the source code. Section 4 discusses how to maintain and extend the software. Section 5 discusses the automatic engine update service. A description of the Phase 1 demo is given in Section 6. Finally, we describe the steganography research activities and results in Section 7.

1 Architecture

Figure 1 shows the highest level of our anti-steganography software architecture. It consists of an Anti-steganographic Engine and a User Interface that, as the name implies, provides the user interface to the Engine.

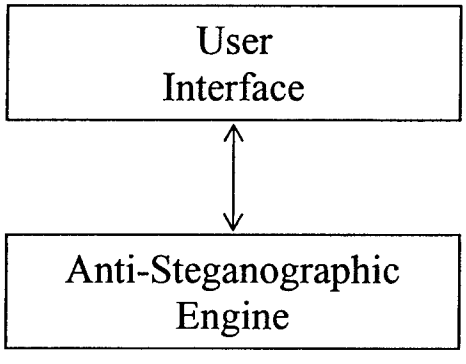

Figure 1. Overview of Anti-Steganographic Architecture

1.1 Anti-Steganographic Engine

The main purpose of the engine is to take a file as an input and perform one of the following three operations:

1. To detect whether or not the file contains a steganography

2. To remove the steganography from the file

3. To extract the steganography from the file

The input file can be passed to the engine as a filename or a data structure that represents the content of the file. The engine can process only one file at a time. Depending on the type of the operation, the output can be a confidence level (probability) that the input file contains a steganography, a file whose embedded steganography being destroyed by the engine, or a file that contains the extracted steganography. The output file can be either a filename or a data structure that represents the content of the file.

1.2 User Interface

The main purpose of the User Interface is to provide an intuitive interface to the engine for the user. With the interface, the user can do many things such as scanning the hard drives for files that contain the steganography, removing the steganography from them, etc... The user can also scan a file downloaded into the browser for steganographic content and alert the user when such content is discovered. The applications are endless as long as they require only the three operations offered by the anti-steganographic engine.

The implementation of the User Interface is flexible. The User Interface can be a stand-alone application such as a Windows application. It can be a browser plugin where it gets invoked automatically by the browser whenever a file is downloaded. Or it can also be a DLL or a component of another application. For the Phase 1 development, we provided a console application that takes the input file as an argument and writes the results to the standard output.

1.3 Data Flow

The data flow between the User Interface and the Anti-steganographic Engine runs in both directions. In general, the User Interface sends a request along with the input file to the Anti-steganographic Engine. The Engine responses with either a confidence level or an output file. The User Interface processes the response then optionally repeats the cycle by sending the Engine another request.

1.3.1 User Interface

At the start, the User Interface presents the user with some kind of dialog box that asks the user to choose the files to run the engine on. The files can be some specific folders, certain email attachments, or the downloaded files from the internet, etc... The dialog box also lets the user to define the scope of the checking such as limiting the file types to be checked and certain classes of algorithms to be used. It also allows the user to do other kinds of customization such as to schedule the time to run the engine, and to automatically examine all the downloaded files after they are downloaded, etc...

Next, the User Interface converts the keyboard and mouse input from the dialog box or dialog boxes into a set of files. Then, it runs the files through the Anti-Steganographic Engine. Depending on the responses from the Engine, the User Interface might present more dialog boxes not only to show the results, but also to allow the user to take actions such as quarantining the files with embedded steganography.

We propose to implement the User Interface in the Phase 2. For the Phase 1 development, we wrote a console application with a very simple UI for the purpose of testing our Anti-Steganographic Engine. The application is called TestASEngine. It must be run from a Windows command prompt or Linux command line as follows:

TestASEngine command filename

where command is one of the followings:

d : detect steganography

e : extract steganography

r : remove steganography

1.3.2 Anti-Steganographic Engine

When the User Interface passes a file to the Anti-Steganographic Engine, it also has to pass other information such as the class of the algorithms to be used, and the type of operation to be run (detection, removal, or extraction). The Engine then invokes a bunch of modules to process the file and returns with either a confidence level or an output file. The complete data flow inside the Anti-steganographic Engine is shown in Figure 2.2. Before we go into the details of the data flow, let's get familiar with all the components of the Engine first.

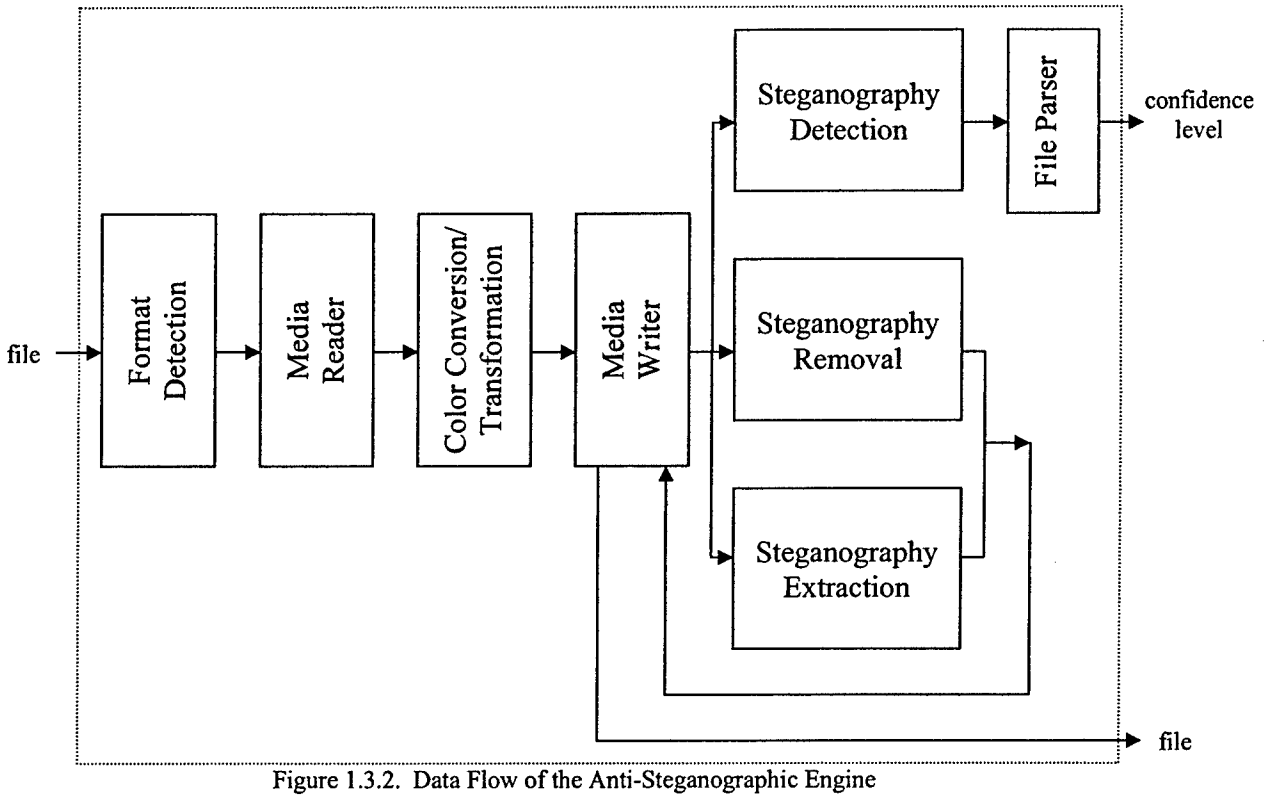

Figure 1.3.2. Data Flow of the Anti-Steganographic Engine

1.3.2.1 Engine Components

The Anti-steganographic Engine consists of 8 modules: Format Detection, Media Reader, Color Conversion/Transformation, Media Writer, Steganography Detection, Steganography Removal, Steganography Extraction, and File Parser. The modules are described below.

Format Detection

This module is intended to speed up the engine. In real life, the user would scan thousands of files for steganographic content. In this case, the engine has to run all the steganography detection algorithms on them. That would take a lot of processing power. Fortunately, each steganography detection/removal/extraction algorithm works only on certain types of files. If the Format Detection module can determine the file type of an input file, the engine can skip all the algorithms that do not apply to that file type. Furthermore, for most file formats, especially image file formats, the file type can be determined very quickly by looking for certain signatures at certain positions within the file. Therefore, having a format detection that can quickly determine the file types of the input files can have great impact on the performance of the engine.

For the file formats where it would take too much time to determine the file type, this module would skip the format detection process. In this case, all the steganography detection algorithms would be called. It is then up to the steganography detection algorithms to eliminate the file if the file type is not acceptable by the algorithms.

Media Reader

Theoretically, the steganography can be embedded into any kinds of files. Practically, it is mostly embedded into the media files, especially the image files. Consequently, a lot of the anti-steganography algorithms work with raw images, regardless of how they are stored as the files. The Media Reader module simply reads and parses the file into a data structure respresenting the media so that the steganography detection/removal/extraction algorithm can work on it. By making the anti-steganography modules independent of the file formats, as new file formats are

invented, only the Media Reader should be extended to support them. That would be a minimum of work compared to modifying all the anti-steganography implementations.

For the anti-steganography algorithms that work with file formats that are not supported by Media Reader, the Media Reader module simply passes the data without any file parsing.

Color Conversion/Transformation

Some steganography softwares embed the information into certain image components such as Red, Green, and Blue channels. Others work on different components such as Hue, Saturation, Luminance. Yet, there are also others that work on palettized images. The anti-steganography softwares must detect, extract, and remove the steganography from the same channel that the steganography softwares have inserted into. This module simply converts between different color channels.

Media Writer

One of the major design requirements of this anti-steganographic engine is the extensibility of the engine. Though it is desirable to implement the anti-steganography algorithms as direct components of this whole architecture, we still want to take advantage of stand-alone anti-steganography applications that are available out there. Because they are stand-alone, they usually take a file in certain file format as an input. To run these applications, the Media Writer will write files in the format acceptable by them.

Steganography Detection

This module performs the steganography detection on the input file. As it will be explained later, this is a collection of many implementations of different steganography detection algorithms. As new algorithms are invented, they can be implemented and added to this module.

Steganography Removal

This module is similar to the Steganography Detection module, except that instead of detecting the existence of the steganography, it modifies the input file so that the embedded steganography is destroyed. For many steganographies, it is possible to detect them, but not to remove them; therefore, the User Interface should let the user to isolate the cover files in this case. Like the Steganography Detection module, as new steganography removal algorithms are invented, they can also be implemented and added to this module.

Steganography Extraction

This module extracts the steganography from the cover files. In most cases, the extraction cannot be done even though the stenographies are detected or even removed; therefore, the User Interface should inform the user when that happens. As new steganography extraction algorithms are invented, they can be implemented and added to this module.

File Parser

The output of the Steganography Detection module is the confidence level that the input file contains a steganography. To take advantage of stand-alone anti-steganography detection applications that usually write the result to a file, the File Parser module will read the result file and return the confidence level as required by the engine API.

1.3.2.2 Data Flow within the Anti-Steganographic Engine

First, the User Interface module sends the input file to the Anti-Steganographic Engine. The file can be a filename, or a data structure representing the content of the file such as a bit stream of the file. For the Phase 1 development, we only accepted the filename, and left the bit stream data structure implementation for the Phase 2 proposal. Then, the User Interface also sends the Engine a request to detect, remove, or extract the steganography, along with extra information such as the class of algorithms to be used. For the Phase 1 development, we required the extra information to be specified inside a configuration file. Next, the Format Detection module tries to detect the file type of the input file. If it fails to detect all the known file formats, the file will be classified as unknown type.

Next, depending on the request, one of the three Anti-Steganographic Modules (Detection, Removal, Extraction) will be called. As the module is invoked, all the implementations of all algorithms in the module will be executed. However, depending on the file type of the input file, some of the implementations can be skipped. For the file with unknown type, all the implementations will be run. The algorithm implementations must be able to skip the input file of the types that the algorithms do not accept.

Depending on the implementation, the file will be routed or skipped through the next three modules: Media Reader, Color Conversion/Transformation, Media Writer. For the algorithms that work with raw images, the file might be passed through the Media Reader. For the algorithms that work with files that are unsupported by Media Reader, the file can bypass the Media Reader module. Similarly, if an algorithm requires a color conversion/transformation, then the Color Conversion/Transformation module is invoked; otherwise, the module is just ignored. And if the implementation works with an input file only, the Media Writer will provide the needed file.

For the Steganography Detection module, any implementation that outputs the file instead of the confidence level, the file parser needs to be run to convert the file into the confidence level value.

For the Steganography Removal and Extraction modules, if the implementation writes the output file directly then the job is done. Otherwise, it might need the Media Writer to write the output file in a desired format. Similar to the input file, the output file can be written to a storage and the filename is returned, or it can be a data structure that holds the content of the file. Again, for the Phase 1 development, we supported only the filename and left the bit stream data structure to be implemented in the Phase 2.

2 Application Programming Interfaces (API) and Software Implementation

Because the input to the User Interface is the keyboard and the mouse, there is no API for the User Interface. To understand the APIs of the Anti-Steganographic Engine and its modules better, first let us discuss how the engine was built.

The Anti-Steganographic Engine has a total of 8 modules. Each module contains a non-extensible core and extensible components (plugins). The non-extensible cores always come with the engine. They can be updated as new implementations of the engine come along, while the extensible components can be added and removed anytime.

Figure 2 shows all the cores and extensible components of the anti-steganographic engine. All 8 modules in the engine will be implemented as the cores, so in figure 2, the cores are shown as 8 rectangular boxes. The major duty of the cores is to manage the extensible components. It controls the data flow and decides when a certain extensible component would be executed. Some of the cores do have a configuration file that not only configures the cores, but also extends the cores to certain limit. For example, the Format Detection core can search any signature at any position. As new file formats are invented, if they always contain a signature at a specific position, the signature and the position can be entered in the configuration file and the core can detect this file format right away.

All the extensible components must be implemented as runtime libraries or executables. Here, we use the Windows term Dynamic Link Libraries (DLLs) to mean runtime libraries, but they are not limited to Windows. Any kinds of runtime libraries would be acceptable. As a matter of fact, for the Phase 1 development, we implemented some extensible components as Windows DLLs as well as Linux runtime library. Since each module can take a file as input and can write output to a file, the extensible component can also be implemented as a stand-alone application or executable. That means many existing softwares can be instantly added to the module to extend the engine. In figure 2, the extensible components are shown as "DLL/EXE". All the extensible components of each module must adhere to the module API.

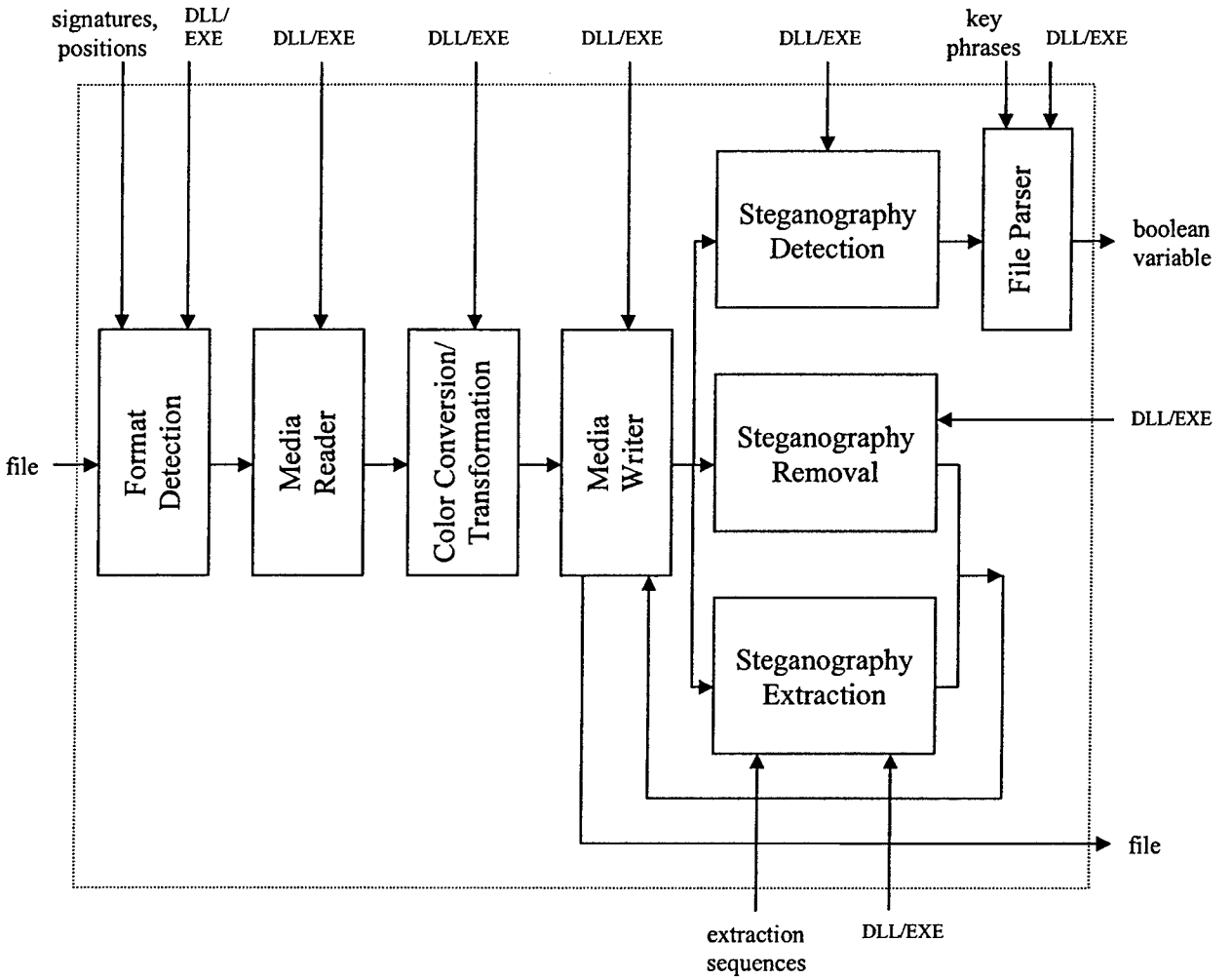

Figure 2. The cores and extensible components of the Anti-Steganographic Engine

The engine API and all the module APIs are described below.

2.1 Main Engine API

Before running the engine, the User Interface must first initialize it by calling the function ***ASEngineStart()*** that will read the engine configuration files and preload some of the extensible components. Then the User Interface can run the engine by calling ***ASEngineRun()*** as many times as desired. When done, it must call ***ASEngineStop()*** to release the memory and to unload the extensible components.

The Main Engine is implemented in the source files **ASEngine.h** and **ASEngine.c**.

```
/*
 * Start the Anti-Steganographic Engine by reading the configuration files
 * and loading the plugins.
 *
 * Return a pointer to ASENGINE structure if successful, NULL otherwise
 * (check 'error' for error code).
```

```
 */
extern ASENGINE *ASEngineStart(
    int  *error,          // status code
                          //    0 = successful
                          //   -1 = not enough memory
                          //   -2 = can't read configuration files
                          //   -3 = error parsing configuration files
    char *errormesg);     // buffer to receive error message (must be big
                          //   enough to hold 256 bytes)


/*
 * Run the Anti-Steganographic Engine on an input file.
 *
 * Return
 *  -1 : not enough memory
 *  -2 : can't open input file (doesn't exist or prohibited)
 *  -3 : can't read input file
 *  -4 : unknown input file format
 *  -5 : error running some plugin
 *  non-negative integer (0 - 100):
 *       For steganography extraction or removal, this means successful.
 *       For steganography detection, this is the confidence level
 */
extern int ASEngineRun(
    ASENGINE *engine,     // engine returned by ASEngineStart()
    char  *ifilename,     // input filename
    int   command,        // 0 = detect, 1 = extract, 2 = remove
    char  *ofilename);    // output filename (ignored if command is 0)


/*
 * Stop the Anti-Steganographic Engine by releasing the memory and unloading
 * the plugins.
 *
 * Return 0 if successful, -1 otherwise
 */
extern int ASEngineStop(ASENGINE *engine);
```

2.2 Format Detection API

This API contains only one function ***FormatDetection()***. The input to this function is the filename of the input file or a data structure that holds the content of the file. The output is a predefined integer that corresponds to the type of the file. A special number is reserved to mean unknown type. If the extensible component of this module is implemented as a stand-alone executable, the program can write either the predefine number or a text message to a file. In the case of the text message, the text message must be entered in the module configuration file with its corresponding predefined number. The core of this module does have a file parser that reads the output file and checks it against the configuration file to map the text message into the predefined number.

The core of this module can search for any signature at any position. A special number is reserved to mean the signature can reside anywhere in the file. The core configuration file should contain one entry per file format. Each file format can have any number of signatures at any positions. The core would say the file is in a certain format only if it can find all the signatures at their correct positions in the file. Note that it is OK for the core to say a file is in a certain format even if it is not, because further checks will be done by the Media Reader and/or one of the anti-steganography modules. However, it is unacceptable to say a file in a certain format that it is not in that format. In otherwords, the signatures and the positions are the necessary but not sufficient conditions for the file to be in the defined format.

The Format Detection module is implemented in the source files **FormatDetection.h** and **FormatDetection.c**.

```
/*
 * Detect the file format of the given file.
 *
 * This is the Format Detection Core function.  It is also the function
 * required by the Format Detection Plugin.  To implement the Format
 * Detection Plugin, just implement this function and compile it as a DLL.
 *
 * Return one of the followings
 *    0 if file format is unknown
 *   -1 if not enough memory
 *   -2 if file can not be opened
 *   -3 if error reading the file
 *   a positive integer if successfully detect the file format.  The returned
 *      integer must be defined in the Anti-Steganographic Engine's file
 *      format configuration file.  For a new file format (not yet supported
 *      by the engine), simply define it in the configuration file and the
 *      engine will recognize it.
 */
EXPORT int FormatDetection(char *filename);
```

2.3 Media Reader API

This API contains two functions: ***MediaReader()*** and ***FreeImage()***. The input to the function ***MediaReader()*** is the filename of the input file or a data structure that holds the content of the file. For the Phase 1 development, we used the filename of the input file only. The output is a data structure that represents a known image or media type. For the Phase 2 implementation, the output can be optionally a file that holds the known image or media in a serial data format. The function ***FreeImage()*** is intended to free the memory allocated for the data structure that ***MediaReader()*** returns. It should be called ***FreeMedia( )*** to match with the function name ***MediaReader()***; however, we have not changed the name yet.

The Media Reader module is implemented in the source files **MediaReader.h**, **MediaReader.c**, **Image.h**, and **Image.c**.

```
/*
 * Read the image from the given file.
 *
 * This is the Media Reader Core function.  It is also the function required
 * by the Media Reader Plugin.  To implement the Media Reader Plugin,
 * implement this function AND the function FreeImage() defined in the file
 * Image.h  (file Image.c can be used for this purpose) then compile both of
 * the functions as a DLL.
 *
 * Return one of the followings
 *    0 if file format is unknown
 *   -1 if not enough memory
 *   -2 if file can not be opened
 *   -3 if error reading the file
 *   a positive integer if successfully read the file.  The returned integer
 *      is the file format of the file and it must be defined in the
 *      Anti-Steganographic Engine's file format configuration file.  For a
 *      new file format (not yet supported by the engine), simply define it
 *      in the configuration file and the engine will recognize it.
 */
EXPORT int MediaReader(
```

```
    IMAGE **image,          // buffer to receive the returned image
    char  *filename);       // file to read the image from


/*
 * Free the image returned by MediaReader().
 */
EXPORT void FreeImage(IMAGE *image);
```

2.4 Color Conversion/Transformation API

This API has only one function: ***ConvertColorSpaceImage()***. Both the input and output of this function is the filename of the known image or media in a serial data format, or a data structure that represents a known image or media type. For the Phase 1 development, this function only accepts the image/media data structures as arguments.

The Color Conversion/Transformation module is implemented in the files **ColorTrans.h** and **ColorTrans.c**.

```
/*
 * Convert the color space of an image.
 *
 * This is the Color Conversion/Transformation Core function.  It is also the
 * function required by the Color Conversion/Transformation Plugin.  To
 * implement the Color Conversion/Transformation Plugin, implement this
 * function AND the function FreeImage() defined in the file Image.h (file
 * Image.c can be used for this purpose) then compile both of the functions
 * as a DLL.
 *
 * Return one of the followings
 *    0 if successful
 *   -1 if not enough memory
 *   -2 if target color space is unknown
 *   -3 if conversion can not be done (wrong image data)
 *   -4 if the conversion is not yet supported
 */
int ConvertColorSpaceImage(
    IMAGE *image_in,        // input image
    IMAGE **image_out,      // buffer to receive the output image
    int   target_color)     // target color space.  The color space is an
                            //   integer that must be defined in the
                            //   Anti-Steganographic Engine's color space
                            //   configuration file.  For a new color space
                            //   (not yet supported by the engine), simply
                            //   define it in the configuration file and the
                            //   engine will automatically recognize it
```

2.5 Media Writer API

This API has only one function: ***MediaWriter()***. The input of this function is the filename of the known image or media in a serial data format, or a data structure that represents a known image or media type. For the Phase 1 development, this function only takes the image/media data structure as input. The output is the filename of an image or media file.

The Media Writer module is implemented in the files **MediaWriter.h** and **MediaWriter.c**.

```
/*
 * Write the image to the given file.
```

```
 *
 * This is the Media Writer Core function.  It is also the function required
 * by the Media Writer Plugin.  To implement the Media Writer Plugin, simply
 * implement this function and compile it as a DLL.
 *
 * Return one of the followings
 *    0 if successful
 *   -1 if not enough memory
 *   -2 if file can not be opened
 *   -3 if error writing the file
 *   -4 if file format is not supported by this function
 */
EXPORT int MediaWriter(
    IMAGE *image,           // image to be written to the file
    char  *filename,        // file to write the image to
    int   fileformat,       // file format to write the image in (see the
                            //   Anti-Steganographic Engine's file format
                            //   configuration file)
    int   quality_factor);  // JPG quality factor between 0 (Worst) and
                            //   100 (Best) inclusive.
                            // Recommended value = 75.
                            //   Ignored if fileformat != JPG
```

2.6 Steganography Detection API

This API has only one function: ***StegDetection()***. The function takes two different kinds of input: the filename of the input file or a data structure that holds the content of the file, and the filename of the known image or media in a serial data format, or a data structure that represents a known image or media type. For the Phase 1 development, this function accepts only the image/media data structure. The output is the confidence level that the steganography is present in the input file. Alternatively, the extensible component of this module can write a text message to the output file. In this case, a keyphrase must be entered into the configuration file of the File Parser module, or an extensible component must be added to the File Parser module to intepret the output file.

The Steganography Detection module is implemented in the files **StegDetection.h** and **StegDetection.c**.

```
/*
 * Detect steganography in the image and return the confidence level that
 * the image contains steganography
 *
 * This is the Steganography Detection Core function.  It is also the
 * function required by the Steganography Detection Plugin.  To implement the
 * Steganography Detection Plugin, simply implement this function and compile
 * it as a DLL.
 *
 * Return one of the following values:
 *   0: successful
 *   1: detection algorithm has some problem with this image, but it still
 *        can give the best estimate of the confidence level
 *  -1: not enough memory
 *  -2: detection algorithm doesn't work with this image
 *  -3: other kinds of error while running the detection algorithm
 */
EXPORT int StegDetection(
    IMAGE *image,           // image data passed from the Anti-Steg Engine
    float *confidence);     // output confidence level in the range [0-1]
```

2.7 Steganography Removal API

This API has only one function: ***StegRemove()***. Both the input and output of this function are the same as the input to the Steganography Detection module.

The Steganography Removal module is implemented in the files **StegRemoval.h** and **StegRemoval.c**.

```
/*
 * Remove steganography from the image.
 *
 * This is the Steganography Removal Core function.  It is also the function
 * required by the Steganography Removal Plugin.  To implement the
 * Steganography Removal Plugin, simply implement this function and compile
 * it as a DLL.
 *
 * If the function is successful, allocate the memory for the new image and
 * return its pointer.  If unsuccesful, return NULL and fill in one of the
 * following values for the variable error:
 *   0: no error
 *   1: the steganography doesn't exist (according to this algorithm)
 *  -1: not enough memory
 *  -2: error running the removal algorithm
 */
EXPORT IMAGE *StegRemove(IMAGE *image, int *error);
```

2.8 Steganography Extraction API

This API has only one function: ***StegExtraction()***. The input of this function is the same as the input of the Steganography Removal module. The output is a filename that will contain the extracted steganography or a data structure that represents the extracted steganography. For the Phase 1 development, the function takes only filename.

To extract the steganography inserted by simple steganography softwares that embed the steganography content in a well defined places inside the cover files, there is no need to write any DLL or EXE to extract it. Simply enter the extraction sequences in the configuration file of this module, and the core will extract the steganography content correctly.

The Steganography Extraction module is implemented in the files **StegExtraction.h** and **StegExtraction.c**.

```
/*
 * Extract steganography from the image and write it to the output file.
 *
 * This is the Steganography Extraction Core function.  It is also the
 * function required by the Steganography Extraction Plugin.  To implement
 * the Steganography Extraction Plugin, simply implement this function and
 * compile it as a DLL.
 *
 * Return one of the following values:
 *   0: no error
 *   1: the steganography doesn't exist (according to this algorithm)
 *  -1: not enough memory
 *  -2: error running the extraction algorithm
 */
EXPORT int StegExtraction(IMAGE *image, char *filename);
```

2.9 File Parser API

This API has only one function: ***FileParse()***. The input of this function is the filename of the output file written by a Steganography Detection extensible component. The output is the confidence level that a steganography content is embedded inside the input file.

Whenever a Steganography Detection extensible component that writes the output to a file is added to the engine, an extensible component must be added to the File Parser to support that Steganography Detection component. Fortunately, in most cases when the Steganography Detection component always writes a certain message to mean there is a steganograph in the cover file, and a certain message to mean there is not, these messages can be entered into the configuration file of the File Parser module and its core will automatically support the Steganography Detection component.

The File Parser module is implemented in the files **FileParser.h** and **FileParser.c**.

```
/*
 * Parse the file to retrieve the Steganography Confidence Level.  The file
 * is the output of some Steganography Detection executable.
 *
 * This is the File Parser Core function.  It is also the function required
 * by the File Parser Plugin.  To implement the File Parser Plugin, just
 * implement this function and compile it as a DLL.
 *
 * Return one of the following values:
 *  -1: not enough memory
 *  -2: can't open the file
 *  -3: can't read the file
 *  -4: error during the parsing
 *  non-negative integer (0 - 100): confidence level that the image contains
 *                                  steganography
 */
EXPORT int FileParse(char *filename);
```

2.10 Internal APIs

All the APIs described above are external APIs that define the interface to the Anti-Steganographic Engine. Internally, the engine also has a set of APIs that different C modules would use to communicate with each other. All the C files that do not implement the external APIs can be combined into 3 major groups: the Configuration Files Manager, the Plugins Manager, and the Main Engine. The Configuration Files Manager is responsible to read and maintain the engine's configuration files. The Plugins Manager manages all the plugins and is able to execute any of the plugins. The Main Engine controls all the data flows inside the engine. To do that, it must control the Configuration Files Manager via the Configuration Files API, and the Plugins Manager via the Plugins API. It also controls the core modules using the Core APIs. Note that the Core APIs are identical to the external APIs described earlier, so there are only 2 sets of internal APIs: the Configuration Files API, and the Plugins API.

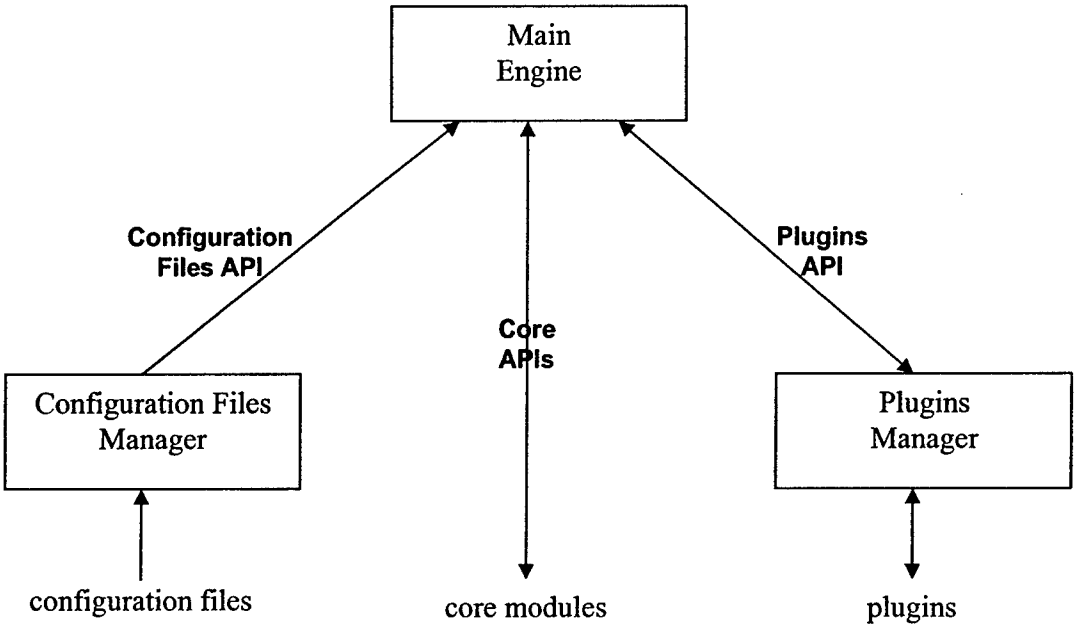

Figure 2.10 The communications between Main Engine and 2 Managers plus core modules

Figure 2.10 shows the APIs between the Main Engine and the 2 managers plus the core modules. We now describe the 2 internal APIs in details.

2.10.1 Configuration Files API

This API has two functions: ***ParseColorAndFormat()*** and ***ParseNextPlugin()***. The function ***ParseColorAndFormat()*** reads the file formats and the color spaces configuration files, while the function ***ParseNextPlugin()*** is designed to read the plugins configuration file.

The Configuration Files API is implemented in the source files **parser.h, parser.c, utils.h, utils.c, ColorAndFormat.h, ColorAndFormat.c**. There are some APIs between these files, but we will not describe them here. They can be found in the forementioned **.h** files.

```
/*
 * Parse the file format and color space configuration files
 *
 * Return the link list containing all definitions of file formats or
 * color spaces if successful, or NULL otherwise (check the error code and
 * error message). Note that if the configuration file doesn't have any
 * definition, the returned value is also NULL but *error is set to 0.
 */
extern COLORANDFORMAT *ParseColorAndFormat(
    FILE *fp,           // configuration file to read from
    int  *error,        // status code
                        //    0 = successful
                        //   -1 = not enough memory
                        //   -2 = can't read configuration file
                        //   -3 = error parsing configuration file
    char *errormesg);   // buffer to receive error message (must be big
```

```
                        //   enough to hold 256 bytes)

/*
 * Read the next plugin from the file
 *
 * Return the pointer to the newly created PLUGIN if successful, or NULL
 * otherwise (check the error code).
 */
extern PLUGIN *ParseNextPlugin(
    FILE *fp,           // configuration file to read from
    int  *linenum,      // line number of next line in the file
    COLORANDFORMAT *formats,
                        // list of file formats read
                        // from format config. file
    COLORANDFORMAT *colors,
                        // list of color spaces read from colors config. file
    int  *error,        // status code
                        //    0 = no more plugin
                        //   -1 = not enough memory
                        //   -2 = can't read configuration file
                        //   -3 = error parsing configuration file
    char *errormesg);   // buffer to receive error message (must be big
                        //   enough to hold 256 bytes)
```

2.10.2 Plugins API

This API contains two types of functions: the first is a group of functions that manage the plugins, whereas the second group consists of functions that execute the plugins.

The functions that manage the plugins are: ***AllocPlugin()***, ***FreePlugin()***, ***LoadPlugin()***, and ***AppendPluginToList()***. The descriptions of these functions can be found in the file **Plugin.h**.

The functions that execute the plugins have the following formats: ***RunNAMEPlugin()*** and ***RunNAMEPluginEXE()***, where ***NAME*** is the name of the plugin to be executed. The functions ***RunNAMEPlugin()*** runs the DLL plugins, while the functions ***RunNAMEPluginEXE()*** executes the executable plugins.

The Plugins API is implemented in the source files **Plugin.h**, **Plugin.c**, **utils.h**, **utils.c**, **Image.h**, and **Image.c**. There are some APIs between these files, but we will not describe them here. They can be found in the forementioned **.h** files.

```
/*
 * Allocate memory for the plugin
 *
 * Return the pointer to the newly created PLUGIN data structure if
 * successful, or NULL if not enough memory.
 */
extern PLUGIN *AllocPlugin(void);


/*
 * Free memory allocated for the plugin.  Also unload the plugin if it has
 * been loaded into memory.
 */
extern void FreePlugin(PLUGIN *plugin);
```

```
/*
 * Load Plugin into memory.  It stays in memory until the Plugin is freed.
 *
 * Return 0 if successful, -1 otherwise
 */
extern int LoadPlugin(PLUGIN *plugin);


/*
 * Append plugin after the link list of plugins
 */
extern void AppendPluginToList(PLUGIN *plugin, PLUGIN **list);


/*
 * Run the XXXX DLL Plugin
 * The arglist must be the same as arglist in the DLL's required Core
 * function.
 *
 * Return -100 if the plugin doesn't have the required Core function, -101
 * if it doesn't have the function "FreeImage"; otherwise return the same
 * value returned by the plugin's Core function
 */
extern int RunXXXXPlugin(PLUGIN *plugin, arglist...);


/*
 * Run the XXXX EXE Plugin
 *
 * Return -100 if the plugin command line doesn't have the keyword %INPUT% or
 * %OUTPUT% and 0 otherwise.
 */
extern int RunXXXXPluginEXE(PLUGIN *plugin,char *ifilename, char *ofilename);
```

2.11 Data Structures

There are many data structures in our code. Here, we only mention some major data structures. All other data structures can be found in the **.h** files. First, there is a main data structure for the Anti-Steganographic Engine. This data structure holds one instance of the engine and contains all the information about the engine. This is the same data structure that is returned by the function ***ASEngineStart()*** to the User Interface module. It is required to run the engine on any input file. Note that the User Interface can run multiple instances of the engine in parallel by creating and executing concurrently multiple data structures for the engine.

The main Anti-Steganographic Engine data structure is called ***ASENGINE*** and is defined in the source file **ASEngine.h**.

```
typedef struct {
    COLORANDFORMAT *fileformats;// link list of file format definitions
    COLORANDFORMAT *colorspaces;// link list of color space definitions

    PLUGIN *piFormatDetection;  // link list of all Format Detection Plugins
    PLUGIN *piMediaReader;      // link list of all Media Reader Plugins
    PLUGIN *piColorConversion;  // link list of all Color Conversion Plugins
    PLUGIN *piMediaWriter;      // link list of all Media Writer Plugins
    PLUGIN *piStegDetection;    // link list of all Steg. Detection Plugins
    PLUGIN *piStegRemoval;      // link list of all Steg. Removal Plugins
    PLUGIN *piStegExtraction;   // link list of all Steg. Extraction Plugins
```

```
    COREFORMATINFO *format_sigs;// link list of signatures and positions to
                               //   detect file format
    CORESTEGEXTRACTINFO *stegextract_orders;
                               // link list of orders to extract stego
} ASENGINE;
```

The ***COLORANDFORMAT*** data structure implements the link list that holds either the definitions of the file formats or the definitions of the color spaces. It is defined in the file **ColorAndFormat.h**.

```
typedef struct CNFNODE {     // pair of string code and number code that
                             // defines file format or color space
    char *scode;                 // string code
    int  ncode;                  // number code
    struct CNFNODE *next;        // to implement link list
} COLORANDFORMAT;
```

The ***PLUGIN*** data structure implements the link list that holds the information of all the plugins. It is defined in the file **Plugin.h**.

```
typedef struct PINODE {      // Plugin
    int  piclass;                // one of PI_CLASS_* define above
    int  pitype;                 // one of PI_TYPE_* defined above
    char *id;                    // name to distinguish from other plugins
    char *command;               // command line if pitype == PI_TYPE_EXE
                                 //   or DLL file name if pitype ==
PI_TYPE_DLL
    INTARRAY *fileformats;       // file formats that can be processed by
                                 //   Format Detection, Media Reader, Media
                                 //   Writer, and 3 EXE_STEG* plugins
    INTARRAY *colorspaces;       // color spaces that can be processed by
                                 //   3 DLL_STEG* plugins
    INTARRAY *fromcolorspaces;   // color spaces that can be converted from
                                 //   by Color Conversion plugin
    INTARRAY *tocolorspaces;     // color spaces that can be converted to
                                 //   by Color Conversion plugin
    char     *fileparser;        // file name of DLL that will parse the
                                 //   output written by this plugin.
                                 //   Applicable only if this plugin is
                                 //   a Steg Detection EXE

    void     *hPluginDLL;        // handle to the plugin DLL after it's loaded
                                 //   into memory (NULL if not yet loaded)

    struct PINODE * next;        // to implement link list
} PLUGIN;
```

The ***COREFORMATINFO*** data structure implements the link list that holds the signatures and positions to detect certain file formats. It is defined in the file **Core.h**.

```
typedef struct CFNODE { // Specific Info for Format Detection Core
```

```
    int  fileformat;        // file format that has signature & position
                            // below:
    char *signature;        // the signature
    int  position;          // offset from the begining of the file
                            //   (-1 = anywhere in the file)
    struct CFNODE *next;    // to implement link list
} COREFORMATINFO;
```

The ***CORESTEGEXTRACTINFO*** data structure implements the link list that holds the orders to extract certain steganographies. It is defined in the file **Core.h**.

```
typedef struct CSNODE { // Specific Info for Steganography Extraction Core
    int  *order;            // array of positions to extract steganography
    int  npositions;        // size of the array 'order'
    struct CSNODE *next;    // to implement link list
} CORESTEGEXTRACTINFO;
```

There is also another major data structure in the code, and it is the data structure that represents the media file. For the Phase 1 development, we called this data structure ***IMAGE*** because it only supports image files. For the Phase 2, we will extend this data structure to other media files and will rename it accordingly.

The ***IMAGE*** data structure holds all the information about the image file. It is defined in the file **Image.h**.

```
typedef struct {        // data structure for color palette
    int numcolors;          // number of colors in the palette
    int colorspace;         // color space of the field 'colors' below (see
                            //   the Anti-Steganographic Engine's color space
                            //   configuration file)
    unsigned char *colors;  // all colors in the palette color map
} PALETTE;


typedef struct {        // data structure for image
    int  width;             // image width
    int  height;            // image height
    int  bpp;               // number of bits per pixel
    int  colorspace;        // color space of the field 'data' below (see
                            //   the Anti-Steganographic Engine's color space
                            //   configuration file)
    PALETTE *palette;       // color palette if applicable
    unsigned char *data;    // image data

    // The following fields are specific for the Anti-Steg. Engine only
    char filename[_MAX_PATH+1];
                            // filename where this IMAGE is orig. read from
    int  fileformat;        // original image file format (see the
                            //   Anti-Steganographic Engine's file format
                            //   configuration file)
    void *freeImageFunc;    // if this image data structure is allocated by a
                            //   DLL plugin, this field is the DLL plugin's
                            //   function that must be used to free the image
    int  sem;               // semaphore for synchronization
} IMAGE;
```

3 Source Code

Our source code has been debugged and tested under Windows and RedHat Linux 7. We are providing the Microsoft Visual C++ project files to compile the code under Windows, and the makefiles to compile it under RedHat Linux. The code is highly portable and it should take a minimum amount of time to compile it for different Operating Systems; however, for the Phase 1 development, we only compiled and tested our code under Windows and RedHat Linux 7 only.

3.1 File/Directory Structure

There are 7 directories in our code distribution:

1. **ASEngine**: this directory contains the source code for the Anti-Steganographic Engine
2. **Demo**: this directory contains the demo of the Phase 1 development. The subdirectory Windows contains the demo to be run on Windows , while the subdirectory Linux is for Linux Operating System.
3. **Image**: this directory contains the source code to read and write image files. Many projects including ASEngine, MediaReader Plugin, and MediaWriter Plugin share this source code.
4. **Plugins**: this directory contains the source code to implement the following DLL plugins:
    a. **FormatDetectionPlugin**: a Format Detection plugin
    b. **MediaPlugins**: a Media Reader plugin and a Media Writer plugin
    c. **StegDetectionPlugin**: a Steganography Detection plugin
    d. **ColorTransPlugin**: a Color Transformation plugin
    e. **FileParserPlugin**: a File Parser plugin
5. **stegoapp**: this directory contains the source code to implement a Steganography Detection algorithm. The executable created from this source code can be used as a Steganograpy Detection EXE plugin. The source code here is also used by the Steganography Detection DLL plugin in the directory **Plugins**.
6. **TestASEngine**: a console application that provides a simple UI to test the Anti-Steganographic Engine.
7. **TestImage**: a console application that tests the Image library in the directory **Image**.
8. **Utils**: some common utilities that are needed by various projects.

3.2 Windows Projects

There are 9 Windows projects created for the Phase 1 development. They were built from the Microsoft Visual C++ 6.0 and can be opened from the Microsoft Visual C++ 6.0 as well as Microsoft Visual C++ .NET (automatically provides the project conversions). The best way to open all 7 projects is to open the workspace **AntiSteg.dsw** in the main directory of the source code. To build all the projects, select *Build->Batch Build.*

Here are the descriptions of all the projects:

1. **ASEngine**: the Anti-Steganographic Engine project. This project creates a library that can be linked with some User Interface to create an Anti-Steganographic application.
2. **ColorTransPlugin**: this project creates a Color Space Transformation DLL plugin
3. **FileParserPlugin**: this project creates a File Parser DLL plugin
4. **FormatDetectionPlugin**: this project creates a Format Detection DLL plugin
5. **Image**: the Image project. This project creates the Image library that reads and writes image files in a number of image file formats. It currently supports TIFF, JPEG, BMP, and GIF formats.
6. **MediaPlugins**: this project creates a DLL that can be used as both a Media Reader plugin and a Media Writer plugin. This project requires the Image library.

7. **StegDetectionPlugin**: this project creates a Steganography Detection DLL plugin.
8. **TestASEngine**: this project creates a console application that tests the Anti-Steganographic Engine. This project requires the ASEngine library and the Image library.
9. **TestImage**: this project creates a console application that tests the Image library. It requires the Image library.

3.3 Linux Projects

The Linux projects created for the Phase 1 development are exactly the same as the Windows projects, except that instead of the Microsoft Visual C++ project files, the makefiles are created to compile the code on Linux. To build all the projects, run *make* on the makefile in the main directory of the source code.

4 How to Maintain and Extend the Software

The Anti-Steganographic Engine can be extended by implementing the extensible component (or plugin) of any of its modules. For example, if we write a plugin for the Media Reader to read a new image file format and plug it into the engine, the engine now can support a new file type, namely the new image file format, without being recompiled.

To write a DLL plugin for a module, all we have to do is implement the API of that module. Note that all the APIs are very simple. They contain only one or two functions. That would simplify the implementation very much. Moreover, the modular design of these modules allows the plugin implementor to focus only on the module that s/he aims for. For example, to implement a steganography detection plugin, the writer needs only to focus on the steganography detection algorithm, without worrying about how to read the image or how to convert the color space to the one the algorithm requires.

Even though all the extensible components should be implemented as DLLs for speed, the implementor still has another flexibility by implementing them as EXEs. Even when the implementor is not aware of this Anti-Steganographic Engine, her or his software can be used as the EXE plugin. For example, a steganography researcher just invented a reliable algorithm that can detect the existence of a steganography if it is embedded anywhere in any uncompressed image. The researcher wrote a program to demonstrate the algorithm on TIFF image files. This program can certainly be used as an EXE plugin to our Anti-Steganographic Engine. Furthermore, if the engine has the Media Reader plugins to read other image file formats, say BMP or TARGA, the new algorithm now can instantly run on these files too, not just TIFF.

4.1 How to create the Plugins

To create a DLL plugin for any module, simply implement the module's API. For example, to implement the Steganography Detection DLL plugin, we only need to write the function ***StegDetection()*** (see section 2.6) then compile it as a DLL.

To create an EXE plugin for any module, write the appropriate application using only files as input and output.

4.2 How to install the Plugins

Once a plugin is built, it can be plugged into the Anti-Steganographic Engine by modifying the plugins configuration file. For the Phase 2 proposal, we will have a nice User Interface to manage the configuration file. However, for the Phase 1 development, we have to edit the file manually.

The plugins configuration file is called **plugins.conf**. A plugin can be installed by adding a few lines to the file with the following rules:

1. A plugin is defined with a series of commands
2. All the commands are line-based with one command per line
3. A plugin must start with the command "**PluginBegin** PluginClass PluginType"
   where PluginClass must be one of the followings:

```
FormatDetection   // to specify the plugin as Format Detection Plugin
MediaReader       // to specify the plugin as Media Reader Plugin
ColorConversion   // to specify the plugin as Color Conversion Plugin
MediaWriter       // to specify the plugin as Media Writer Plugin
StegDetection     // to specify the plugin as Steganography Detection
                  // Plugin
StegRemoval       // to specify the plugin as Steganography Removal
                  // Plugin
StegExtraction    // to specify the plugin as Steganography
                  // Extraction Plugin
```

and PluginType must be one of the followings:

```
DLL               // to specify the plugin as DLL plugin
EXE               // to specify the plugin as EXE plugin
```

4. Plugin ends with the command "**PluginEnd**"
5. Between the two commands "**PluginBegin**" and "**PluginEnd**", the plugin can have any number of commands in the format "Name = Value" where Name and Value must be one of the followings:

```
ID = any string       // name to distinguish from other plugins
Command = c:\t.dll    // EXE command line or DLL filename.
                      // Use the keywords %INPUT%, %OUTPUT% to
                      // specify the input and output
                      // filenames for the EXE plugin
FileFormats = TIF BMP      // list of file formats accepted by
                           // this plugin.
                           // The string codes must be defined
                           // in the file format configuration file.
ColorSpaces = RGB MONO     // list of color spaces accepted by this
                           // plugin
                           // The string codes must be defined in
                           // the color space configuration file
FromColorSpaces = RGB      // list of color spaces that the Color
                           // Conversion plugin can convert from
ToColorSpaces = YUV HSV    // list of color spaces that the Color
                           // Conversion plugin can convert to
FileParser                 // filename of DLL that will be
                           //used to parse the output written by
                           // the Steganography Detection
                           // EXE plugin
```

6. Comments (everything in the line after "//") and blank lines are ignored
7. All the keywords are case-INsensitive (pluginBEgin is same as PluginBegin)

The execution order of the plugins will be the same order specified in the configuration file. Here are the examples of some installed plugins:

```
// A Format Detection DLL plugin
PluginBegin FormatDetection DLL
  ID = IDzap Format Detection DLL Plugin  // name of this plugin
```

```
  Command = FormatDetectionPlugin.dll      // plugin filename
  FileFormats = TIF GIF JPG BMP            // file formats this plugin
                                           // can process
  ColorSpaces = RGB                        // color spaces this plugin can
                                           // process
PluginEnd

// A Media Reader DLL plugin
PluginBegin MediaReader DLL
  ID = IDzap Media Reader DLL Plugin       // name of this plugin
  Command = MediaPlugins.dll               // plugin filename
  FileFormats = BMP TIF GIF GIF            // file formats this plugin
                                           // can read
PluginEnd

// A Steganography Detection DLL plugin
PluginBegin StegDetection DLL
  ID = IDzap LSB Stego Detection DLL Plugin     // name of this plugin
  Command = StegDetectionPlugin.dll             // plugin filename
  ColorSpaces = RGB                             // color space this plugin
                                                // works on
PluginEnd

// A Steganography Extraction EXE plugin
PluginBegin StegExtraction EXE
  ID = IDzap LSB Stego Extraction EXE Plugin    // name of this plugin
  Command = stego.exe -x %INPUT% %OUTPUT%       // command line to run
                                                // EXE file
  FileFormats = BMP                             // file format this plugin
                                                // works on
PluginEnd
```

5 Automatic Engine Update

The Steganography Engine Installer is to provide a user-friendly interface for the installation and update of the anti-steganographic application package in MS Windows environment with security measures. Rather than relying on the IE-only and virus-prone ActiveX technologies, we developed a browser-platform independent client-server application to carry out the tasks.

The client application is a small piece of code that we shall refer to as **pre-installer**. Upon clicking a link at our web-site, the user receives the pre-installer, which then communicates with the server to carry out further tasks. The server application may reside in a single machine or a cluster of machines.

The class **TicketManager** is used by both the client and the server program. It handles the common information needed for valid communication between the client and the server. When the pre-installer (client) begins execution, it first checks if the user's machine already has the current or a newer version of the steganography engine. Its interface is shown below:

```
class TicketManager {
public:
  TicketManager();              //constructor
  char *getTicket();           //gets an installation ticket used by  both
                               // client and server
  char *getTicketu();          //gets an update ticket used by both client
                               // and server
```

```
  bool  checkTicket( char *t );      //server checks if received ticket
                                     // is correct
  char *versionFile();
  char *getVersion();          //gets latest installation version
  char *getVersionu();         //gets latest update version
private:                       //these are needed by both clients and server
  char *ticket;                //ticket for installation
  char *ticketu;               //ticket for update
  char *version;               //engine version associated with installation
  char *versionu;              //engine version associated with update
};
```

The **TicketManager** acts as a middle tier in the program hierarchy. A change of its implementation will not affect the client or server implementation and vice versa. It thus provides a lot of flexibility and robustness to the whole application.

If the user's machine does not have a copy of the engine or only have older versions, the pre-installer will then communicate with the server and try to retrieve the installer from it. It first obtains a ticket from a **TicketManager** object and sends the ticket to the server.

When the server receives a request from any source, it forks off a child process to handle the request and resumes listening to further requests.

Upon receiving the ticket, the child process verifies if it is a valid ticket by checking the received ticket, which is a sequence of characters, against some preset values via the member function **checkTicket()** of **TicketManager**. If the incoming ticket is valid, the **TicketManager** will further determine if it signifies an installation or an update request. On the other hand, if it is invalid, the ticket could be simply a garbage message sent by a malicious source to probe our server and could be an origin of attack. To fend off any possible attack or malicious act, the process sleeps for two seconds and then exits without doing anything. The purpose of sleeping two seconds is to deceive the attacker which then could not know if the slow response is due to a slow Internet connection or some other unknown underlying reasons; moreover, the attacker is tied up to our server during this period, and it is difficult for it to make diagnostics or making numerous connection trials to send a correct ticket.

Thus the ticket serves as an entrance pass; it is similar to a serial number or a key of a commercial software package that a user has to enter in order to install the package.

If the incoming ticket is valid, the server will send the self-extracted installation or update **installer** to the client (pre-installer). After the client has received the complete installer, it executes it and passes control to it. The installer is GUI-based and user-friendly, which guides the user step-by-step to install or update the steganogaphy engine and associated software applications in her machine. After the installer has finished execution, the pre-installer regains control and displays any necessary round-up messages to the user.

6 Software Demo

There are 3 types of binary code that are created from the source code: the C libraries, the DLL plugins, and the executables. Because the C libraries have been compiled into the executables, we don't have to demonstrate them, we simply demonstrate the executables. We have placed all the DLL plugins and the executables into a common directory called **Demo** and created a number of configuration files for the demonstration purposes. Note that the directory **Demo** actually contains two subdirectories **Windows** and **Linux** that contain the same demo for Windows and Linux 7 respectively. Only the binary code is different between the two Operating Systems, everything else is the same.

Here are the list of all the files in the **Demo** directory

```
colors.conf                : the engine color space configuration file
```

```
formats.conf                : the engine file format configuration file
plugins.conf                : the engine plugin configuration file
fsigpos.conf                : list of signatures & positions to detect
                              common file formats
stegseq.conf                : list of sequences to extract simple stego
FormatDetectionPlugin.dll/so    : our Format Detection plugin
MediaPlugins.dll/so         : our Media Reader and Writer plugins
StegDetectionPlugin.dll/so      : our steganography detection plugin
ColorTransPlugin.dll/so     : Color Conversion/Transformation plugin
FileParserPlugin.dll/so     : a sample File Parser plugin
stego.exe/stego             : our steganography extraction program. We will
                              try this program as a plugin.
test.bmp                    : test image containing LSB steganography
test.tif                    : same as test.bmp but in TIFF format
TestASEngine.exe/TestASEngine   : Program to test the Anti-Stego Engine
README.TXT                  : instructions to run the demo
```

To run the demo, we will run the program TestASEngine.exe (or TestASEngine for Linux). Simply type "TestASEngine" and the program will show its syntax.

When invoked, TestAsEngine.exe first reads all 5 configuration files. The color space configuration file defines all color spaces that are used by the engine and all the plugins. Similarly, the file format configuration file defines the file formats. The plugin configuration file defines all the available plugins. The fsispos.conf file contains the list of signatures and positions to detect common file formats. The stegseq.conf file is the list of sequences to extract simple steganographies (not yet in use). Please view the 5 configuration files for details. Next, the engine examines the input file then calls the appropriate plugins to carry out the required operation (one of the 3 steganography operations: detection, extraction, and removal).

Here are the detailed steps to run the demo:

1. Try to detect the steganography on the test file 'test.bmp' by typing

   ```
   TestASEngine d test.bmp
   ```

   As you can see, the engine first runs a Format Detection plugin to check the file format of the input file. Then it calls an appropriate Media Reader plugin to read the image into an internal data structure. Finally, it sends the internal data structure to the Steganography Detection plugin to perform the detection.

2. Try to extract the steganography from the test file 'test.bmp' by typing

   ```
   TestASEngine e test.bmp
   ```

   The output would be "No Steganography Extraction Plugin available" because we don't have any Steganography Extraction Plugin yet. Fortunately, we have the executable program 'stego.exe' that can extract the steganography from any BMP files (actually, our stego.exe program can process JPEG, TIFF, and GIF file too, but for the purpose of this demo, let's assume it reads only BMP file). We can use this executable as an Steganography Extraction plugin by modifying the configuration file 'plugins.conf'. Please edit the file 'plugins.conf' and remove the "//" from the last 5 lines. Note that the command "FileFormats" lists only BMP as the acceptable file format by the executable. Now, rerun

   ```
   TestASEngine e test.bmp
   ```

   The extraction is carried out successfully. Now try

   ```
   TestASEngine e test.tif
   ```

Here, the input file is a TIFF file, but the plugin 'stego.exe' can only read BMP file, so the engine has to run the Media Reader and Media Writer plugins to convert the input file to BMP before passing it to the plugin 'stego.exe'.

We have just demonstrated that our engine can extend the algorithm implemented in 'stego.exe' to more file formats without rewriting 'stego.exe'. Besides the file formats, the extension also applies to the color spaces as we will see in the next 2 demo steps.

3. Modify the configuration file 'plugins.conf' as follows: the 8th line from the bottom of the file says "ColorSpaces = RGB". Change it to "ColorSpaces = HSV LAB YUV XYZ".

   What we just did was to tell the engine that the IDzap LSB Stego Detection DLL Plugin works on any of the four color spaces HSV, LAB, YUV, and XYZ. Now run

   ```
   TestASEngine d test.tif
   ```

   This time, because the image file test.tif is an RGB image, but the IDzap LSB Stego Detection DLL Plugin doesn't work on the RGB color space (that was what we told the engine), the engine tries to convert the image into one of the color spaces that are acceptable by the Plugin. It tries RGB->HSV and RGB->LAB unsuccessfully because the available Color Conversion Plugins do not support them yet. Finally, it succeeds when it tries RGB->YUV. The new YUV image is then passed to the IDzap LSB Stego Detection Plugin.

   We have demonstrated the ability to extend a Stego Plugin to support more color spaces without reimplementing the Stego algorithm.

   Note that the Stego Detection Plugin returns Confidence Level = 0 because it was in fact designed to run on RGB image, not YUV image. Here, we fooled the engine just for the demonstration purpose. Make sure to restore the line "ColorSpaces = RGB".

7 Steganography Research

We have focused on three tasks in this project. The fist was benchmarking of steganalysis techniques to determine their efficacy and cost performance trade-offs. Using the results of benchmarking, we then looked at the problem of 1) Computing a detectability metric for image steganography and 2) A preliminary study of fusion techniques. Detailed results on each of these three tasks are presented below:

7.1. Benchmarking

7.1.1. Data Set: In the benchmarking project, our goal was to compare a number of universal steganalysis techniques proposed in the literature which include techniques based on binary similarity measures, wavelet coefficients' statistics, and DCT based image features. These universal steganalysis techniques are tested against a number of well know embedding techniques, including Outguess, F5, Model based, and perturbed quantization. Our experiments as mentioned in the previous reports were done using a large dataset of JPEG images, obtained by a random crawl of publicly available websites.
We then is discarded all but the medium quality images (shown in red below) and cleaned this set by removing contraband images. Also the maximum number of images in each category was capped to a maximum of 100 thousand images.

Pixels\Quality	High (90~100)	Medium (75~90)	Low (50~75)	Poor (50~0)
Large (750K~2000k)	74848	60060	22307	10932
Medium (300K~750K)				31340
Small (10K~300K)	77120	10000	10000	44329

Table 7.1. Cover image dataset.

7.1.2 Embedding Techniques Tested: We use a number of techniques to create our stego datasets:

1. *Outguess:* C source code for this technique is publicly available. The only problem of the code is that it only operates on color images, and crashes with grayscale images. Thus we had to modify the code in order to make the code usable with grayscale images.
2. *F5:* Java code for this technique is publicly available, and we have used it.
3. *Model Based Embedding:* There are two parts to this technique, in the first part the embedding operations is preformed, and in the second part of the technique a de-blocking algorithm is used to minimize the blockiness artifact introduced by the embedding operation. We have implemented the embedding part in C, but have been unable to implement the deblocking algorithm since much needed detail is missing on it in the relative paper. The main bottleneck of our implementations was in a minimum search function which we were able to speed up by using the Gnu Scientific Library. Also this embedding technique works by first obtaining the DCT coefficients from the JPEG image, modifying them, and then writing them back to the JPEG file. Thus we had to modify the publicly available JPEG library, in order to read and write DCT coefficients directly to the JPEG file.
4. *Perturbed Quantization:* We have implemented this code using C as well, although we have been unable to achieve embedding speeds suggested in the paper. We are currently at final stages of debugging the code.

7.1.3 Steganalysis Techniques Tested:

1. *BSM:* We are using a modified version of the BSM technique, were the only difference is in the neighborhood selection. We were only considering 4 neighbors (cross shaped neighborhood) in the old implementation but now we consider all 8 neighbors.
2. *Feature based technique (FBS)* This technique was proposed by Jessica Fridrich, in which all but a few of the statistical features used to distinguish between cover and stego images are obtained from DCT coefficients directly. The main challenge in implementing this technique was to modify the publicly available JPEG library, in order to obtain the DCT coefficients directly from the image. After obtaining the DCT coefficient from the JPEG files, we wrote a set of routines to calculate the features proposed by Jessica.
3. *Wavelet Based Steganalysis:* This technique was introduced by Farid, in which a set of statistical features are collected from the image after wavelet decomposition of the image. There were a number of challenges in implementing this technique; first of all we needed to find the appropriate wavelet transformation code in C which was flexible enough to accept the specific type of filter used by Farid. The second challenge was to implement parts of the algorithm which calculated the prediction error. The main challenge here was finding an optimized matrix manipulation library. Many of the freely available matrix manipulation libraries have restrictions on matrix dimension or are not fast enough for our purposes. For this reason we implemented a matrix manipulation library by borrowing code from well known libraries and writing some code ourselves. Since the image was decomposed in to several parts of varying sizes the data structures used to represent the image are fairly complicated. We wanted to provide fast and space efficient data structures that gave our application a high level of abstraction.

In terms of speed BSM seems to be the fastest, processing an image in about 500msec, feature based technique comes in with about 700msec per image, and the slowest technique is wavelet based steganalysis were each image takes roughly about 2 sec to process. These performance numbers were obtained by running each technique on 100 thousand images and averaging the processing time.

7.1.4 Experiments and Results: A separate classifier was designed for each 3-tuple of embedding technique, embedded message size, and steganalysis technique. Thus overall we designed and tested 87 classifiers. Classification accuracy for each 3 tuple was presented as the AUR (area under the ROC curve). These numbers are tabulated in table 1(a). From the results it is evident that with embedding technique which re-compress the image before the embedding process, the universal steganalysis techniques tend to confuse re-compressed images as stego

images to different extents. Thus we also compared stego vs. re-compressed images for cases where the image would be re-compressed before embedding process. The obtained results, in AUR, could be seen in table 1(b).

	Outguess	F5	Model Based	PQ	
.05	51.66	50.12	50.11	75.36	BSM
.05	52.50	51.76	50.14	76.61	WBS
.05	77.61	71.32	53.35	85.09	FBS
.1	54.06	50.56	50.85	75.50	BSM
.1	53.77	52.58	50.85	76.59	WBS
.1	89.05	77.12	57.06	85.55	FBS
.2	55.39	51.76	51.53	75.53	BSM
.2	58.16	54.97	53.41	75.92	WBS
.2	95.41	85.59	64.65	85.79	FBS
.4	NA	53.86	53.62	76.90	BSM
.4	NA	61.46	56.79	79.36	WBS
.4	NA	93.27	79.01	86.96	FBS
.6	NA	NA	56.40	NA	BSM
.6	NA	NA	61.61	NA	WBS
.6	NA	NA	87.29	NA	FBS

(a)

	Outguess	F5	PQ	
.05	51.61	49.94	51.23	BSM
.05	50.76	49.87	50.79	WBS
.05	65.10	55.20	50.27	FBS
.1	53.98	50.23	52.16	BSM
.1	53.27	50.58	51.90	WBS
.1	78.77	62.74	50.87	FBS
.2	55.82	51.25	53.33	BSM
.2	57.77	53.44	52.82	WBS
.2	90.91	76.39	52.64	FBS
.4	NA	52.55	55.34	BSM
.4	NA	59.94	55.54	WBS
.4	NA	89.93	56.95	FBS

(b)

Table 1.

It is evident from the results that FBS has the best performance among the 3 techniques studied. However, it should be noted that the embedding technique considered in this work operate in DCT domain, and mostly they tend to violate similar regularities in DCT domain (e.g., joint block-wise dependencies between 8_8 blocks, global and marginal histograms). Consequently techniques (such as FBS) that rely on DCT based statistical features are expected to perform better. We are currently expanding our work to other embedding techniques which operate on spatial as well as wavelet domain.

7.2 Detectability

Our work on the cover selection problem can be stated in the context of the well known prisoner's problem. Here Alice would like to send Bob a message, and she has access to a set of cover images. She would like to know that given a fixed message, and an embedding technique, which of the available covers would be least detectable by the warden Wendy's steganalyzer. (Of course we are assuming that all cover images have a similar number of changeable coefficients otherwise the image with the largest number of changeable coefficients would be the best cover).

At this point we could have a number of cases.

1. Alice has knowledge of the steganalyzer used by the warden Wendy.
2. Alice has some information on the steganalyzer being used. (i.e. steganalysis, is done in wavelet domain, or spatial domain ...)
3. Alice has no information on the steganalyzer being used.

Unlike the usual steganalysis approach, here Alice has both access to the cover and stego object and could measure directly the effects of the embedding process on the cover. Given the computational constraint which Alice has, she would like to have to herself a set of simple measures which could measure the distortion introduced by an embedding technique and with it choose the best cover object.

One might ask that in the first case, Alice has knowledge of the steganalysis technique being employed, then why not choose the cover image, with which the set of features calculated by steganalyzer are least modified. The problem here is that the feature set used by Wendy may be weighted differently and based on a training stage, and minimizing the error over all features with them being weighted uniformly could be useless. Thus Alice would require knowledge of how each feature is weighted in order to minimize the chance of detectability by Wendy. Such knowledge would require a training stage and access to a training dataset.

In this context we studied a set of distance measures which include, error in DCT domain, MSE in DCT domain weighted by the quantization steps, Information theoretic distance among the pdf's of cover and stego image calculated independently for each DCT mode, and MSE in wavelet domain.

Experiments were done with a dataset of 100K images in to two parts, each containing 50K images, which we will denote as set A and B respectively. The following steps are then executed for a 2-tuple of embedding and steganalysis technique to identify images which are consistently misclassified:

- For 100 iterations
    - Design a classifier with a random 20% of set A.
    - Classify images from set B, using the designed classifier.
    - Collect decision values for each image.

Afterwards, given an image, we count the number of time the image was misclassified. The normalized histogram of the obtained results could be seen in figure 7.1. One could observe three areas of interest in the figure. The first is the peak located at bin 0, indicates that a large number of images were always classified correctly as expected, since the accuracy of the classifiers are quite high. The second peak which is located at bin 100, indicated that there are a set of images, which are consistently misclassified. Lastly there are a set of images which fall in between these two peaks; these images are at times misclassified. We believe these images to fall close to the classifier's separating hyper plane, thus falling on either side by small variation to the hyper plane location which could be caused every time a new classifier is designed. The question arises that how well the misclassified images intersect over a set of embedding techniques? This is one of the questions we will investigate in the next phase.

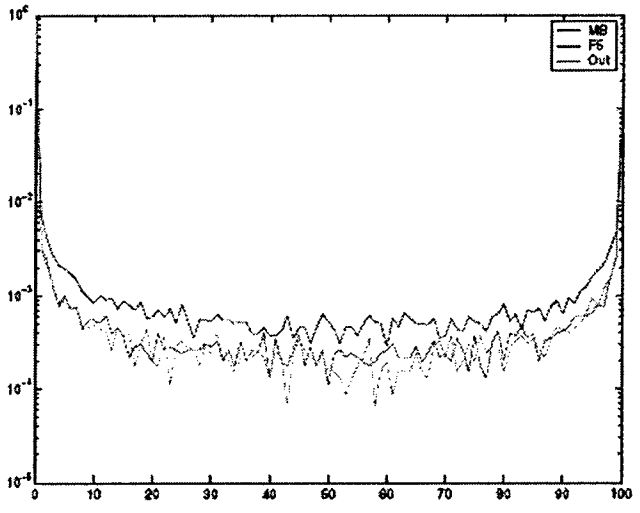

Figure 7.1.

Another question if we can predict if images would be misclassified as either of the two error types mentioned above, then with such information we could define a measure of goodness of cover. Where we define good covers to be images which when used in the embedding process, produce stego-image that are least detectable. Our initial results indicate that when using a specific steganalysis and embedding technique, we are able to rank how well an

image is distanced from the separating hyper-plane of the classifier. In other words, we could tell if an image is closer to the image types which are FN or FP.

7.3. Fusion

Although universal steganalysis techniques could obtain high accuracy results in distinguishing between stego and cover objects, they are not free of errors. Furthermore such techniques do not perform consistently with different embedding techniques as shown in our benchmarking study, but by fusion these techniques one could obtain superior performance and consistent performance could be expected over a range of embedding techniques. Such improvements could be understood by observing that errors made by different techniques do not overlap entirely. Fusion could be applied in a number of scenarios, for example among a set of classifiers all destined using one feature vector, but with different settings. (i.e. given a feature vector, we could design a linear as well as non-linear classifier and fuse the results together. On the other hand, fusion could also be applied to a set of classifiers each designed with a separate feature vector. Furthermore the two approaches could be combined into a hybrid approach. But more importantly is the stage at which fusion should be done. Jain et al. provide the following categorization on possible classification stages in at which fusion could be applied:

7.3.1 Pre-classification: Fusion at this stage could be done by concatenating the feature vectors obtained from each steganalysis techniques. Fusion at this stage would be best in theory, since the features are incorporated with out any processing and thus no information loss occurs. But in practice a number of problems arise with such approach. Firstly with large set of features one should be careful with the curse of dimensionality. Secondly correlated features need to be excluded. Third with such approach each time a new steganalysis technique is proposed we need to redesign the classifier. Furthermore different feature set could require different designing parameters. For example in our experiments we have observed that some feature vectors gain much improvement with more computationally expensive non-linear classifiers, where as other gain very little improvement. Thus is such cases one would be interested in assigning feature vectors specific parameters for designing the classifiers. We should note that fusion at such stage will not be applicable when different classifier are designed using the same feature vector but with different design parameters.

7.3.2 Post-classification:

1. *Abstract Level:* Fusion could also be applied at the other extreme and at the abstract level in which a decision as to the class of the object being tested has already made. In this case technique such as voting could be used to obtain a collective decision from a set of steganalyzers. This stage is not preferable since the final decision of the classifier is based on a threshold and that would provided minimal information for fusion.
2. *Measurement Level:* In between the two extremes we have measurement information obtained from the classifier, but the classifier has yet to make a decision as to the class membership of the object in question. For example such measurements could be conditional probabilities obtained for each possible class. This stage seems to be the favorite stage for fusion. Here the measurement info obtained from a set of steganalyzers could be either input into a second stage classifier for a final decision or could be combined using schemes such as the sum rule, or max rule which we will discuss briefly below. Given feature vectors $x_{1},x_{2},x_{3},...$ which represent pattern Z each classifier will provide us with conditional probability $P(c_{i}|x_{i})$, i.e. the posterior probability of pattern Z represented by feature x_{i} belonging to class c_{i}. Thus at this stage we could apply a set of rules in order to fuse the obtained conditional probability distributions from each classifier:

    a. Sum Rule: $C=arg max_{j} \sum_{i=1}^{R} P(c_{j}|x_{i})$
    With this rule the class assigned to input pattern is the class with which the Sum of the conditional probabilities for that class are maximized.

    b. Max Rule: $C=arg max_{j} max_{i} P(c_{j}|x_{i})$ Here the class is assigned to input pattern with which the maximum conditional probability is obtained.

7.3.3 Preliminary Results: To show the effectiveness of fusion in steganalysis, we have carried out a set of experiment which we will go over in this section. An initial database consisting of 1800 natural images were used. The images were converted to gray-scale and the borders around them were cropped, resulting in images of size

640x480 pixels, after which they were re-compressed with a quality factor of 75. A stego dataset was created using the LSB and LSB +/- embedding. Message size was set as the percentage of bit per pixel value, more specifically we used to message sizes of 10% (3840 Bytes) and 20% (7680 Bytes) in creating the stego set. Three universal steganalysis techniques were employed and a classifier was built for each message length using the feature vectors obtained by each steganalysis technique.

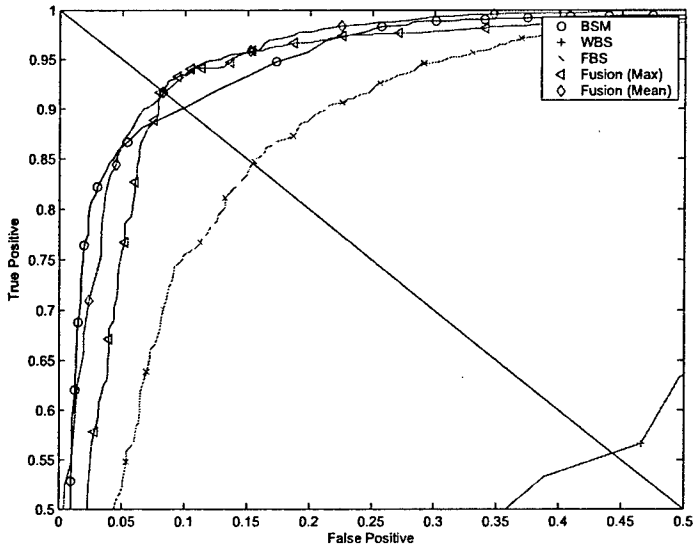

Figure 7.2: ROC curves obtained for LSB vs. Unmarked images. Here a 10% messages size was used.

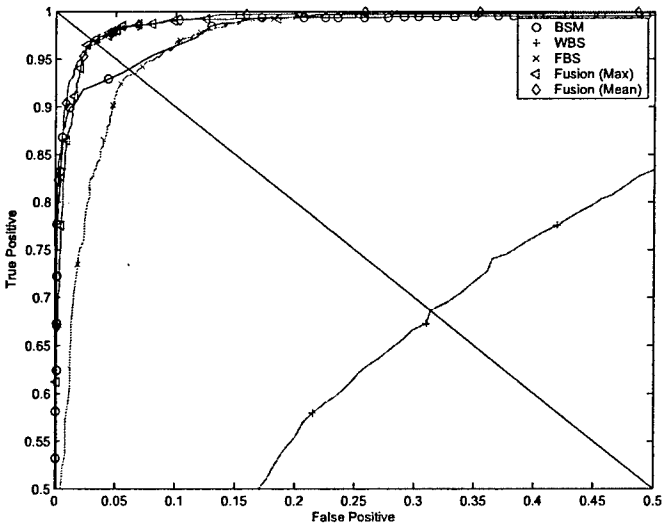

Figure 7.3: ROC curves obtained for LSB vs. Unmarked images. Here a 20% message size was used.

Using the class conditional distribution from each of the 3 steganalyzers two fusion schemes at measurement level, max rule and average rule, were used. The obtained ROC curves could be seen in the figures for the LSB technique. As seen in these Figures, fusion does provide considerable improvement over the 3 steganalysis technique.

7.4 Steganalysis of plus-minus one embedding

In thie project, we have also considered the steganalysis of plus-minus one embedding. Specifically, we proposed a method for steganalysis using a state transition of the embedding process, plus a wavelet denoising approach to estimate the original cover image. State transition of the embedding procedure is defined using neighboring pixel values. Each transition path is specified by a probability related to the embedding rate. To estimate the embedding rate, it is necessary to estimate the original image. To this end, we proposed to use a wavelet denoising method. Using these components, we can relate the information between the measured state and the estimated original state, from which the embedding rate is estimated. Experimental results using JPEG compressed images is very encouraging. We have been considering extension of the method to support ohter types of images, e.g. images that have never been JPEG compressed.

Publications

M. Kharrazi, H. T. Sencar, N. Memon, "Benchmarking steganographic and steganalysis techniques," SPIE 2005.

P.W. Wong, H. Chen and Z.J. Tang, "On steganalysis of plus-minus one embedding of continuous tone images," SPIE 2005.

REPORT DOCUMENTATION PAGE

AFRL-SR-AR-TR-05-

0272

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		FINAL REPORT

4. TITLE AND SUBTITLE
AUTOMATED DETECTION OF STEGANOGRAPHIC CONTENT

5. FUNDING NUMBERS
FA9550-04-C-0110

6. AUTHOR(S)
Dr. Ping Wah Wong

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Office of Scientific Research
875 North Randolph Street
Suite 325, Room 3112
Arlington, VA 22203
Dr. Robert Herklotz

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DCMA Northern California
P.O. Box 232
700 E Roth Road, Bldg, 330 (Lathro
French Camp CA 95231-0232 NM

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release;
distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

In this document, we will describe our Anti-Steganography project for Phase 1. First, we will present the Software Architecture in Section 1. In Section 2, we will describe in details the Application Programming Interfaces (APIs) and the Software Implementation. In Section 3, we will show the organization of the source code. Section 4 discusses how to maintain and extend the software. Section 5 discusses the automatic engine update service. A description of the Phase I demo is given in Section 6. Finally, we describe the steganography research activities and results in Section 7.

14. SUBJECT TERMS			15. NUMBER OF PAGES
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

7-13-05