A Framework for Media Fingerprinting

Master's Thesis

Douglas J. Hickok

University of Wisconsin – Platteville

Master's Thesis Approval
Signature Page

A Framework for Media Fingerprinting

Submitted to

Department of Computer Science and Software Engineering
University of Wisconsin - Platteville

Signatures of Committee Members:

Major advisor: Michael C. Rowe date: 6-Nov-2008

(Print name) MICHAEL C. ROWE

Committee Member: Robert W. Hasker date: 6-Nov-2008

(Print name) Robert W. Hasker

Committee Member: [signature] date: 6-NOV-2008

(Print name) Bruce Litow (JCU)

A Framework for Media Fingerprinting

A Thesis

Presented to

The Graduate Faculty

University of Wisconsin – Platteville

In Partial Fulfillment

Of the Requirement for the Degree

Master of Science in Computer Science

By

Douglas J. Hickok

2008

Acknowledgments

I would like to thank a number of people who have supported me in this thesis project.

Family and friends including John & Brigitte Hickok, Phil & Inez, Helma & Jerry, and the rest of my family, Amanda Springborn and her family. Daine Lesniak, John Trunek, Kris Whisler, Bill Pyne, Pat McCarthy, Jon Schendt, Andy Schaumberg, and everyone else who has participated in some form or another.

University of Wisconsin – Platteville faculty, past and present, including Dr. Mike Rowe, Dr. Rob Hasker, Mohan Gill, Dr. Joe Clifton, Tom Scanlan, Dr. Qi Yang. Inspiration for this thesis topic was from Dr. Jim Gast.

James Cook University, Townsville, Australia faculty including Dr. Bruce Litow, Dr. Ian Atkinson, Dr. Hossein Ghodosi, Dr. Ling Qiu , and the rest of the staff that made my studies abroad enjoyable.

The MICS crew, who have accepted and published all of my work thus far. Thanks for the "best graduate paper" award, confirming that I was on the right track.

The UWP ACM members who invited me to speak on the subject, and enjoyed all the technical details.

Esterline AVISTA, for giving me the extra time to work on this.

Abstract

Anonymous Internet piracy and copyright infringement is a concern for most digital media producers. Digital media is an easy target for piracy, since making and distributing copies is instantaneous, free, and anonymous. This results in lost profits for both large companies and independent producers. Current solutions are inadequate, expensive, or target the wrong people.

This paper describes a new and improved solution to current methods. This solution is a web framework for embedding a traceable fingerprint, via a watermark, to all purchased or downloaded media. Using this framework, a media producer can find who originally downloaded the media file from their site. This ability is handy when copies of a file turn up in unauthorized places.

The framework has three key components: a watermark algorithm, a fingerprint protocol, and a plugin architecture.

Material Published From This Thesis

Hickok, D., and Rowe, M. C., 'Fighting Piracy: A Framework for Media Fingerprinting', *40th Annual Midwest Instruction and Computing Symposium*, April 2007, Grand Forks, ND. Awarded “Best Graduate Paper”.

Table of Contents

1 Statement of the Problem

The digital age has made media sales and distribution quick, easy, and autonomous. Unfortunately, it has also made media piracy quick, easy, and autonomous as well. A powerful weapon against piracy is digital watermarking, since it embeds data into the media itself. There are a wide variety of algorithms and methods, but relatively few useful implementations, especially for robust invisible watermarks. Additionally, the state-of-the-art in digital watermarking has recently swayed towards digital rights management (DRM) and proving genuine copies of the media. While it is great that the owner of a specific media file can prove that they own it, the anonymity of the Internet can mask the identity of the attacker. Thus, this thesis proposes a solution that can identify the original copyright infringer and remain strong enough to hold up in court.

The end product will finally give media producers such as photographers, video producers, artists, and musicians (depending on available watermarking plugins) a weapon to fight back against piracy. Using the system, a media producer will be able to prove the stolen content is theirs, and more importantly, trace exactly which customer is responsible for illegally distributing it. Additionally, the system should be easy to use, keep the media quality intact, be able to integrate into their existing system, keep reasonable server performance, and have the ability to keep up with new and improved algorithms.

The idea for the Media Marking Framework was inspired by the lack of current technology for practical protection of copyrights. A survey of the field shows that most watermarking systems are proprietary and expensive [1], single-format [2], or not very useful [3]. There are many excellent watermarking algorithms and ideas in academia, but not many have made it to the real world. This project aims to bridge the gap.

The goal of this thesis was to research, design, and implement a modular server-side framework for fingerprinting, via an invisible watermark, all distributed media. Then, evaluate performance and conclude if the system would be worth using by media distributors. "Media distributors" is defined as people who own websites, and either sell or allow free downloads of their media to non-anonymous authenticated users. The overall question is: Why are there no practical media protection implementations for small media distributors? This thesis presents one, proving that the technology is available and that the idea is feasible.

2 Watermarking

The digital watermarking system consists of an algorithm for embedding and extracting data from a media file. Generally, it does this by changing the file in such a way that it is digitally different, but perceptually the same. There are a wide variety of watermarking algorithms available [4], and each has its own specific purposes and strengths.

There are many classifications of watermark algorithms, and the basic ones covered in [5] will also be summarized here in order to gain perspective. The most basic classification is *visible* vs. *invisible* watermarking. Visible watermarks, which are popular on many Internet sites, consist of a logo or copyright text being stamped onto the media file. As the name implies, they can be seen by humans. Invisible watermarks are embedded into the media file in such a way so that they cannot be seen by humans. The advantage to invisible watermarks is that they cannot be cropped off easily, and do not disrupt the perceived quality of the media.

Another classification is *fragile* vs. *robust* watermarks. Fragile watermarks are designed to be destroyed if any sort of file modification is done. These are used to prove that any given file is authentic and in its original form. It does this by embedding data in the least significant bits of the media file, which are generally the first to be destroyed with even the slightest modification. (This is also the technique that Steganography uses.) Robust watermarks work in the opposite way -- to survive as much file modification as possible. Instead of embedding data in the least significant bits, robust watermarks embed into the most significant bits that still allow them to remain imperceivable. This method survives most attacks, unless the attack also perceptually reduces the quality of the file. Destroy the watermark, and the file quality is also destroyed with it. Robust watermarks are best to ensure that the embedded data survives.

The last classification to be covered here is the difference between *blind*, *semi-blind*, and *non-blind* detection of the watermark. Blind watermarks can be recovered by applying the algorithm directly to the distributed media itself, which is useful for applications such as media players to check if the media is copyrighted. In [6] they apply this method in panoramic creation software (to detect unregistered versions being used for commercial purposes) since the original file will never be available. Semi-blind watermarks require the original published (watermarked) version of the file, which would be handy for media creators who license their works to be distributed elsewhere, and the distributors are responsible for enforcing copyrights but never have the original. Lastly, non-blind watermarks require the original non-watermarked media file, which limits watermark extraction to only the owner(s) of the original file. For semi-blind and non-blind watermark extraction, a file found "*in-the-wild*" can be resynchronized

to match the size, shape, brightness, contrast, and other basic attributes of the original before being tested. [7]

Since there is a wide variety of watermark algorithms and many different classifications that are good for different tasks, choosing a suitable algorithm depends on the requirements of the system.

2.1 Requirements

1. The watermarking system shall be easily upgradeable.
2. The watermarking system shall have a way to recover the watermark from media.
3. The watermark shall defend against unauthorized removal or tampering.

Watermark algorithms are constantly changing and evolving with new research, so requirement 1 helps keep the system up-to-date with the latest in watermarking technology. Requirement 2 specifies that any possible watermark algorithm that embeds a watermark must also be able to recover the watermark, assuming it still exists in the media. For this project, it is unacceptable to have to manually recover the watermark, since it breaks the goal of being easy to use.

Requirement 3 is important since it is how the media is protected, and the watermark classifications are helpful in choosing the type of algorithm that will be suitable for the protection. Since the implemented watermarker is for image files, the classifications will be discussed as applicable to image files for the rest of the section. Visible watermarks on images can be removed by cropping, or by blurring or covering the watermark. Invisible watermarks are spread throughout the entire area of the image and don't immediately alert the attacker to the presence of a watermark. A robust watermark, as opposed to a fragile watermark, is clearly needed to survive image attacks. Lastly, since the project targets media producers, it is most applicable to make the algorithm non-blind. This limits the watermark extraction process to only those with the original image.

Another important decision to make is where to hide the watermark data. This depends on what type of image format is being dealt with, and if it is lossless or lossy. Lossless image formats retain all of the original image information, and lossy image formats cut out the least noticeable data from the image to favor better compression. Hiding in the spatial domain, the individual color values of each pixel, is easiest and will be preserved in the lossless BMP and PNG image formats. However, JPEG is one of the most common image formats used on the Internet today. [8] JPEG is a very lossy format, and any data saved in the spatial domain will likely be destroyed by JPEG compression.

To start the JPEG compression process, the image is converted from the Red/Green/Blue (RGB) color space to the Luminance/Chrominance (YCbCr) color space so that the two chrominance components can be downsampled. [9] Downsampling is a process which is equivalent to resizing the image to 50% of it's original size. Since the luminance contains most of the detail, only the chrominance components are downsampled. Each component is split into 8 pixel by 8 pixel blocks and prepared for the Discrete Cosine Transform (DCT). The DCT takes the spatial domain data and converts it into the frequency domain. After the DCT, the data is quantized. In quantization each number is divided by a constant so that only the important perceptual data remains. After this, the remaining data is compressed with Entropy coding and wrapped in the JPEG Interchange File Format.

Although the JPEG encoding process is complex, the two parts that are applicable to this watermark algorithm implementation are the DCT and quantization. The frequency domain (calculated by the DCT) is where the watermark will be embedded, and the quantization constants dictate how much the values will need to be changed.

2.2 Implementation

Although there are several other forms of transformations, [10] and [11] for example, the one used in this implementation is the same as what is used in JPEG, the DCT. Any would work fine, but the DCT is perhaps the simplest and best understood.

The watermarking algorithm implementation logically consists of a watermark_block function and a block_chooser function. The block_chooser function acts as the main function for the watermarker, and makes calls to the watermark_block function to embed the data. Finally, the whole watermarker implementation is wrapped as a plugin for the Media Marking Framework, as detailed in section 4.

The first step in the watermark_block function is to convert the 8 pixel by 8 pixel image block from RGB to YCbCr, exactly like the JPEG compression process noted above. In order to make a watermark algorithm as robust as possible, the worst case scenario should be assumed. Since the chrominance color components may be downsampled during JPEG compression, it seems logical to downsample them for the embedding process. However, this will result in an overall quality loss of the watermarked media. More specifically, 50% of the color information would be lost by downsampling the chrominance components. If watermark data is embedded without downsampling, values in the higher frequencies will likely be lost. For this implementation, the watermark is inserted into the middle

frequencies making them less likely to be lost in downsampling. Therefore, there is no need to downsample the chrominance color spaces in order to do the watermark embedding.

At this point there are three image blocks, each of which have pixel values from 0 to 255 representing the Y, Cb, and Cr components. Each pixel value of each block is reduced by 128 to prepare it for the Discrete Cosine Transform, resulting in a value range from -128 to 127 that is centered around zero.

After the DCT, the image block values can range from about -2047 to 2047. Different ways of calculating the DCT may produce slightly different numbers. The resulting block represents the frequency domain of the original block. The value in the top left corner is the DC component, and the rest of the values are AC components. The DC represents the average intensity of the entire block. The AC components near the top left are the low frequencies found in the block. The AC values closer to the bottom right are the high frequencies. The ideal range for making changes is in the middle to lower frequencies. Changes in the very low frequencies will result in a blocky look, and changes to the mid-high and high frequencies will result in a noisy look and are more likely to be chopped off by JPEG compression.

Since the middle to lower frequencies are the least likely to be noticed, how should they be changed? Since the algorithm will have access to the original image, it can compare the value in the watermarked image to the corresponding value in the original image. Therefore, to embed a 1 we would make watermarked[x, y] > original[x, y], and to embed a 0 make watermarked[x, y] < original[x, y]. While looping through the watermarkable values, a constant is added to the value to indicate a 1, and the same constant is subtracted from the value to indicate a 0, depending on the next bit that needs to be embedded.

In order to find the constants, the worst case scenario needs to be considered. The next step in JPEG compression, and the primary cause of its lossy nature, is the quantization. This is where each value is divided by a constant. After some searching, the most destructive commonly used quantization table available seems to be the table used in Photoshop CS2 00-quality JPEG encoding. [12] Using that as a baseline, the constants used for the watermarking are shown in table 1.

0	0	51	81	66	40	51	0	0	0	52	47	99	99	99	0
0	36	48	47	28	58	0	0	0	38	26	66	99	99	0	0
51	48	47	28	40	0	0	0	52	24	56	99	99	0	0	0
81	47	28	29	0	0	0	0	47	66	99	99	0	0	0	0
66	28	37	0	0	0	0	0	99	99	99	0	0	0	0	0
39	35	0	0	0	0	0	0	99	99	0	0	0	0	0	0
49	0	0	0	0	0	0	0	99	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 1: Luminance (left) and chrominance (right) constant tables for watermarking in the frequency domain.

After changing the values in the frequency domain, the inverse DCT is calculated, 128 is added back to each value, and the Y, Cb, and Cr image blocks are converted back to a single RGB image block. The result is a watermarked 8 pixel by 8 pixel block that should appear to humans to be the same as the original. However, this was not the case during the first round of testing.

After some experimentation, artifacts from the watermarking process were clearly visible in non-textured areas, and not at all visible in textured areas. The conclusion that was reached is that if a value will normally end up as zero after quantization, don't change it. If it is changed, texture is added to a normally non-textured area and the human eye will quickly notice it. Therefore, this change was implemented: If any watermarkable value is less than the quantization constant, skip it. During watermark recovery, we do this same test on the original copy of the image to know if the value was skipped or not. After the change, the resulting watermarked images were perceptually identical to the originals.

Figure 1: Original Lena on the left, and watermarked Lena on the right with 1117 bytes embedded.

The block_chooser function pseudo-randomly picks unwatermarked blocks and embeds the watermark in each until it reaches its maximum number of blocks. The number of blocks is defined as a constant in the system configuration, and should be tuned based on the length of time it should spend watermarking. This way, if the same algorithm is executed on an already-watermarked file, it will have less of an impact on the original watermark and both watermarks should be recoverable.

To initialize the block_chooser function, the pseudo-random number generator needs to be seeded with some sort of private key. The item that is meant to stay private makes an ideal choice -- the original media file. Since the entire contents of the file is too large for the number generator, a 32-bit CRC is performed on the data and the result is used as the seed.

For recovering a watermark, the same process is repeated except that the values are checked instead of changed. At the end, the algorithm computes the mode for each bit in the watermark, and displays the watermark with the percentages of how often each bit occurred. For example, if the recovery shows a 1 but that bit was a 1 only 52% of the time, it is unlikely to be correct. If all the bits have percentages closer to 70%, they are likely correct and show that the image has undergone some processing.

Remember that this implementation is just a single plugin for the system. Plugins can be developed using any sort of watermarking technology and work with any sort of file format. Now that there is a reasonably robust image watermarker implementation, the next step is to find some useful data to insert as the

watermark.

3 Fingerprinting

The *fingerprint* is a specialized form of watermark, which acts as a unique identifier for each copy of the media. A typical watermark can identify who owns a specific media file, normally by the use of an embedded logo or copyright text. A fingerprint goes one step further and can identify exactly which copy of the media file it is, normally by the use of a unique ID number. A watermark may be useful to prove who owns the media file, but an anonymously posted file renders the watermark somewhat useless. The watermark proves ownership, but doesn't prove who is guilty for illegal redistribution. The media industry is realizing this, and many are adding some sort of fingerprint technology to their media distribution systems. [13]

Imagine this scenario: Alice owns a specific media file, and sells a copy to Bob. Bob likes the media and decides to post it anonymously to several sites, and shares it over some peer-2-peer networks. In the end, a lot of people get a free copy of this media instead of purchasing it from Alice, and Alice looses a lot of money. There are three different endings to this story. First is that there is no watermark in the media. Alice has no way to figure out who redistributed it and is unable to prove that she owns it. Second is that she embedded the media with a watermark containing her name. In this case, Alice is able to send cease-and-desist letters to all of the websites hosting it, because she can prove that she owns it. She is unable to get compensation from the websites because most are not responsible for the content that gets posted. Also, it is unlikely that she will ever be able to find out who originally posted the file. The third ending to the scenario is that she fingerprinted the media with her name and Bob's name. This time she knows who to take to court in order to get compensation, and can prove which publicly available copies are Bob's fault.

For this project, a strong fingerprint is needed for the watermarker to embed. Unforeseen privacy problems could arise if we simply use a name or user ID for the fingerprint. Also, a serial number for each media file lacks any sort of tamper protection for both the buyer and the seller. The watermarker does its best to keep the fingerprint intact. However, there is always the possibility that the buyer can change the serial number, or that the seller can change the log file holding the serial number, both in order to frame someone else. These possible attacks were considered in the fingerprint requirements.

3.1 Requirements

1. Fingerprints shall identify the original downloaders.
2. Fingerprints shall maintain downloader privacy.
3. Fingerprints shall be difficult to duplicate or forge.

The first requirement specifies that there must be some sort of traceable link between the fingerprint data and the user who is downloading the media. This link may not be able to point out the exact person, but can at least point out certain information such as the username they are using. It is up to the webmaster to keep accurate details of the users.

The second requirement keeps private data about the downloader out of the media files. If a name is inserted, it could later be recovered from in-the-wild media files and be considered invasion of privacy. Bob's music downloads are a private exchange between Bob and Alice. For example, if an in-the-wild music file attributes Bob's name to an anti-government song, and Bob works a government job, Alice could be held responsible for Bob getting fired. A simple solution to satisfy this requirement would be to store a hash of the private information, instead of the private information itself.

The last requirement is a best-effort resistance to fingerprint forgery. The easier it is to tamper with the fingerprint, both on the downloader and webmaster sides, the less authentic it becomes. If the downloader knows anything about the webmaster's databases, then they could change the fingerprint to point to someone else. If the webmaster knows everything about the fingerprints, then they could also change the database or log file to point to someone else, or regenerate the media file that was distributed. Even if none of this happens, either side could argue that it is possible and that the fingerprint is not authentic.

There are numerous papers on good buyer-seller watermarking protocols, [14] for example. They protect the anonymity of the buyers and offer a great protocol using public key cryptography. However, the anonymity is protected via a third party, and the cryptography requires special software if a third party is not used. The end effect is that third parties cost money, and special software requirements limit the buyers who are able to participate in the exchange, which defeats the end goal of being free and simple.

3.2 Implementation

The ideas behind the Diffie-Hellman [15] exchange were adapted for the implementation. The buyer and seller each have a private number for the session. The commonly known information between them is the hash of buyer's private

number, and the hash of the seller's private number. That way, neither side's private number can be guessed by the other.

Although this sort of exchange is supposed to be done independently by both sides, it is unlikely that the buyer will insert their own private number hash into the media after it is downloaded. The only way to ensure that the buyer's private number is inserted (without the use of a third party or custom software) would be to embed it on the seller's server. Since the server needs to know everything about the exchange anyway, it might as well do the exchange internally before sending out the fingerprinted media file. Although it may appear that all security is lost in this process, a chain of verifiable hashes keeps the webmaster from tampering with any of the code used to generate the media and the log files. Any change to either the code or the log entry would put the seller at a disadvantage, because then hashes would not match up and the verifiable chain would be broken.

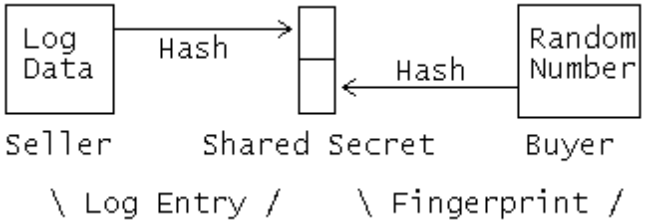

Figure 2: Similarity to Diffie-Hellman exchange, with both parties having a shared secret and each having a private secret. The line on the bottom shows the data that gets inserted into the server log and into the media as a fingerprint.

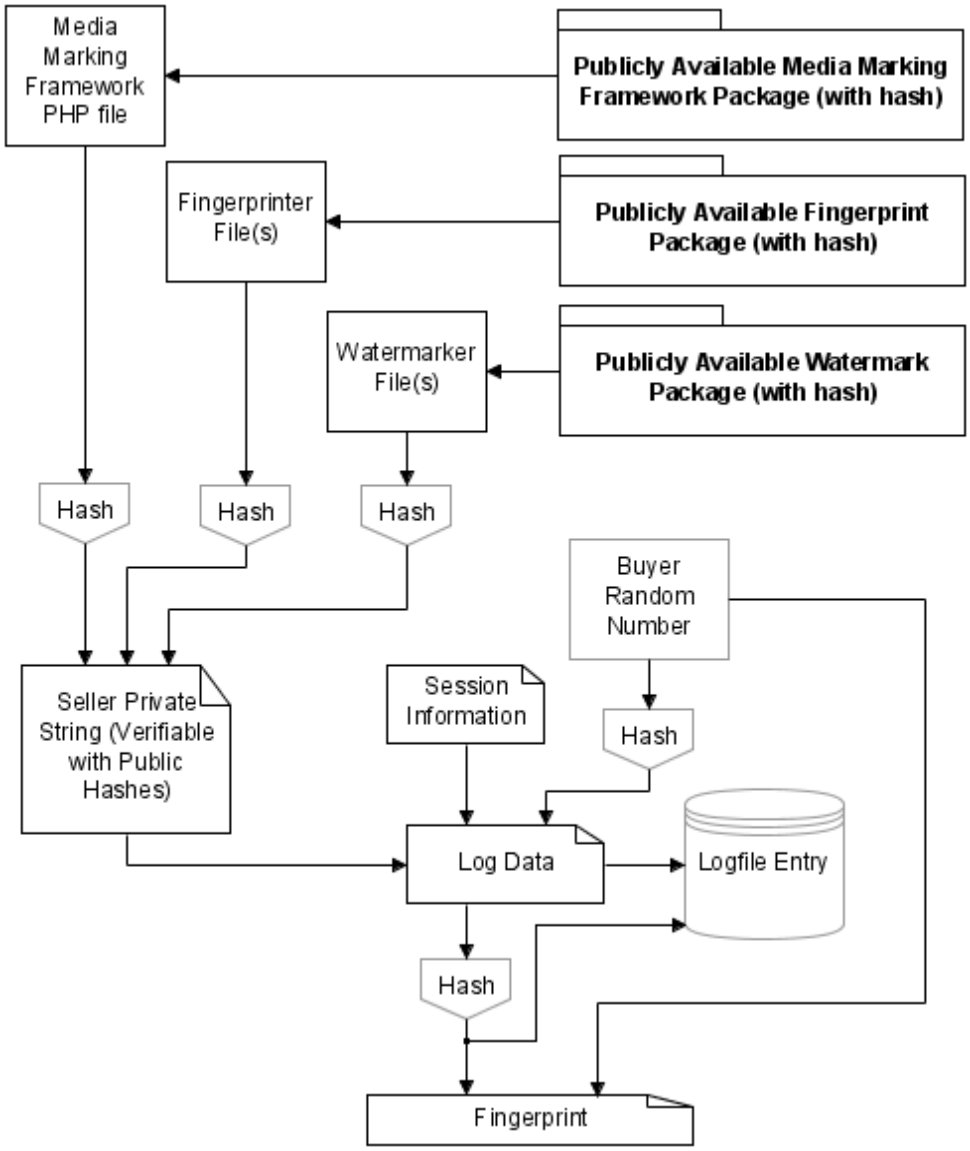

Figure 3: The flow of fingerprint generation. Note that the hashes are one-way algorithms (MD5) that prevent information from flowing backwards.

As shown in Figure 3, the Media Marking Framework, Fingerprinter, and Watermarker are publicly available, either open source or commercial, and are distributed with an MD5 hash. [16] The MD5 hash is used because it's a one-way algorithm. The hash of some block of data is always the same, but any tiny change to the data will result in a completely different hash result. Also, there is no way to figure out the original data from the hash. During execution, the Framework calculates the hash of all critical code files required to process the transaction, and stores it as the Seller Private String. The string includes the calculated MD5 hashes of filenames, and version information of the files.. This

string is included in the Log Data, and can later be used to verify that the code running on the server is the same as the code in the official distributions. Another item in the Log Data is the hash of the random 16-byte number that was generated for the buyer. The original number forms half of the final fingerprint, and acts as a reverse check and prevents the webmaster from regenerating the file. The rest of the Log Data consists of session information such as the date/time, username, file requested, and IP address. Finally, the Log Data gets stamped into the log file with a hash that is generated to complete the fingerprint. The total fingerprint size is 32 bytes, which is then embedded into the media file.

Tracing back the fingerprint from an in-the-wild media file is fairly simple by reversing the fingerprinting process. The hash contained in the first part of the fingerprint is also in the log file, so a search of the log file will result in the log entry. Additionally, the random number in the second part of the fingerprint can be hashed, and should match the one listed in the log entry. Various editing of the media will most likely result in some loss of watermark information. If enough watermark information is lost so that a few bits are incorrect in the recovered fingerprint, it can still be matched with some degree of certainty to the log entry. However, the hash of the random number will fail to match with what is in the log, unless the unknown bits are found via brute-force. Since the hash size and number size are the same, there is a 1-to-1 mapping between the two making the brute-force method acceptable to fill in the gaps.

Just like the watermarking system, the fingerprinting system is plugin oriented. This allows new forms of fingerprints to be created that can keep up with current technology. If a webmaster would like to use the secure and anonymizing services of a third party, a fingerprint plugin could be created and distributed by the third party. The framework allows quite a bit of flexibility for new fingerprinting plugins.

4 The Plugin Architecture

The Media Marking Framework Plugin Architecture acts as the ringmaster for all the plugins, log files/databases, downloaders, and webmasters, and can easily be set up on most common web servers. It also provides a level of security by acting as a firewall between the original media and the rest of the world. It assumes that the webmaster has a configured and running website, and does its best to be non-intrusive and easy to integrate into whatever may already be there. The webmaster has a simple interface for checking on the system and testing files found in-the-wild, and the downloaders have a completely transparent interface which only sends them a file.

For this portion of the project, design is very important. The system needs high

cohesion and low coupling to keep it small, elegant, and easy to maintain. However, a few small portions of code need to be optimized for speed in order to reduce the download lag time. To understand the system, it is best to review the requirements and discover how they were fulfilled.

4.1 Requirements and Implementation

1. All downloaded media shall be properly fingerprinted.
2. The system shall embed a fingerprint into media by using a watermark.
3. The system shall run on most common web servers.
4. The system shall have the ability to handle various file formats.
5. The system shall integrate with existing web services.
6. The system shall be efficient, scalable, and flexible.

The first requirement states that only fingerprinted media should leave the server. Under no conditions should the server be tricked into sending out something that is either not watermarked or not logged. The chain of events as implemented will only log the transaction if a fingerprint was successfully generated, and will only begin sending out the watermarked media if the transaction was successfully logged. An error along the way will cause the file to not be sent, and even a partially sent file remains logged.

Requirements 2 and 3 cover a lot of ground in the project. Requirement 2 is thoroughly covered by watermarking and fingerprinting sections of this paper. Requirement 3 constrains what programming language the project can use. [17] shows that the most common web servers are Apache (either on Linux or Windows) and Microsoft-IIS related, and both have been the top two since 1998. Both have support for PHP, other scripting languages, and executable binaries. Of these choices, PHP doesn't need permissions to be set in order to execute and doesn't need to be compiled, which make it handy for websites hosted by another company. PHP has an enormous feature set that is geared specifically for web applications [18], making it a great choice for this project.

Requirement 4 keeps the system open to any possible file format, and any possible file size. When working with pictures, it is easy to read in the whole file, do processing, then send it to the script output. What happens when the pictures get too big, or when the webmaster decides to switch to audio or video? The system must be designed to handle large files. The framework is implemented in such a way to chain together watermarking plugins in a similar manner to the Unix pipe operator. The chain starts with the media file output stream, which goes to the plugin1 input stream, whose output goes to the plugin2 input stream, and so on. The output of the last plugin in the chain gets sent to the downloader in reply to the HTTP request. Even though PHP itself is unable to do multi-threading, this

setup can become multi-threaded with plugins that wrap an external program, such as watermarking algorithms implemented in another language. The framework loops through the plugin chain until all data is through.

Integrating into existing systems, as per requirement 5, has to be as flexible as possible. The framework is designed to be initiated by a PHP function call, with parameters that tell it what to do. By sending it a web request, it simply returns a file. In an existing PHP-based website, the function call can be made from anywhere. In other systems, a simple wrapper can be used. Instead of a link to "picture.jpg", a link can be made to "get_file.php?file=picture.jpg" which will appear to the downloader to be the same. The wrapper will make the call into the framework and tell it how to watermark the file.

Requirement 6 defines the overall goals for the system. Although PHP is a scripting language, and therefore far less efficient than a compiled language, the framework does its best to minimize transaction processing time. Most parts of the framework are only loaded on demand. If the logger is not needed, it is never loaded and initialized. All of the plugins are loaded on demand. This minimizes the file reads for PHP, and processing time to initialize each of those parts. Most importantly, the majority of the processing is not in the framework at all – it is in the watermarking plugins.

For better performance, it is possible to implement the watermarking algorithm in a faster language such as C or C++. This executable can be called from PHP with any needed command-line arguments, and all input and output can be hooked to PHP IO streams. This process runs independently from the PHP script, so it can do the number crunching while PHP worries about taking care of the input and output. The only drawback is that the executable will need to be compiled for each different system it runs on, and permissions will need to be set to allow it to execute on the server.

The scalability in requirement 6 is also considered in the design. This is satisfied by keeping no session information and supporting the option of logging to a database. Since the framework can process the requests independently and on-demand, no session information or temporary files need to be kept. The option of logging to a database prevents multiple instances of the framework from fighting over write access to a log file. These two considerations make it possible to run the system on a load balancing cluster or distributed system setup.

The flexibility in requirement 6 is satisfied by generalizing the design as much as possible, the plugin architecture, and the inherent flexibility of the PHP language. Generalizing the design meant not making assumptions about the system, such as that it will only work with images or small files. The logger can work with a file or with a variety of databases (via the PEAR DB functions). [19] There is a great

deal of flexibility in making the function call into the framework, since that call controls what the framework needs to do. The framework for the plugins allows the plugins to do just about anything they need to, as long as they follow a few simple guidelines to work with the framework. Because of the plugin flexibility and the flexibility of the PHP language, a few interesting side benefits have cropped up.

In the configuration, the webmaster needs to specify the path to the original media files. The first side benefit, thanks to the flexibility of the PHP language, is that this path does not need to be a local directory. Instead, it can be either a HTTP or FTP URL with support for user authentication. There is no need to store all the original media in an unsecure place on the web server, instead a URL such as "https://user:password@securelocalserver/media" can be used. The secure local server can be blocked from any other Internet access except for "user" on the IP of the framework. If this is not enough, PHP allows custom stream definitions. Instead of opening a local file or web URL, a stream could be created that pulls the media from a database or from a cryptographically secure place.

The plugins allow for a large variety of side benefits. Although the watermarking plugins were designed for watermarking, they happen to also make great processing plugins. They could be written for thumbnail generation, audio/video preview generation, format conversions, compression, package creation, statistics gathering, 3rd party utilities and services, or just about anything else. In the example system use case, several of these ideas are shown and put to good use.

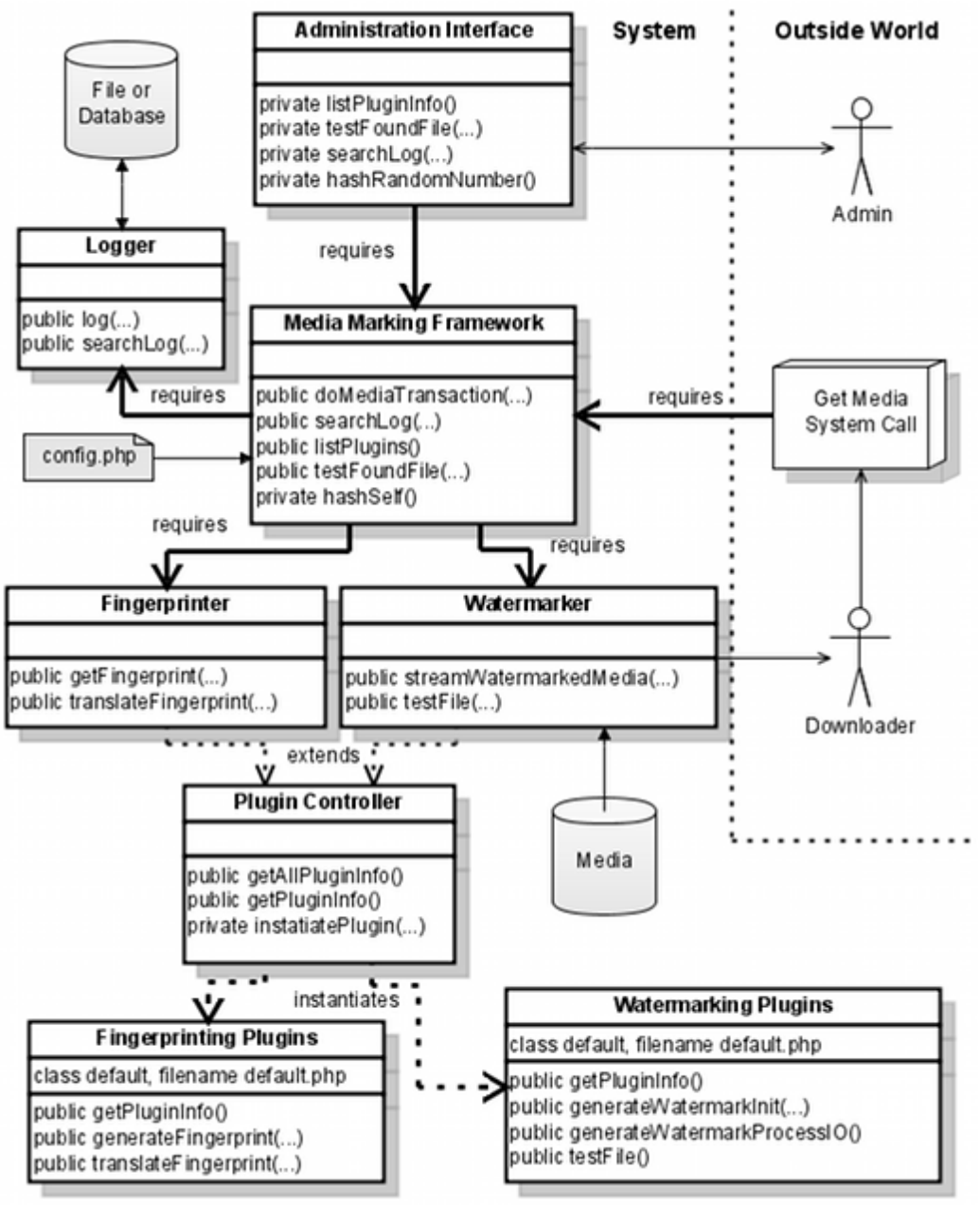

Figure 4: Resulting structure of the Media Marking Framework, including its interaction with the outside world.

4.2 System Use Case

Alice sells her landscape photos on her hosted website. Thumbnails of the photos, all generated manually by her, are currently shown using a PHP photo album, and orders are taken and filled manually. Alice wants an automated process that can

distribute digital copies, but secure enough so that the images don't get stolen.

She decides to install the Media Marking Framework on her server, and does so by copying all the files from the distribution to her server. Following the install guide, she sets up a directory for the media files and changes the Apache ".htaccess" files to limit the "get_file.php" to registered users, and the administration interface and media directory to be only accessible by her. After setting a few options in the configuration file, such as the media directory to be "http://alice:pass@localhost/media", she can add a few plugins to the system besides the default fingerprinter and default watermarker. She decides to add a visible watermarker, JPEG compressor, zip creator, image resizer, and a plugin that interacts with PayPal to bill users when the download finishes, all of which are dropped into the plugins directory in the framework.

With the help of the plugin documents, she modifies the get_file.php to bill via PayPal, watermark the media, then zip it with a copyright agreement. She also creates a similar file, get_thumbnail.php, to automatically resize the image to be small, add copyright text as a visible watermark to the bottom, and JPEG compress it at low quality. Next, with the help of a web forum, she changes her photo album program to get images from "get_thumbnail.php?file=$filename" instead of just "$filename". Also, a link to purchase the file is added, and points to "get_file.php?file=$filename". At this point, her new system is set up and ready to test and use.

Bob has registered with Alice's site, and decides to purchase a photo. (The album he sees has low quality small-sized images that have Alice's name stamped on them.) He clicks the purchase link for the photo he wants, and is prompted for his username and password for Alice's site. His download begins, and he gets billed via PayPal for the purchase.

Internally, the get_file.php script passes the commands and requested file to the Media Marker Framework. The framework first saves the PayPal watermarker request to an internal array of watermarker commands. Next, the framework recognizes that the next commands are to be logged, so it hashes the currently running framework file, the default fingerprinter plugin file, and the default watermarker plugin files. Also, it loads the rest of the session data needed for the log and the fingerprinter plugin. It loads the default fingerprinter and generates a new fingerprint. This fingerprint is used by the watermarker, so it is saved with the watermarker request to the internal array of watermarker commands. Next, the logger is loaded and a new line is appended to the log file. The zip_creator watermarker request, just like the PayPal one, is saved to the array. The array is now passed to the watermarker, which loads each of the requested watermarker plugins and chains them together. After the IO is finished, the result is a fingerprinted media file that is packaged in a ZIP file with an agreement for

private use only.

If Alice finds this copy of her photo in-the-wild, she can use the administration interface (a small collection of web forms) to extract the fingerprint data using the default watermarker plugin and default fingerprinter plugin. It tells her that the first part can be found in the log, so she copies it to the log search form and finds the log entry that matches. It also tells her how to verify the file with the remainder of the fingerprint, which she does and which also matches up. Additionally, the hashes of the code match the version she was using at the time, so the log and the code were never tampered with and she has the paper-trail to prove it. If Alice chooses, she could take Bob to court for compensation.

5 Maintenance

5.1 Compatibility

How does a user know if the Media Marking Framework will work on their server? The simple solution is to just try it, and see if any errors are returned. This is not an elegant solution, and doesn't provide the developer much help with making the system compatible. One of the problems with PHP is that functionality needs to be added at compile-time. (That is, the time when the PHP interpreter is compiled.) Web servers generally run different versions of PHP and have different functionality added. For example, some web hosting services still run the previous version of PHP, and some have "safe mode" enabled which limits some functionality. Some common libraries, such as the GD or PEAR libraries, aren't available if the webmaster hasn't added them. Additionally, some functionality may be limited by the OS PHP is currently running on. An ideal solution would be to generate a report from a compatibility test.

My implementation of the compatibility test starts by listing and including all of the Media Marking Framework files into the running code. Then, each file is parsed for function names, and each function name is checked with the "function_exists" and "method_exists" (if it is inside a class) PHP functions. If it is found, that means that it is a valid function that PHP can run. The test script outputs a "PASS" or "FAIL" for each tested function, so that the user knows exactly what will work before setting everything up.

5.2 Debugging

PHP doesn't have any sort of debugger. Debugging is normally done by executing the script, and watching for errors and warnings. Things can get

complicated since PHP is not strict with variables. Often, if a function normally returns an array, and an error occurs, it may return a boolean "false" instead. Passing "false" to another function that expects an array only causes more issues. Warnings or errors may or may not be raised, depending on the function. Debugging a PHP script sometimes becomes a difficult trial-and-error process.

In a production system, the performance hit is unbearable if there is debug code running all the time. Yet, there needs to be a way to turn on debugging in the production environment, so that bugs applicable to the unique environment can be fixed. There are no built-in debugging capabilities for PHP. When it seems like all hope is lost, an idea emerges that makes use of the power of PHP.

For most larger PHP applications, additional classes and files are included by the use of a "require" statement. It reads, parses, and executes a PHP file. This makes an ideal entry point for adding a debugger, since the debugger can emulate this behavior and include additional functionality. For the Media Marking Framework, a user can change the "require('media_marking_framework.php');" to "require('mmf_debug.php');" and then call the framework as normal. The "mmf_debug" emulates the original require statement, but adds debugging functionality.

For the Media Marking Framework it was desirable to include the ability to print text of what the system is doing, print the contents of variables, and have the ability to skip certain statements. The first two are typical for generating debug output, and the last has a unique application in the system. Some statements might cause problems if they are executed during a debug run. For example, garbage entries should not be added to a database while debugging. Also, the output needs to be the text debug output instead of the watermarked binary file. Skipping certain statements acts to sandbox the system and keep things clean.

Adding debug code into the production code can be messy, and can reduce performance, and therefore should be avoided. Instead, one can make use of something that should already exist to make the code less messy – the comments. The debugger makes use of a little bit of markup that has been added to the comments.

The debugger will translate anything that starts with “//d:” into a call to “debug_print”, so that only designated comments will be displayed. The translation includes passing the line number and current file to the “debug_print” function, so that it is easy to trace where the call was made from. the “//i:” is identical to the “//d:”, but signals that the next line should be skipped while debugging. For both types of debugging commands, inserting variable names into the comment will cause PHP to display the contents of the variable, which makes value checking easy.

The debugging system works by reading in the target file and storing each line as a string in an array. As it iterates through the array, it makes a number of changes to the code and processes the debugger comments. As mentioned before, it replaces anything that starts with “//d:” or “//i:” with a function call to “debug_print”. In the case of “//i:”, it sets the next line to empty. Additional code changes need to be made when they occur, so that the code can be properly processed by PHP's “eval” function. The “__FILE__” constant needs to be replaced with a string containing the filename, if it happens to be used anywhere in the code. The “<?php” and “?>” directives, which indicate that PHP code is between them, need to be removed. Lastly, any “require” statement made in the code needs to be removed, and the file it is pointing to needs to be recursively processed by the “process_and_require_file” function. After all these changes are made, the entire contents of the file can be passed to PHP's “eval” function, which processes the contents as-if it was another file. At this point the behavior of the “require” statement has been fully emulated, and the result has been fully instrumented with debugger code.

5.3 Profiling

Since there is a "debug_print" function that gets called from every converted comment, it is easy to add a timestamp to the debug output. Comparing these timestamps can indicate how long a function, or a block of statements, takes to execute. However, this would include the time it takes to repeatedly call the debug function, to output all the text, and includes the time when the OS is running other tasks. Currently, profiling is not implemented in the debugger for the Media Marking Framework.

5.4 Regression Testing

With the debugger instrumented code, regression tests can be written which look for certain text output to ensure that things are executing correctly. For regression tests, certain variables can be checked in the debug output to ensure that it is doing things correctly. Since most execution branches are also displayed, coverage testing can be performed to ensure that each segment of code gets hit when it is intended to.

For the Media Marking Framework, a test can be set up that initializes the framework with "require('mmf_debug');", starts a capture of the debug output buffer with the built-in "ob_start();" function, then makes the function call into the system to do the testing. Afterwords, the built-in "ob_get_clean();" function returns the debugger output and any errors or warnings, which can be parsed to

determine if the test passes. Regression tests are important to ensure that future bug fixes or enhancements will not break existing functionality, and that the framework works as expected in a target environment.

6 Satisfaction of Testable Metrics

6.1 Test Methodology

The testable metrics for the system are the requirements. Requirements define what the system "shall" do. There are two assessment types. Most of the requirements are of the traditional pass/fail type. When test criteria were hard to define they were subjectively evaluated as to whether they satisfy a "best effort". Testing a pass/fail type is easier since it can either pass or fail with no gray area in between. The "best effort" requirements are tested in a fuzzy logic fashion with some arbitrary, but satisfactory, threshold for determining if the measurement is considered passing.

A majority of the requirements can be verified by examining resources such as source code, existing studies, statistics, and other resources. The rest required some form of testing in order to reach a conclusion. For each requirement, the resources and tests will be presented in order to prove that the system meets the requirement. For all tests, the following provides details on the testing environment that was used.

Since computing resources were limited, the sample set consisted of five images. Image 1 is the classic Lena, a 512x512 pixel scanned image that is heavy in the reds. Image 2 is a small, 256x256 pixel higher-contrast image of a tree. Image 3 is a very dark, computer generated 800x600 pixel landscape image. Images 4 and 5 are scenery photos at 3 megapixels (2048x1536) and 5 megapixels (2592x1944) respectively. The images were chosen based on what a photographer might display or sell on their site.

Tests for the watermarker strength were conducted on a single lower-end computer using XAMPP for Windows. These tests consisted of basic number crunching where time and resources were not relevant. Scripts were run to generate the results, and are detailed below. The default watermarker was set to watermark up to 5,000 image blocks (8 pixels by 8 pixels in size) for all tests.

For testing the efficiency of the system, a lab environment was needed to better simulate a production environment. The best available machines at the University of Wisconsin – Platteville were in the upper level Computer Science and Software Engineering lab. They have an Intel Core2 processor (2 x 1.8 gigahertz), 2 gigabytes of RAM, and are connected over gigabit ethernet. For a clean software

environment, a Linux live CD was chosen to minimize the number of running programs. A live CD is a bootable CD-ROM disc which loads an operating system into RAM without the use of the hard drive. The live CD was Ubuntu 7.10, with Linux kernel 2.6.22-12-generic. The live CD was not modified, and ran in the default graphical mode. One of the two gigabytes of RAM was dedicated as a RAMdrive by Ubuntu. XAMPP for Linux (version 1.6.3b) was unzipped into the RAMdrive with the Media Marking Framework and test images. XAMPP was running Apache version 2.2.4, MySQL version 5.0.45, and PHP version 5.2.3.

6.2 Requirements

All downloaded media shall be properly fingerprinted.

Due to the plugin structure and flexibility of the system, it is entirely possible for the server to send out non-fingerprinted media. The assumption needs to made that the owner wants the media to be fingerprinted, and has set up the system accordingly. Under this mode of operation, we need to verify that the system cannot be tricked into sending out non-fingerprinted media.

There are a number of errors that are checked for during system execution, such as a failure to generate a fingerprint or a failure to log to the database. There are many more that can happen, and PHP is designed to print an error or warning, then ignore them and continue execution.

In order to display the error or warning, PHP sends the response headers and then the debug text. Just before sending the image, the watermarker does a check to see if the headers have been sent already. If they have, it is an indication that something has gone wrong and the execution terminates. It only sends the image after the fingerprint has been logged and after there have been no errors or warnings.

Errors can still happen in the watermarking stream, where the watermarking plugin fails to insert the watermark correctly. It is the plugin's responsibility to detect these errors and abort. It was assumed that all the plugins used in the system will be capable of doing this.

From a system standpoint, this requirement is satisfied under the assumptions above.

The system shall embed a fingerprint into media by using a watermark.
The watermarking system shall have a way to recover the watermark from

media.

Both requirements compliment each-other in that one generates the watermark, and the other recovers the watermark. Both of these requirements assume that the watermarking plugin works as expected by successfully inserting and extracting watermarks. In order to prove that both of these requirements are working as expected with a specific watermarking plugin, the following general steps must be followed.

Set up the system to use the target watermarking plugin, and download a sample media file from the system. A fingerprint should be embedded, and a corresponding entry should be added to the log file. Download another media file, and verify that the logged hashes are different. Use the administration interface to recover the embedded watermark from one of the downloaded media files, and verify that the watermark was recovered and matches what is in the log file. (Remember that for the default fingerprinter, the log contains the hash of the second half of the watermark.) Next, open the media in an editor and destroy it by making images a solid color, recording silence over sound, or whatever it takes to overwrite the media. Attempt to recover the watermark the same way as before, and verify that it is unable to find a watermark.

The process of embedding and extracting a watermark proves that a watermark can be embedded, and there is a way to recover it as well. Overwriting the media content proves that the watermark resides in the media content itself. Lastly, the watermark that is recovered matches the fingerprint generated, and downloading two images results in a different watermark, proves that the fingerprint is embedded as the watermark. Thus, with the default watermarker plugin, the two requirements are satisfied.

The system shall run on most common web servers.

According to the most recent (June 2008) Netcraft web server survey [17], about 85% of the active web servers on the Internet run either Apache or Microsoft IIS. Apache is the test environment for the Media Marking Framework, so it obviously supports PHP. For Microsoft IIS, PHP is supported by the use of a CGI binary. [20] PHP can be used on any web server implementation which allows CGI, thereby expanding the number of capable web servers to more than 85%.

If PHP is installed with the default extensions, the Media Marking Framework should run on it. No permissions need to be set to allow it to execute, which makes it friendly with hosting companies that only allow copying files. Also, it is not system dependent. PHP code runs the same on Linux and Windows no matter what type of processor. Based on all the above arguments, it is safe to say this

requirement is satisfied.

The system shall have the ability to handle various file formats.

The system reads the input file as a binary file, which means that the specific format of the file does not matter. Whether it is an image, video, music, or text, the file is processed as data and therefore any format is accepted.

The main concern about the file is its size. A small file can be read in and processed using a small amount of memory. A big file, however, could potentially require more memory than the system has available. Since the system will likely have to process larger and larger media files as time goes on, this concern was the primary drive for the design of the watermarker.

As shown in the function "streamWatermarkedMedia" in watermarker.php, a plugin chain is used to process the media file block by block. A loop is created to read a block of the input file and pass the data to the first plugin. The output from the first plugin is passed to the input of the next, and so on. The output of the last plugin is sent to the user. The loop continues until all of the plugins have finished. This method keeps the memory usage controlled no matter how large the input file is. Therefore, any size of any file format can be processed by the system, assuming the plugins are able to work with the data blocks.

It is important to note that the default watermarking plugin doesn't support this. It requires the entire contents of the image data so that it can be loaded by the built-in image functions. As the images get larger, PHP consumes more memory to process the image as a whole. If the images get too large, the system fails. A block-by-block watermarking plugin for images could be created, depending on the image format and how the watermark is added. The larger media formats, such as video and audio, are easier to process on a frame-by-frame or block-by-block basis due to the nature of the formats.

The system shall integrate with existing web services.

The system resides in its own directory to not interfere with whatever may already exist on the web server. PHP files do not require any extra setup to be performed on the server, assuming PHP has been previously set up. Simply copy the files to the server and they are able to be executed.

In order to use the framework, a PHP function call must be made. If the existing system already uses PHP to gather statistics on file downloads, do financial transactions, enforce access policies, or whatever else, the function call can be

made there instead of streaming the file contents. For many other systems, the HTML contains a link to the media. A file wrapper can be used to make the call into the framework and deliver the media. Instead of "<a href='movie.avi'>", the wrapper can be added to the link: "<a href='*get_file.php?file*=movie.avi'>". If that is too much work, the web server software may be set up to redirect web requests. If a call is made into the media directory, a redirect can be made to the file wrapper instead.

Since the installation is non-intrusive, and with the flexibility of calling the system, the system should integrate into most existing web services.

The system shall be efficient, scalable, and flexible.

This best-effort requirement was considered for all aspects of the design and implementation of the system. Efficiency was improved by only loading the plugins as needed and minimizing the code for the most time-critical functionality. Scalability was improved by limiting the dependence on external items, such as files and databases. Flexibility was inherited from PHP and arbitrary limitations were avoided.

Efficiency was tested in the lab in two ways. First, requests were timed on the live system in order to find an average response time. Second, the live system was measured to find the maximum server load that could be handled.

The response times were measured as being the time it takes from the initial connection to the time when all data has been received. The same five images were used as with all other tests. Table 2 lists the average response times. The first column is the framework using the "get_file.php" wrapper, generating a fingerprint, logging to both a local file and database, and using a null plugin which only passes the original data through it. The second column uses the "get_thumb.php" wrapper and only generates a thumbnail scaled down to 300 pixels. The third column is the same as the first, except that the default watermarker is used to embed the fingerprint. Numbers in the third column are rounded up to the nearest whole number, and the rest are rounded to two decimal places.

	Null Plugin	Thumbnail	Watermarked
Image 1	0.23s	0.43s	18s
Image 2	0.22s	0.35s	5s
Image 3	0.22s	0.40s	22s
Image 4	1.26s	2.08s	27s
Image 5	1.88s	3.12s	29s

Table 2: Response times for web requests.

The results show that the framework itself executes quickly, and the image processing is the major showdown. Typical web-sized images can be processed by PHP in under half a second. Passing large amounts of data is shown to have an effect on the response times as shown by images 4 and 5. However, the major slowdown is the manual image processing done by the default watermarker plugin.

The server performance was measured using httperf [21], a website performance measurement tool. Httperf was set to make web requests at a specific rate for 60 seconds and to measure the actual rate that the server could handle. For low request rates, the specified value and actual value were equal. At some point, the actual rate would plateau. The plateau is the peak load that the server is able to handle. Table 3 reports the requests per second (rps) that the server was able to handle for thumbnailed and watermarked images. For comparison, Image 1 with the null watermarker was measured to be 92 rps. Other images were not tested with the null watermarker.

	Thumbnail	Watermarked
Image 1	8.7 rps	0.1 rps
Image 2	14.7 rps	0.4 rps
Image 3	10.9 rps	< 0.1 rps
Image 4	2.2 rps	< 0.1 rps
Image 5	1.4 rps	< 0.1 rps

Table 3: Measured peak server performance, in requests per second.

Since a single watermark request takes up to 30 seconds, it should come as no surprise that the server is unable to respond to more than a few each minute. All of the performance numbers correspond to the response times, since the response

time has a direct effect on how many requests the server can handle. The processing is CPU intensive, and the CPU is what limits the server performance in this case.

There are several ways in which performance can be improved. First is by changing the 5000 block limit, so that the watermarker processes a smaller number of blocks. The side-effect of this method is that the watermark will be less redundant, and therefore more susceptible to attack. Another option is to distribute the processing in a cluster to improve the requests per second performance, but not change the response time. If static content needs a performance boost, such as the generated thumbnails that don't have a unique fingerprint in them, a caching plugin can be introduced. Lastly, an upgrade of the hardware or of the watermarker plugin should help.

Although PHP provides a great deal of flexibility, its performance is limited due to being interpreted. One would expect that the CPU intensive tasks could ported to a compiled language. In addition this would make it possible to get multiple processes going in order to make better use out of multicore environments.

All things considered, the framework itself is efficient. The problem is with the watermarking plugins and the amount of processing that needs to go into those tasks. Speed can be improved, but at the expense of the watermark robustness. Performance should improve as technology improves.

Scalability is primarily held back by the limitations of PHP. There is a limit on how big files can get, and how much processing the plugins can do before the PHP configuration needs to be changed. Other than that, the system can scale up as far as it needs to go. There are no limitations on the number of media files. The media repository and database can be external, so the framework can be used in a redundant or load balanced cluster of any number of nodes.

Flexibility is inherited from PHP, and was consideration in the framework design. The media repository can be a local directory or website, or a custom file handler can be added for a data repository. Plugins are unrestricted, as long as they pass data through or generate a fingerprint. This allows a wide variety of plugins to be created for things beyond only watermarking. Lastly, the database logging uses PHP's PEAR interface allowing support for any common database.

The watermarking system shall be easily upgradeable.

The watermarking plugins are only required to process the data as it passes through, which is usually the extent of the functionality for most watermarking algorithms presented in academic papers. Generating a plugin from an existing

academic watermarker should be an easy task for the academics. Adding the plugin to the framework requires two steps. First, the plugin is copied to the plugins directory. Second, the framework entry point (such as “get_file.php”) is changed to use the new plugin. A simple upgrade only requires the first step, assuming that the old watermarker is overwritten with a newer version.

The watermark should defend against unauthorized removal or tampering.

This best-effort requirement relies on the strength of the watermark plugin. The default watermarker was tested against common image manipulation attacks to find the survival rate and the point at which the watermark was damaged beyond recovery.

Table 4 shows the survival rate of a text message under JPEG compression. The survival rate for a bit is found by tallying the binary values. If there are 49 0's and 51 1's in a very noisy file, we would declare the bit as a 1 with a 2% survival rate. If anything is below 15%, it is marked as a question mark because it's too noisy. To find the survival rate for all the bits, the values are averaged. In Table 4, the message fully survives with a 32% survival rate, but practical recovery can happen down to 25%. Searching the log for the in-tact section “This is a” will likely find a limited number of entries. Although some of the message is corrupt, the corrupt portions can be guessed and match up with what is in the log. Although the corruption is highly dependent on the attack, manual testing showed that the point in which the watermark is damaged beyond recovery is normally between 20% and 30%.

Survival (%)	***Text***
20	T??s i? ??t??????s?AG?. ???2?45??
22	Th?s is ??t??? ??s?AGe. ???2?45??
24	Th?s is ??t??t?m?ssA?e. 0?2?4567
25	This is a?te?t?m?ss??e. 012?456?
27	This i? ??te?t mess??e. 01234567
28	This is a?te?t?m?ssa?e. 01234567
30	This is a?te?t messa?e. 012345?7
32	This is a test message. 01234567

Table 4: Survival rate and recovered text from a sample JPEG compressed image. The image was watermarked and subjected to JPEG compression at varying qualities in order to create the table.

Watermark survivability was tested on the same five images and are shown as the following colors (in order): black, red, green, blue, purple. Images were tested with eight different binary patterns and the results were averaged together. The eight patterns were: 01010101, 10101010, 11001100, 00110011, 11110000, 00001111, 11111111, 00000000.

Cropping was performed by replacing the outside border of the image with black of variable size. A cropped image is a smaller subset of the original. Since the image sizes need to be identical, it is assumed that the cropped image is correctly positioned on a black background in order to match up the good blocks. As expected, the more of the image that was black, the lower the survival rate. The black may have been contributing to the noise, so a future improvement may be to detect that the image block is gone and not include it in the recovery. The watermarks survived down to about 65% of the image missing.

The resizing algorithm that was used is PHP's 'imagecopyresampled'. The algorithm is tough on the watermark, even when making the image larger. The only exceptions are half and quarter increments. The survivability was very good for 50%, 75%, 100%, 125%, 150%, 175%, 200%, and beyond but was not so good for other increments due to the nature of the algorithm. Recovery might be difficult, but the watermarks survive down to 30-60% of its original size, depending on the image.

Blur and sharpness don't have much of an effect on the watermark, unless the blur or sharpness is beyond what PHP's 3x3 convolution matrix is capable of. As indicated in the graph, smaller numbers produced a larger effect on the image. It is fair to say the watermark survived everything in these tests.

Contrast and brightness tests had high survival rates. The only exception is a very low-contrast image. By this point, the quality is poor. The images would need to be very bright, very dark, very grainy, or very flat-toned in order to remove the watermark.

JPEG survivability is as expected since JPEG directly clips off the watermark data as quality decreases. The watermark survives down to about 20% quality, which is a fairly blocky image with the JPEG artifacts.

Format conversions generally try to preserve quality, and therefore preserve the watermark. The only exception is the GIF format, which replaces every pixel by a palette value. This alone shouldn't have an effect on the watermark since it is hidden in the frequency domain. However, GIF downsamples to only 256 colors, which has an effect in the frequency domain. Similar colors all become identical, which flatlines the frequency, and changes between color values are much more

rigid. This conversion is enough to wipe out the watermark, but reduces image quality by making it grainy.

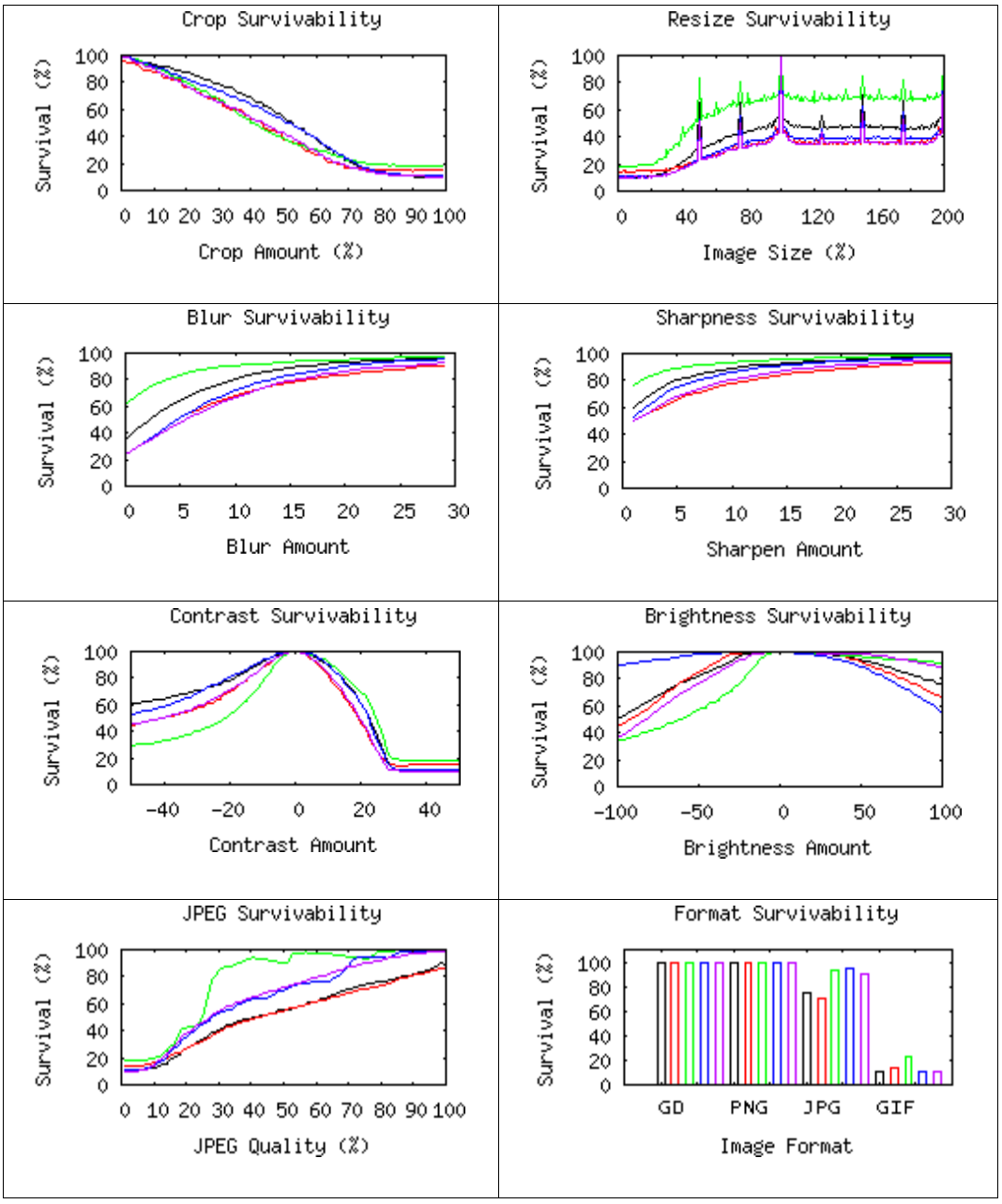

Figure 5: Survivability charts for the images listed in the Test Methodology section. Assigned colors for images 1 to 5 are black, red, green, blue, and purple respectively.

Although there are methods that can damage the watermark beyond recovery, they come at the cost of also damaging the image quality. Remember that the watermark is hidden in the least significant portion of perceived image quality. If the image quality survives, it is very likely that the watermark will as well. An attacker would need to purposely damage the image and hope that it is enough to render the watermark unrecoverable.

In conclusion, this requirement is satisfied under the best effort criteria. Methods that are used to compromise the watermark have the undeliverable effect of also compromising the quality and thus the value of the image. To remove the watermark, an attacker needs to willingly sacrifice the quality and hope that it is enough.

Fingerprints shall identify the original downloaders.

The log file or database stores the information for a transaction and generates a hash of that information which is inserted into the media. The fingerprint identifies the specific transaction in the log, which in turn identifies the username of the person logged in. This requirement is satisfied as long as two assumptions are made. First is that the framework entry point ("get_file.php") is password protected by the server, requiring the user to log in. Second is that the username corresponds to a real person. With websites who sell media, there is a real person identified by each of the payment sources. It is up to the website administrator to ensure that a real person can be identified by the username that they are using.

Fingerprints shall maintain downloader privacy.

Fingerprints generated using the default fingerprinter plugin protect downloader privacy. There is no personal information in the fingerprint. It consists of a random number, and a hash of the session information. A hash is a one-way algorithm that scrambles data, so that the original data cannot be found based on the scramble. The fingerprint data is only useful when the log database can be searched. In other words, it is only useful to the people who distribute the media.

Fingerprints shall be difficult to duplicate or forge.

Looking at Figure 3, consider all of the points in which something can be forged. Remember that there is a chain built up between the public packages and the fingerprint. Breaking any link in the chain provides evidence that something has been forged. Additionally, the chain minimizes the possibility of duplication.

The publicly available packages should be verified by the publicly available hash after being downloaded. A mismatch means that the public packages are invalid, or that the download is corrupt.

For each transaction, the framework does a hash of itself and all needed files. This creates a publicly verifiable string representing the running system. If the code has been changed by the administrator, the generated hashes will not match the public ones. However, this is best-effort. A truly savvy administrator could change the code to forge the Seller Private String by skipping the hash calculations, and have the ability to duplicate the watermarked media files. This possibility should be evaluated if there is evidence of tampering on the server, but is unlikely since the administrator risks losing compensation.

The log line for each transaction consists of a session info and a hash. If tampering occurs in the log, such as changing a username, IP address, date, time, or anything else, the hash will no longer match the data. In order to make things match, the hash can be regenerated with the new data, but this only breaks the next link in the chain. It is important to note that the original fingerprint cannot be found in the log, therefore making it impossible for the administrator to regenerate the watermarked media file. Half is in the log, and the other half only exists in the watermarked media file which the administrator should never see.

The two-way hash link between the watermarked media file and the log ensures that both parties cannot forge the fingerprint. The administrator cannot duplicate the watermarked media file and release it to frame someone else. Also, the user cannot change the fingerprint to point to someone else. Any modification by either party breaks the chain. Note that the watermarker should defend against any sort of tampering by the user, since the user wants to break the chain.

One assumption that is made is that the watermarked media file is not copied while in transit to the user. The website administrator should employ link-level encryption, such as SSL, to ensure that only the target user receives the media file.

7 Improvement Over Existing Systems

DigiMarc [1] uses the latest in watermarking technology to insert owner information into the media. It is the leader in commercial watermarking technology, and they offer a variety of services that are far beyond the scope of this project. Although they are a leader in watermarking technology, they are lacking in fingerprinting technology. Their solutions for “Content Serialization (Forensic Tracking)” appear to only insert a serial number into each media file. This method opens the door to forgery and duplication of the media by the

administrator. It is a weak link between the media file and the media distributor. The default fingerprinting method for the Media Marking Framework offers a much stronger link that eliminates the risks associated with a simple serial number.

Attributor [22] offers a service that scans the entire internet looking for a user's media content. This service is great for finding where media content is, but offers no way to find who stole it from the user and posted it elsewhere. This service could be a compliment to the Media Marking Framework, but they require the original media. Since each copy is unique in a slight way, Attributor may not detect them as a match. Additionally, copies of the original media should only be in the hands of the producer.

The Fraunhofer Institute [2] has created a tool to embed unique fingerprints, via a watermark, into MP3 audio files. Similar to the Media Marking Framework, they embed a hash value that identifies a specific registered downloader. However, this single hash value can be duplicated by the website owner, limiting its authenticity. Their system only supports their specific watermark algorithm and MP3 files. The Media Marking Framework by default uses a stronger fingerprint and allows any algorithm for any file format via the use of plugins.

The ISWAR Watermarker [3] offers a fairly robust watermarking option, but only embeds a logo. Testing the media file only reveals a “yes” or “no”, with no further information. It is possible that a “no” would be returned while a degraded logo still exists. Finding the logo proves ownership, but offers no information about who illegally copied the media. The Media Marking Framework default fingerprint offers much more information than what a logo can contain.

Lastly, there are a variety of DRM solutions available. DRM focuses on preventing illegal use of content by the users by restricting the media to specific client-side players and programs. There are ways around most DRM systems, such as the “analog hole” which records the music as it is played out of the speakers. Additionally, many DRM systems are over-restrictive and prevent legitimate uses of the media such as making backups or playing it on a non-DRM music player. The Media Marking Framework does not require client-side software and allows the users to do anything they wish with the media.

8 Future Work

As a plugin-oriented framework, additional plugins are always handy. Several non-watermarking plugin suggestions have been presented in other parts of the paper. Watermarkers should also be developed for other file formats, and stay up-to-date with the latest technology.

The recovery process should be streamlined for automated services such as Attributor. A service that crawls the web looking for a producer's files would prove valuable for quickly finding infringement and measuring the impact. Not only would the producer know who to take to court, they would also know the extent of the damage from the infringement and could better estimate a dollar value.

Currently, the seller's private key is generated by taking the MD5 hash of the original image, and crunching it down further using a CRC to bring the size to 4 bytes. This is the maximum size that is allowed to seed PHP's random number generator. Improvement can be made by introducing a random number generator that accepts input of any size, such as the one used in the RC4 encryption algorithm.

For improved quality of the Media Marking Framework, regression tests and coverage tests should be created. Regression tests are used to test all of the features of the system, ensuring that nothing is broken during updates for new releases. Coverage tests extend regression tests to ensure that every line of code can be hit, and that the system remains stable and clean.

The aim of the Media Marking Framework is to be accessible by even small, independent producers who may not make profit. Although the Framework is intended to be open-source, commercialization is possible in two ways. First is that plugin authors can license their closed-source plugins. If a producer wants a better plugin, or a plugin for a different format, they could purchase one. Second is that a website could be created as a distribution point for producers using the Framework as a back-end. Producers could upload and maintain the media in a secure repository, and the website could handle sales transactions and distribution.

9 Conclusion

With a strong fingerprinting algorithm, the latest and greatest watermarking algorithms, and an elegant framework to house it all, the Media Marking Framework will likely be a strong weapon against copyright infringement and piracy.

References

[1] Digimarc Corporation, “Digimarc MyPictureMarc”, 2007. [Online] Available: http://www.digimarc.com/mypicturemarc. [Accessed: March 7, 2007].

[2] J. Blau, “Fraunhofer makes tool to fight music piracy using digital watermarking technology.”, PC Advisor, 2006. [Online] Available: http://www.pcadvisor.co.uk/news/index.cfm?newsid=5671. [Accessed: March 7, 2007].

[3] S. Mohanty, “ISWAR (Imaging System with Watermarking and Attack Resilience)”, University of North Texas, 2000. [Online] Available: http://www.cse.unt.edu/~smohanty/ISWARwatermarker. [Accessed: March 7, 2007].

[4] E. Muharemagic and B. Furht, “Survey of Watermarking Techniques and Applications”, Florida Atlantic University, 2005.

[5] M. Stumpfl, “Digital Watermarking”, University of Southampton, 2001.

[6] D. Chen, M. Ouhyoung, J. Wu, “A Shift-Resisting Public Watermarking System for Protecting Image Processing Software” in *IEEE Transactions on Consumer Electronics*, Vol. 46, No. 3, 2000.

[7] C. Jin and J. Peng, “Robustness of a Blind Image Watermark Detector Designed by Orthogonal Projection”, in *Electronic Letters on Computer Vision and Image Analysis*, 2004.

[8] WorldStart.com, “Image File Guide”. [Online]. Available: http://www.worldstart.com/guides/imagefile.htm. [Accessed: March 7, 2007].

[9] G. Hoffmann, “JPEG Compression”, 2003. [Online]. Available: http://www.fho-emden.de/~hoffmann/jpeg131200.pdf. [Accessed: March 7, 2007].

[10] G. El-Taweel, H. Onsi, M. Samy, M. Darwish, “Secure and Non-Blind Watermarking Scheme for Color Images Based on DWT”, Cairo University, 2003.

[11] F. Wang, J. Pan, L, Jain, H. Huang, “Digital Image Watermarking Approach Based on Lapped Orthogonal Transform” presented at IEEE Asia-Pacific Conference on Circuits and Systems, 2004.

[12] C. Hass, “JPEG Quality and Quantization Tables for Digital Cameras, Photoshop”, ImpulseAdventure, 2007. [Online] Available: http://www.impulseadventure.com/photo/jpeg-quantization.html. [Accessed: March 7, 2007].

[13] B. Rosenblatt, “2006 Year in Review: DRM Technologies”, *DRM Watch*, 2006. [Online]. Available: http://www.drmwatch.com/drmtech/article.php/3650401. [Accessed: March 7, 2007].

[14] N. Memon and P. Wong, “A Buyer-Seller Watermarking Protocol”, Polytechnic University, 2001.

[15] E. Rescorla, “RFC 2631 Diffie-Hellman Key Agreement Method”, *The Internet Engineering Task Force*, 1999. [Online]. Available: http://tools.ietf.org/html/rfc2631. [Accessed: March 7, 2007].

[16] R. Rivest, “RFC1321: The MD5 Message-Digest Algorithm”, RFC Editor, 1992.

[17] Netcraft Ltd., “June 2008 Web Server Survey”, 2008. [Online] Available: http://news.netcraft.com/archives/2008/06/22/june_2008_web_server_survey.html. [Accessed: July 12, 2008].

[18] D. Sklar and A. Trachtenberg, *PHP Cookbook*, O-Reilly & Associates, Inc., 2003.

[19] The PHP Group, “PEAR :: Documentation”, 2008. [Online] Available: http://pear.php.net/manual/. [Accessed: July 12, 2008].

[20] Microsoft Corporation, “PHP on IIS”, 2008. [Online] Available: http://www.iis.net/php/. [Accessed: July 12, 2008].

[21] Hewlett-Packard Development Company, L.P., “Welcome to the httperf homepage”, 2008. [Online] Available: http://www.hpl.hp.com/research/linux/httperf/. [Accessed: July 12, 2008].

[22] Attributor Corporation, “Attributor: Program the Web”, 2008. [Online] Available: http://www.attributor.com. [Accessed: July 12, 2008].

Appendix A: Plugin Specifications

Naming

The class name must exactly match the file name, with exception to the ".php" extension. For example, default_fingerprinter.php begins with "class default_fingerprinter". This applies to both the fingerprint and watermark plugins.

Command-Line Functionality

For testing purposes and command line usage, code can be added outside of the class implementation. The plugins are executed by the framework without any command line arguments, so there must be no output unless there are one or more command line arguments. In other words, all command line functionality should be wrapped in an 'if' statement checking if 'argc' is non-zero or if 'argv' it is empty.

Fingerprint Plugin

function getPluginInfo() returns a hashtable with the following attributes:
- 'files' – lists each needed file, including the current one.
  - 'name' – filename, ex. "default_fingerprinter.php".
  - 'version' – version information, as text.
- 'fpsize' – size of the fingerprint in bytes if constant, or 0 if variable.
- 'info' – text description of this plugin.

function generateFingerprint(
- Requested media filename,
- Username of the client,
- IP address of the client,
- Timestamp of when the request was made,
- Random number for this session (binary, 16 bytes),
- Hash of the log data in hex (text, 32 bytes),
- Extra arguments that can be included)

Return the binary fingerprint. If an error occurs, terminate execution ('die') with a helpful error message.

function translateFingerprint(
- Fingerprint data)

The fingerprint data can be text or hex depending on the watermarker. If it is hex, the question mark ('?') represents any nibbles that could not be recovered.
Return a hashtable with the following attributes:
- 'report' – a full text report of what the fingerprint means or contains.
- 'searchfor' – search for this text in the log files.

Watermark Plugin

function getPluginInfo() returns a hashtable with the following attributes:

'files' – lists each needed file, including the current one.

'name' – filename, ex. “default_fingerprinter.php”.

'version' – version information, as text.

'info' – text description of this plugin.

'mime' – MIME type of the output file, such as “image/png”.

'ext' – file extension, such as “png”.

function generateWatermarkInit(

Fingerprint data,

Extra arguments that can be included)

Doesn't return anything. If an error occurs, terminate execution ('die') with a helpful error message.

function generateWatermarkProcessIO(

A block of binary data (or null if end of file))

Return a block of processed binary data, an empty string if still processing, or null if end of file. If an error occurs, terminate execution ('die') with a helpful error message.

function testFile(

Found file to test (already open file resource),

Original file (already open file resource),

Key or password,

Fingerprint size in bytes,

Be quick (boolean, do a full detailed report if false),

Extra arguments that can be included)

Return a hashtable with the following attributes:

'report' – A text report of the recovery and what the watermark is.

'watermark' – Text of the found watermark.

If the watermark is binary, the 'watermark' output should be in hex with a question mark ('?') representing any unrecoverable nibbles.

Appendix B: Code

Availability

Source code is available at the University of Wisconsin – Platteville Computer Science and Software Engineering website.

http://www.uwplatt.edu/csse/Theses

Contents

media_marking_framework_thesis_code.zip

\example_gallery.html
\get_file.php
\get_thumb.php
\media\3mp.jpg
\media\5mp.jpg
\media\face.png
\media\house.png
\media\house2.png
\media\jet.png
\media\lena.png
\media\tree.png
\media_marking_framework\compatability_check.php
\media_marking_framework\fingerprinter.php
\media_marking_framework\logger.php
\media_marking_framework\media_marking_framework.php
\media_marking_framework\mmf_debug.php
\media_marking_framework\mm_admin.php
\media_marking_framework\mm_config.php
\media_marking_framework\plugin_controller.php
\media_marking_framework\watermarker.php
\media_marking_framework\plugins\fingerprinter\default_fingerprinter.php
\media_marking_framework\plugins\watermarker\invisible_watermark.php
\media_marking_framework\plugins\watermarker\thumb_gen.php
\media_marking_framework\plugins\watermarker\visible_text_watermark.php