# Fingerprinting web application platforms by variations in PNG implementations

Dominique Bongard[1]

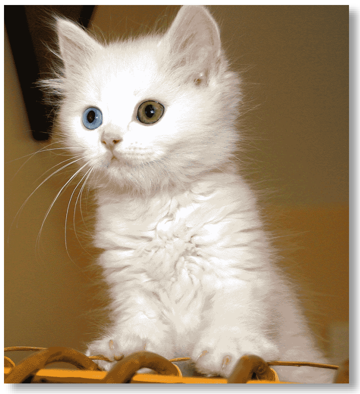
[1]*0xcite, Switzerland, dominique.bongard@0xcite.ch*

**Abstract**. Fingerprinting is an important preliminary step when auditing web applications. But the usual techniques based on the analysis of cookies, headers and static files are easy to fool. Fingerprinting digital images is a technique commonly used for forensic investigations but rarely during security audits. Moreover, it is mostly based on the analysis of JPEG images only. In this paper, we study the implementation differences between a number of PNG decoders/encoders, either build-in or commonly used with the main web application development platforms. As a result, we release a python script and a set of crafted images that can discriminate between various PNG libraries. With this tool, it is often possible to identify the platform behind a web application even when an effort has been made to prevent fingerprinting, as long as said application will decode arbitrary PNG images.

**Keywords:** audit, forensic, imaging

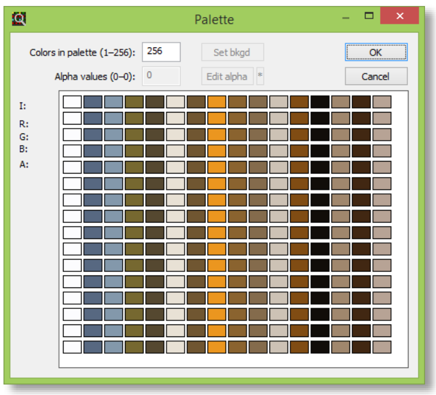
## 1 Introduction

After reading the infamous *POC || GTFO 0x03* [1] in which Ange Albertini, the author of *corkami.com* [2], demonstrated his talent at file format manipulation, we got interested in playing similar tricks with PNG files. In particular, we crafted an out-of-spec image in such a way that some picture viewers will show a kitten while others will show a tiger. Of course, many applications will simply refuse to display such a corrupted file.

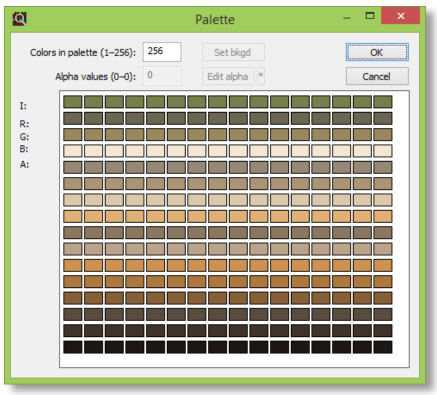


The trick to generate such a file is to start with two 16 color (4bit) indexed images of the same dimensions. The two pictures are then merged together in a 256 color (8bit) indexed image in such a way that the color index of each resulting pixel contains the 4bit color index of the corresponding pixel in the first image in its high nibble, respectively the second image in the low nibble.

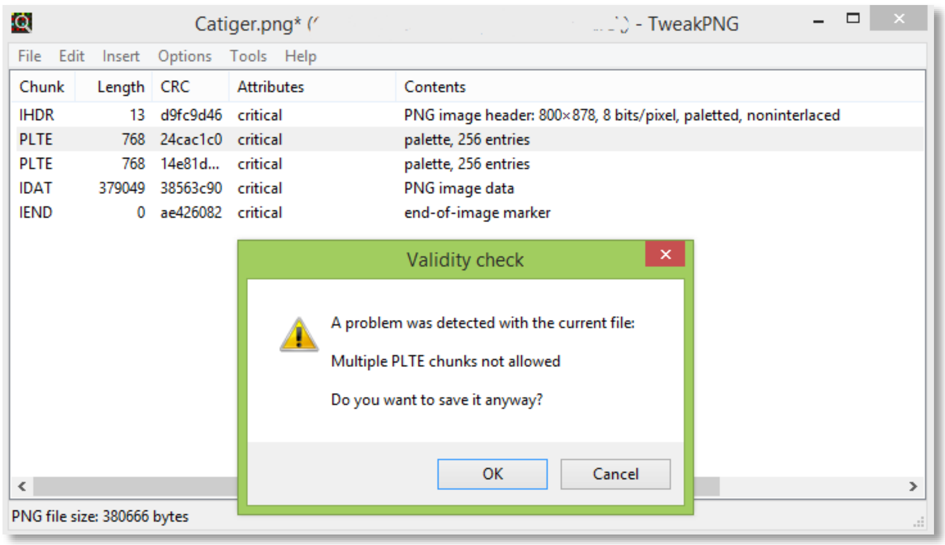
It is then possible to produce two 256 color palettes for this resulting image, one which will decode the kitten and one which will decode the tiger. This technique was once used on older 256 color systems to produce smooth transitions between images by gradually morphing the palette [3].



Colors in palette (1–256): 256
Alpha values (0–0): 0



Colors in palette (1–256): 256
Alpha values (0–0): 0



| Chunk | Length | CRC | Attributes | Contents |
|---|---|---|---|---|
| IHDR | 13 | d9fc9d46 | critical | PNG image header: 800×878, 8 bits/pixel, paletted, noninterlaced |
| PLTE | 768 | 24cac1c0 | critical | palette, 256 entries |
| PLTE | 768 | 14e81d... | critical | palette, 256 entries |
| IDAT | 379049 | 38563c90 | critical | PNG image data |
| IEND | 0 | ae426082 | critical | end-of-image marker |

Validity check

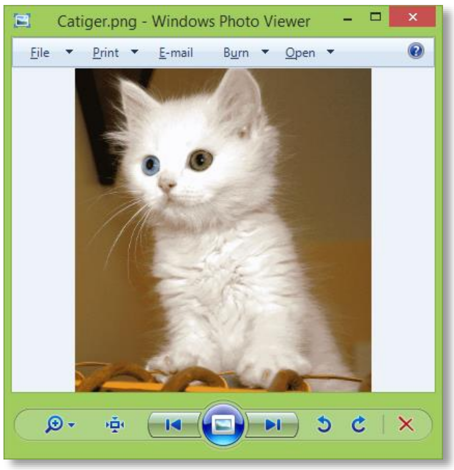
A problem was detected with the current file:

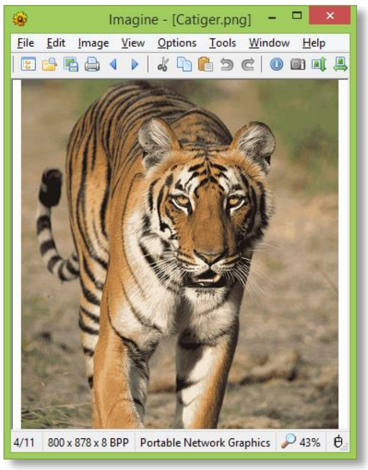
Multiple PLTE chunks not allowed

Do you want to save it anyway?

PNG file size: 380666 bytes

We can then use TweakPNG to craft a PNG containing both palettes. While some programs will correctly detect that the image is invalid, others will decode the image using the first palette and the rest will use the second one. Debian’s file manager for example will not display such images. Windows on the other hand will use the first palette, showing the kitten, but the Imagine viewer[1] will use the second palette and therefore display the tiger.





[1] http://www.fosshub.com/Imagine.html

While searching for software and web services that would display either one or the other version of the "Catiger" crafted picture, it struck us that this technique could be used for fingerprinting web application platforms through theur underlying image libraries.

## 2 Value of Fingerprinting

Digital image fingerprinting is a technique mainly used in the world of forensics. It is often necessary to show in court that an image has not been tampered with, or that it comes from a given camera, or at least a specific model. In such a setting, fingerprinting is usually done on JPEG images only. The features used include physical characteristics of the image sensor, pixel-based analysis, image metadata as well as variations in JPEG encoder implementations. A good introduction to Digital Image Forensics is found in [4].

In penetration testing, fingerprinting significantly helps the auditor assess his target and then plan his attacks for the maximum likelihood of success [5] [6]. Most web application fingerprinting tools rely on markers such as HTTP headers, cookies or static files.

Image fingerprinting has also found more niche applications. The backend of at least one social network application for iPhone, whose developer prefers that it were not mentioned in this paper, uses JPEG file fingerprinting to detect spam images, which are significantly different from pictures genuinely generated by an iPhone.

Last but not least, many web applications, even among those written in modern high level languages, rely on native image manipulation libraries written in C. Those might contain memory corruption bugs leading to remote code execution, as has happened with the iPhone [7].

## 3 Related Work

Digital image fingerprinting has been extensively studied for the purpose of legal forensics but most methods work at the pixel level. In [8] however, EXIF metadata in JPEG files is used to generate fingerprints. And in [9], the authors extend this technique to other metadata such as quantization tables. In particular, they are able to recognize pictures produced by *Flickr*.

In [10] Michał Składnikiewicz, shows how to craft a BMP file which will decode to different pictures depending on the image viewer.

In [11] Saumil Shah presents techniques to identify web applications at the HTTP level. In [12], Anant Shrivastava, shows how to thwart web application fingerprinting methods that are based on the analysis of HTML strings and static files. Several tools are available for the purpose of fingerprinting web applications, such as BlindElephant[2], WhatWeb[3] and Wappalyzer[4]. None of them however seems to be able to fingerprint based on the artefacts of PNG image encoders / decoders.

## 4 Abusing PNG decoders and encoders

### 4.1 Leveraging ambiguities in the PNG specification

The PNG specification [13] is useful to find mandatory characteristics, ambiguities, undefined behaviours and optional features. Indeed, the word "probably" appears 7 times, "might" 18 times and

[2] http://blindelephant.sourceforge.net
[3] http://www.morningstarsecurity.com/research/whatweb
[4] https://wappalyzer.com

“should” 115 times. While the specification describes fatal error conditions, it also declares: “**Recover from an error, if possible**”. This gives developers a lot of room for interpretation.

The source code of various image libraries can also be used to find bugs. Change logs, source control commit messages and code comments are as well most useful in finding sources of unique behaviour.

The information thus gathered was then used to craft a set of PNG images that trigger different results depending on the library that is used to decode them. Our set of crafted images was mainly created with the help of TweakPNG[5] and a hexadecimal editor.

## 4.2 Basics of the PNG file format

Every PNG file starts with the eight bytes signature (in decimal):

```
137 80 78 71 13 10 26 10
```

The remainder of the file is composed of “chunks” in a Length, Tag, Data, CRC format:

| | | | |
|---|---|---|---|
| LENGTH | TAG | DATA | CRC |

| | |
|---|---|
| Length | A four bytes unsigned integer giving the number of bytes in the chunk. Zero is valid. |
| Tag | A four letter character tag. Each character is restricted to the range 65 to 90 and 97 to 122. Bit 5 of each character has a special signification. |
| Data | The actual data of the chunk. Not present if Length == 0. |
| CRC | A four bytes CRC of the chunk. |

### 4.2.1 File signature

The 8 bytes PNG signature is quite long, and we can imagine that a lazy developer would only check part of it. We therefore generated a test image for which we modified the last byte of the signature. Surprisingly enough, all the tested libraries verify the complete signature.

### 4.2.2 Chunk properties

**Chunk CRC**

The verification of the CRC of chunks is not mandatory, at least for chunks that are not critical. Decoding libraries have varying policies regarding the validation of chunk checksums.

**Chunk length**

There is no much point in setting an invalid chunk length, since it will impede the parsing of the rest of the file. However, the various PNG decoders differ in behaviour when the length of the last chunk is invalid.

Some chunks always have the same length, for example 13 bytes for IHDR and 0 bytes for IEND. Some decoders don’t mind oversized chunks.

[5] http://entropymine.com/jason/tweakpng/

**Chunk naming**

Each chunk has a four alphabetical letters name. For each letter, its capitalization (bit 5) has a special signification. Most importantly, the capitalization of the first letter indicates if a chunk is critical for decoding the picture or not (ancillary). A decoder should ignore ancillary chunks it doesn't understand but abort if it encounters a critical chunk it doesn't support.

The third character of a chunk name must always be capitalized, but not all decoders test if this is the case.

Some decoders will simply ignore chunks with non-alphabetical characters in their names, others will abort and report the file as invalid and another batch will act as if they had reached the last chunk in the file. A similar behaviour occurs with critical chunks that have an unregistered name (not appearing in the specification). Some decoders will ignore such chunks and other will report an error.

**Chunk ordering**

The specification mandates a specific position or ordering for certain chunks in order to simplify the development of decoders. However, there is no fundamental reason while a PNG file with out of order chunks could not be decoded. As a consequence, some decoders are perfectly happy when chunks are not placed where they should be while others strictly enforce the rules set in the specification.

In output images, the ordering of chunks that can be freely placed will give information about the encoder.

**Chunk presence and repetition**

Some chunks can only appear in some types of images (e.g. only indexed images have a palette). As usual, some decoders will ignore out of place chunks and others will refuse the offending image.

Moreover, some chunks, like IHDR or PLTE can only appear once in a PNG file. Decoders have 3 ways of dealing with files that don't respect this rule. They can reject the file, keep the first instance of the chunk or keep the second instance.

### 4.2.3 Critical chunks

All the decoders must be able to process the critical chunks detailed in the specification. But we obtain diverging behaviour with a crafted PNG file which contains an unknown chunk tagged as critical.

**The IHDR chunk**

The IHDR chunk is always 13 bytes long. It must be the first chunk and appear a single time in the file. Some decoders don't care about the position of this chunk, while other bail out if it is not first. If two IHDR chunks appear in a file, some decoders will abort but others will either use the first or the second instance. Some decoders will verify the length of the chunk, but others will gladly accept a longer one with garbage at the end.

The IHDR chunk contains various information about the picture to be decoded like its dimensions, type, filtering method and compression. Either dimension must never be zero. Since only one valid value exists for the filtering and compression methods, some decoders do not bother checking these fields.

**The IEND chunk**

The IEND chunk is kind of the opposite of IHDR. This mandatory chunk signals the end of the file and must be the last chunk. Its length must be zero but some decoders don't care.

Some decoders ignore the IEND chunk and rely on the end of the file instead. Others will stop decoding as soon as they reach an IEND chunk, but don't mind if more data follows.

**The PLTE chunk**

The palette chunk is only present in indexed colour images. As it name implies, it contains the palette for the picture. It must appear only once and before the first data (IDAT) chunk. As we have seen, if two PLTE chunks appear in a file, different viewers will use either the first or the second palette if they don't reject the file outright. Some decoders will even accept a picture without a palette, but they will display an invalid image. As with IHDR chunks, some decoders care about the position of the chunk and other don't.

**The IDAT chunks**

IDAT chunks contain the pixels values of the image in a zlib compressed form. A PNG file must contain at least one IDAT chunk, but it can contain several which must be consecutive. Even though this would be useless, the length of an IDAT chunk can be zero. As we have seen before, only some decoders enforce that IDAT chunk follow each other. Some decoders balk on zero length IDAT chunks, even though those are legal. Most decoders simply feed the content of the IDAT chunks to the zlib decoder to get the pixels, but some verify that no junk bytes follow the compressed data.

### 4.2.4 Ancillary chunks

PNG decoders can optionally support a set of non-critical chunks and / or private chunks. If a crafted image featuring an ancillary (non-critical) chunk with an invalid content triggers an error, we can deduce that the targeted decoder attempted to process said chunk.

Some ancillary chunks, like the gamma chunk, can also influence the appearance of the decoded picture. For files with such chunks, the look of the output image will give information about the decoder.

The presence or absence of some specific ancillary chunks in output images can also give information about the encoder that generated them.

### 4.2.5 Zlib compressed data

The concatenation of the data in the IDAT chunks forms a binary blob in the zlib format [14]. Most PNG decoders use the official zlib implementation to decompress the data, but Go and Dart use a different implementation which has the particularity of accepting compressed blobs with an invalid window size.

Some decoders make sure that the size of the decompressed data is exactly the size of the necessary image data.

The zlib blob contains a compression level flag, which can be used to differentiate PNG encoders.

### 4.2.6 Scan line filtering

Each row of pixels in a PNG image can optionally be processed by one of four filters to help obtain better compression. Many encoders only support a subset of filters and can therefore be differentiated from the filters present in output files.

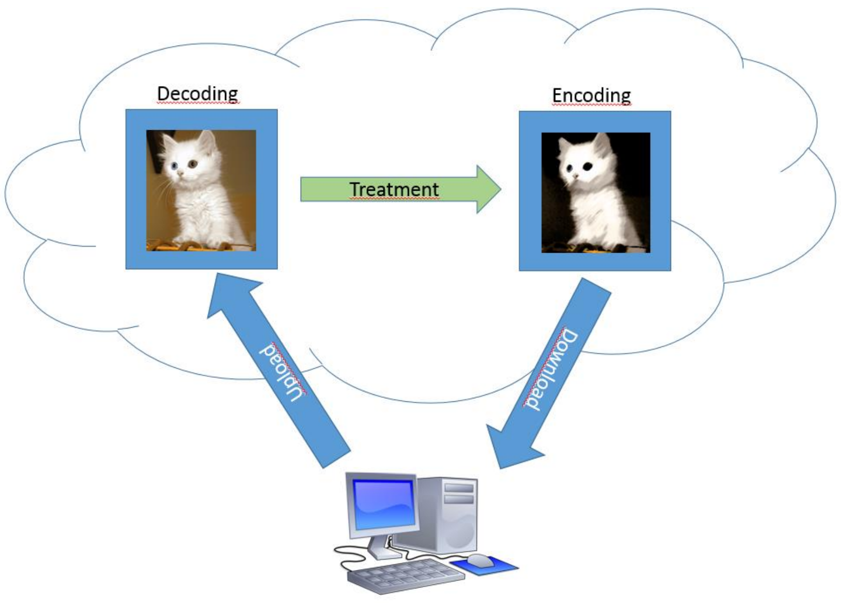
Many decoders will accept images with scan lines which have an invalid filter value.

## 5 Practical web application and library fingerprinting

Since the advent of Participative Websites, Web2.0, Social Networks, The Cloud and other similar buzzwords, many sites allow visitors to upload images for various purposes including photos albums, profile pictures, illustrations for classified ads etc.

Most web applications decode and re-encode the uploaded images before publishing them, either to verify they are indeed valid images, to remove unwanted metadata, to resize them or to achieve better compression etc.

The process is illustrated in the figure below.



It is therefore possible to generate a set of specially crafted input images and then use the process described above as an oracle to get a set of output images from a target web application.

Thanks to variations in the implementation of the PNG specification between different programming languages and web application frameworks, particularities in the set of output images will leak information about the identity of the image library that generated them.

## 5.1 The fingerping tool

We have crafted more than 50 PNG files designed to test the various scenarios described in this paper. We have then submitted those files to programs made with various common languages, frameworks and tools as listed below:

| | |
|---|---|
| Golang | 1.0.2 linux |
| PHP 5 GD | 5.4.9-4ubuntu2.4 |
| OpenJDK 7 | 7u21-2.3.9-1ubuntu1 |
| Python | PyPNG 0.0.16 |
| Python | PIL 1.1.17 |
| C# Mono | Debian 2.10.8.1-5ubuntu1 |
| C# MS .NET | 12.0.21005.1 |
| Node.js | Pngjs 0.4.0 |
| Ruby | ChunkyPNG 1.3.1 |
| ImageMagick | 6.7.7-10 2013-09-10 Q16 |
| Dart | Dart Image 1.1.21 |
| Erlang | erl_img  evanmiller fork |
| LodePNG | 20140609 |
| Haskell | JuicyPixels 3.1.5.2 |

The resulting data set was used to create *fingerping* [15], a Python script which can determine the identity of an image library from its unique way of processing our crafted PNGs.

## 5.2 Uncovered bugs

As shown below, some of our crafted images triggered serious errors in some of the tested image libraries:

**Java**

```
Exception in thread "main" java.lang.NegativeArraySizeException
      at com.sun.imageio.plugins.png.PNGImageReader.readMetadata(PNGImageReader.java:745)
      at com.sun.imageio.plugins.png.PNGImageReader.readImage(PNGImageReader.java:1229)
      at com.sun.imageio.plugins.png.PNGImageReader.read(PNGImageReader.java:1577)
      at javax.imageio.ImageIO.read(ImageIO.java:1448)
      at javax.imageio.ImageIO.read(ImageIO.java:1308)
      at Test.main(Test.java:15)
```

**Mono**

```
[ERROR] FATAL UNHANDLED EXCEPTION: System.ArgumentException: A null reference or invalid value
was found [GDI+ status: InvalidParameter]
  at System.Drawing.GDIPlus.CheckStatus (Status status) [0x00000] in <filename unknown>:0
  at System.Drawing.Bitmap..ctor (System.String filename, Boolean useIcm) [0x00000] in
<filename unknown>:0
  at System.Drawing.Bitmap..ctor (System.String filename) [0x00000] in <filename unknown>:0
  at (wrapper remoting-invoke-with-check) System.Drawing.Bitmap:.ctor (string)
  at Test.Main (System.String[] args) [0x00000] in <filename unknown>:0
```

**GoLang**

```
panic: runtime error: invalid memory address or nil pointer dereference
[signal 0xb code=0x1 addr=0x20 pc=0x4246cd]

goroutine 1 [running]:
runtime.panic(0x4d5120, 0x5f9dc8)
      /usr/lib/go/src/pkg/runtime/panic.c:266 +0xb6
image/png.(*decoder).decode(0xc21004f800, 0x0, 0x0, 0x0, 0x0)
      /usr/lib/go/src/pkg/image/png/reader.go:510 +0x82d
image/png.(*decoder).parseIDAT(0xc21004f800, 0x42c, 0x0, 0x0)
      /usr/lib/go/src/pkg/image/png/reader.go:521 +0x40
image/png.(*decoder).parseChunk(0xc21004f800, 0x0, 0x0)
      /usr/lib/go/src/pkg/image/png/reader.go:569 +0x39e
image/png.Decode(0x7f4c5237d088, 0xc210000060, 0x5fd8e0, 0x7f4c5237d088, 0x0, ...)
      /usr/lib/go/src/pkg/image/png/reader.go:625 +0x140
main.main()
      /home/debian/png/golang/test.go:15 +0x157
exit status 2
```

**Ruby**

```
/usr/lib/ruby/gems/1.9.1/gems/chunky_png-1.3.0/lib/chunky_png/canvas/png_decoding.rb:69:in
`from_datastream': undefined method `width' for nil:NilClass (NoMethodError)
```

**.NET**

```
System.Runtime.InteropServices.ExternalException: A generic error occurred in GDI+.

   at System.Drawing.Image.Save(String filename, ImageCodecInfo encoder,

EncoderParameters encoderParams)

   at Test.Main(String[] args)
```

**Haskell**

```
22698 Segmentation fault      ./test $1
```

## 6 Acknowledgement

We would like to thank Ange Albertini and Jean-Philippe Aumasson for their help and encouragement with this research.

The cat image is by Bertil Videt and the tiger image is by Sumeet Moghe. Both images were published under the Creative Commons Attribution-Share Alike licence.

## 7 References

[1] A. Albertini and T. Goodspeed, "POC || GTFO 0x03," 2 3 2014. [Online]. Available: http://www.exploit-db.com/wp-content/themes/exploit/docs/pocorgtfo03.pdf.

[2] A. Albertini, "Corkami.com," [Online]. Available: http//www.corkami.com.

[3] Polaris, "Creating Demos - Coder Tutorial #4 (Crossfading)," [Online]. Available: http://hugi.scene.org/online/hugi31/hugi%2031%20-%20coding%20corner%20polaris%20creating%20demos%20-%20coder%20tutorial%204.htm.

[4] H. Farid, "Digital Image Forensics," [Online]. Available: http://www.cs.dartmouth.edu/farid/downloads/tutorials/digitalimageforensics.pdf.

[5] OWASP, "Fingerprint Web Application Framework," 03 2014. [Online]. Available: https://www.owasp.org/index.php/Fingerprint_Web_Application_Framework_(OTG-INFO-009).

[6] OWASP, "Testing for Web Application Fingerprint," 09 2013. [Online]. Available: https://www.owasp.org/index.php/Testing_for_Web_Application_Fingerprint_(OWASP-IG-004).

[7] F. Truta, "Apple Fixes Multiple FreeType, libpng Flaws with iOS 4.1 for Apple TV," [Online]. Available: http://news.softpedia.com/news/Apple-Fixes-Multiple-FreeType-Flaws-with-iOS-4-1-for-Apple-TV-168237.shtml.

[8] P. Alvarez, "Using Extended File Information (EXIF) File Headers in Digital Evidence Analysis," *International Journal of Digital Evidence,* no. Winter 2004, Volume 2, Issue 3, 2004.

[9] E. Kee, M. Johnson and H. Farid, "Digital Image Authentication From JPEG Headers," *IEEE Information Forensics and Security,* vol. 6, no. 3, pp. 1066 - 1075, 2011.

[10] M. G. C. Składnikiewicz. [Online]. Available: http://nfsec.pl/hakin9/bmp.pdf.

[11] S. Shah, "HTTP Fingerprinting and Advanced Assessment Techniques," 07 2003. [Online]. Available: www.blackhat.com/presentations/bh-usa-03/bh-us.../bh-us-03-shah.ppt.

[12] A. Shrivastava, "Web Application finger printing Methods/Techniques and Prevention," 7 2011. [Online]. Available: http://anantshri.info/articles/web_app_finger_printing.html#.Uz840PmSwYM.

[13] W3, "Portable Network Graphics (PNG) Specification (Second Edition)," 11 2003. [Online]. Available: http://www.w3.org/TR/PNG/.

[14] P. Deutsch, "ZLIB Compressed Data Format Specification version 3.3," [Online]. Available: http://www.ietf.org/rfc/rfc1950.txt.

[15] D. Bongard, "fingerping Github repository," [Online]. Available: https://github.com/0xcite/fingerping.