Music Encoding Through Binary Watermarking

Conner Colligan
School of Computing and Informatics
University of Louisiana at Lafayette
Lafayette, LA, USA
conner.colligan1@louisiana.edu

Martin Margala
School of Computing and Informatics
University of Louisiana at Lafayette
Lafayette, LA, USA
martin.margala@louisiana.edu

***Abstract*—The need to secure one's own intellectual property (IP) within the music industry has become ever more prevalent in the 21st century due to piracy and the illegal redistribution of that property. Digital watermarking technology is becoming a more common method to combat forms of copyright infringement which can allow an artist's work to retain its originality. A method to encode the binary bits of a specified watermark is proposed. Through the use of MATLAB, the watermark image can be broken down into its equivalent of binary bits and inserted into a music/audio file to act as a digital "signature" of the artist. In order to prove the work belongs to the artist, the watermark image can be extracted in a similar method from the file and displayed as proof. Testing shows that this method works well for any audio file using the .WAV format, but further work is needed to make this method of encoding more robust against compression and manipulation of the original file.**

I. Introduction

Due to advancements in digital multimedia within the past few decades, it has become increasingly easier and more convenient to download and/or distribute such content. However, the need for security arises from the increased vulnerability to piracy, unauthorized use, and copyright infringement of an artist's work. Therefore, digital watermarking has emerged as an effective tool to overcome these issues by embedding unique identifiers into multimedia files, providing robust protection for an artist's intellectual property. This paper goes into detail on a custom watermarking method involving the encoding of binary bits into music/audio files in order to protect the owner's intellectual property without affecting the quality of their work.

II. Watermark Encoding Process

A. Audio & Image Import

The method of encoding was done through MATLAB and created in the file called *DrumImageSpliceV3.m* which starts by importing the audio and image files necessary for testing. The audio file used in this experiment is a drum roll recorded in mono (1 audio channel) at a sample rate of 44.1 kHz in the .WAV format called *DrumLoop.wav*. Using audio in the .WAV format allows for better manipulation since it is in the raw format originally recorded in, giving a clearer sound and more amplitude values to work with without compression. In this instance, audio recorded in mono at 44.1 kHz is preferred since most music is output this way and allows a streamline method to watermark most files since MATLAB imports each audio channel as a one column matrix. The image used is taken from a paper titled, *Audio Zero-Watermarking Algorithm based on Beat Tracking*, which is a small 32x32 image called *WatermarkImage.jpg* with distinct letters to be easily identifiable after the extraction process.

In the MATLAB file, the drum loop audio is imported and stored in a matrix variable *y*, then the image is imported into its own variable *i*. Any image imported into MATLAB becomes a three-dimensional matrix for the RGB color values, so it's converted to grayscale then resized to 32x32 to match the image from the previously mentioned paper. The image is now a 32x32 sized matrix with each value representing how black or white that pixel is within a range of 0 to 255. To obtain the binary values of those pixels, the matrix is converted to binary with a function and defined in *i4* which now holds a one column matrix of 8-bit binary values, or bytes of type 'char,' not integer or float values.

B. Encoding & Audio Export

In order to watermark images of different sizes into other audio files with a larger or smaller number of values, variables must be created to hold the elements for both image and audio files. The number of elements contained in each file directly affects how the bits of the image will be imbedded and spaced apart within the audio so as to not create a noticeable disruption in sound quality and fluidity.

The first variable, *i4Rownum* holds the number of total bytes (1,024) within the image, *i4Elnum* holds the number of bits (8,192) within the image, and *yElnum* holds the number of elements or values within the audio file (~580,000). Another variable, *yElnumOffset* is used to offset the placement where bits start to be inserted so that they end up closer to the beginning of the audio rather than the end. Other variables shown like, *byteStep* and *bitStep* define the spacing between each bit and once 8 bits are inserted, a larger space is taken before the next bit is inserted to discern which bits are part of which bytes in the extraction process. Lastly, *nextStep* is initially defined to start at the last value of the audio file with the offset mentioned before, but with each bit insertion will decrease in its placement value until it gets to the beginning of the audio.

A pair of nested *for* loops are used in the encoding process with the first one ranging from 1 to *i4Rownum*, and the second from 1 to 8 which accounts for each bit in a byte, and ultimately every byte from the image, in this case, 1,024 bytes. Since the matrix in *i4* contains the binary values as 'char' values, they need to be converted to integer values to be inserted into the audio file. In the second *for* loop, *bit* is defined to convert one of the bits from a byte in *i4* into an integer, then it's checked whether it is a 1 or 0. If it is a 1, this bit takes the value of the number placed after it in the audio file, but if it's a 0, it takes the value of the number placed before it. The binary values of these bits are changed to ultimately "blend" in with other values because the amplitude of such values in an audio file vary in how it's imported, and in this case, not changing them causes noticeable distortion in the audio quality.

To insert these bits, the matrix *y* is updated by taking every value from the beginning to whatever *nextStep* is, and every value from *nextStep*+1 to the end, then placing *bit* in between them, effectively creating a new matrix with the value of *bit* spliced into it. The value of *nextStep* updates based on the size of *bitStep* and the process repeats 7 more times. On the 8th iteration of this loop, *nextStep* is updated, but this time based on the size of *byteStep* and loops another 8 times before doing it again. For this experiment, the process of splicing bits into the audio file repeats 8,192 times based on the 32x32 image size. Once the *for* loop completes, *y* will now have all of the bits from the watermark image encoded into it, so the function *audiowrite* is used to export it back into an .WAV file called *DrumLoopImageSpliceV3.wav*.

III. Watermark Extraction Process

A. Audio Import & Variable Declaration

When it comes to extracting the watermark image from the newly encoded audio file, a similar, but reversed process from encoding is used to find each bit and recreate the original image. Extraction of the watermark image was conducted in a MATLAB file called *DrumLoopImageExtract*.m. The audio file *DrumLoopImageSpliceV3.wav* is imported the same way by storing it in a variable called *y*.

In order to identify and extract the bits from the audio file, the size of the watermark image must be known in order to give the right number of elements or bits in determining the placement of them, otherwise the original watermark would be needed to find the number of elements. Since we know the image size, *bitTotal* is defined to hold the total number of bits (8,192), *yElnum* now holds the new larger number of elements from the audio file, and the original number of elements is calculated in *origElnum* by subtracting *bitTotal* from *yElnum*. The two variables, *byteStep* and *bitStep* are defined and used in the same way as the encoding process. The offset from the encoding process is redefined as *origElnumOffset* and based on *origElnum* in order to help locate the placement of the first bit inserted which is defined by *nextBitPlace* and is equivalent to *nextStep* in the previous MATLAB file. Lastly, a variable called *z* is predefined as a blank matrix full of zeroes with the purpose of storing the bits that are extracted through each loop iteration.

B. Image Extraction

As mentioned above, the extraction process is similar to how the bits were encoded, so two nested *for* loops are used with the first ranging from 1 to the total bytes (1,024), and the second ranging from 1 to 8. The variable *bit* is defined again by storing the value from *y* given the placement of *nextBitPlace*. A variable named *prevPlace* stores the value of the element before *nextBitPlace* and another variable named *nextPlace* stores the value of the element after *nextBitPlace*. Using these variables, *if* checks are done with *bit*, so if it's equal to *nextPlace*, it's a 1, but if it's equal to *prevPlace* it's a 0. In the case that *bit* is equal to neither, it defaults to 0 in order to output a proper image at the end. After the value of *bit* is determined, it's added to the matrix of *z*.

Like in the encoding process, *nextBitPlace* is updated based on *bitStep* 7 more times until it reaches the next byte, so it updates based on *byteStep*. The process is repeated for however many bits in the image there are, in this case, 8,192 bits. Once all bits are extracted and appended to the matrix in *z*, the function *bin2dec* is used to convert the bits in *z* from type 'char' to integer values, and the new matrix is stored in *z2*. Another variable, *z3*, is defined and used to reshape the matrix of *z2* in order to make it 32x32 so it has the correct format to be output as an image. Finally, the function *imwrite* is used to export the watermark image into a file called *WatermarkImageExtracted.jpg*.

IV. Test Results

The overall results of this project were a success in this given scope but leaves much to be desired in terms of robustness and future integrity. When comparing the newly encoded audio with the original, there is no noticeable difference or change in quality after listening to them except for the encoded audio being approximately 0.1 seconds longer due to the added values from the watermark. As for the extracted image, one can plainly see that it looks the same as the original watermark image but upon further inspection, there are slight differences such as small black spots in the white background as shown in Fig. 1. These differences are a result of bits being mistaken for the wrong binary value which is caused by a slight change in the element values from when the encoded audio file is extracted, then imported back to extract the watermark image. When conducting *if* checks against the *bit* variable, *prevPlace* or *nextPlace* was altered in the original audio extraction process that caused a 1 to become 0, or a 0 to become 1. These errors were relatively negligible given the image was still identifiable, but the real challenge is revealed when the encoded audio file is compressed into a .MP3 file and uncompressed back to a .WAV file.

a) Original b) Extracted

Fig. 1. Extracted Watermark Images (Post Compression)

The testing methods based on the effects of compression were not discussed in this paper due to no viable solution yet that can make this encoding method robust enough. The MATLAB file *DrumLoopImageExtractPostComp.m* contains similar code to the previous extract file that attempts to identify and extract the necessary bits by checking if the previous and next place values fall within a certain range or tolerance of *bit*. The values of the elements in the post compression audio file seemed to change a little or a lot so it was difficult to find a good tolerance range that helped to identify most of the correct bits but couldn't reproduce the watermark image as shown in Fig. 2a. Another MATLAB file named *DrumLoopImageExtractPostCompV2.m*, was another attempt to identify and extract the bits by calculating the difference between *bit* with the next and previous place values, then comparing those differences. If the difference from the previous place value is less than the next place value, that means the previous place value was closer to *bit* and most likely the one it copied, so it's counted as a 1. Otherwise, if the

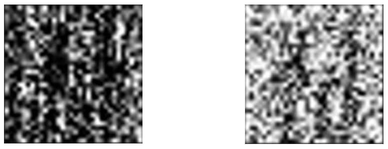

Fig. 2. Extracted Watermark Images (Post Compression)

difference from the next place value is less than the previous place value, the next place value would be closer to *bit* and counted as a 0. This method proved to be slightly better than the first version, but ultimately still couldn't reproduce the watermark image as shown in Fig. 2b.

V. Conclusion

The original goal of this project was to implement an encoding method through MATLAB that could be converted to Verilog code and implemented onto an ASIC capable of watermarking music files. In order for this project to continue, the next step should be to develop a method that builds on the one stated in this paper or create an entirely new one capable of being robust against compression and the changes in values of edited, that would mean the number of elements in the audio file will likely have changed and be missing segments containing bits from the watermark image. The new method must be capable of identifying the encoded bits based on some type of pattern recognition, even when missing multiple bits or bytes from the image. To implement this method, the original watermark image will likely be needed to cross reference itself with whatever bits that are still encoded into the audio file. If this project ever comes to fruition, the potential to bring security of intellectual property to the music industry can truly alleviate the occurrences of copyright infringement, in turn, promoting more original work to be output and possibly influence future innovations in digital security.