**Abstract**

In a digital world, media is easily moved between multiple users. This makes it vulnerable to pirating as well as eavesdropping. Not only this, but data is not easily verifiable. Even though there are other encryption schemes, ne'er do wells dedicated to breaking them will do so with enough time. Not only this, but encryptions can be broken and the files redistributed without the encryption.

We propose a solution to this problem with audio watermarking. By embedding a watermark in each audio file, the original owner can be discerned. Multiple bits are inserted to each audio file that helps distinguish it from every other audio file. This idea follows steganography, the science of writing hidden messages in objects. The existence of these bits is only known to the sender and receiver and therefore eavesdroppers will not even attempt to intercept and decrypt the hidden message.

Hardware implementation for steganography has seen heavy research in the image space, but is lacking in the audio domain. The goal of this project is to implement a real-time audio watermark encoder and decoder in hardware that can embed a secret message of appropriate length into an arbitrary audio input.

**Previous Work**

Natgunanathan et al [2012] proposed a patchwork based algorithm for embedding, detecting, and extracting a watermark into an audio stream. This technique is generic enough to apply to other communication medium that can be represented as an input stream. The patchwork approach will be implemented for this project. Other frequency based techniques, such as spread spectrum, have successfully been implemented in hardware [Karthigaikumar 2010]. The work done by Karthigaikumar wrote the algorithm in a high level language (MATLAB & Simulink) and used a conversion tool to generate VHDL code. Tools such as Xilinx ISE and synopsys were used to increase performance and actually synthesize an FPGA and ASIC implementations. This project will also use Xilinx ISE to generate Verilog code and an optimized FPGA implementation. There has been significant research into real time video watermarking in hardware [Mohanty 2009][Strycker 2000]. Although the algorithm used in this project is slightly different from these image based implementations, the architecture choices should still provide insight on how to optimize an audio watermark encoder.

**Software Implementation**

At a high level this digital watermarking scheme generates a set of common frequencies that represent the audio input. The input signal is broken up into N segments, each of length L. These segments are then split into two subsegments (called the front and rear segments). After these frequencies are split, the Discrete Fourier Transform (DCT) is taken for each subsegment and the DCT coefficients are stored. For each subsegment, the DCT coefficients are split into R groups of coefficients. Each of these R groups have 2 \* M coefficients. These groups of coefficients is called a frame. This is to increase robustness when encoding a watermark bit in a given subsegment. The frames are further broken down into multiple fragments. The fragments are a set of random DCT coefficients from the parent frame. By using the average value of the fragments, if the audio file is modified in any manner, the original watermark information can be retrieved.

Following the division of each segment, a set of values are determined that define the cases for when watermark bits should be embedded. Each watermark bit is defined as a ‘1’ or a ‘0’. The frequencies generated earlier play a large role in determining where the watermark bits can be added. Since any modification can change the perception of the audio, by averaging the different frequencies obtained from the DCT, minimal changes are made to the perception of the audio with the watermark added. The idea is that if the average frequencies across the fragments are “close enough” that modifying the DCT coefficient of the segment will be undetectable. As long as the change is not too large, the perceptible change in the audio should be negligible. This impact of the change is controlled by two parameters (α1 and α2). The exact rules for embedding a watermark bit can be found in [Natgunanathan 2012].

With the watermarks added to the audio input, the Inverse Discrete Fourier Transform is used to convert the audio frequencies back to the time domain for audio playback.

**Synthesized Hardware**

To be synthesized

**Hardware Optimizations**

The goals for hardware optimization are 2-fold. Because real-time encoding and decoding need go only as fast as generating an output to be sampled at 44.1kHz, performance is capped at 352kbps (based on the Shannon-Nyquist Theorem). If this performance target can be met, then the next goal is to minimize area and power while maintaining this performance.

Looking at more common audio bit rates, 192kbps was chosen as the benchmark audio generation value. This implies that every second 192,000 bits are generated for audio sampling. Each sample at 44.1kHz requires 16 bits, providing 192kbps with 12,000 samples/sec. The synthesized hardware can run up to 50MHz and we perceive no issues meeting this goal. If this is met, then higher performance (such as 320 kbps or even 1,411 kbps) targets will be attempted.

With regards to architecture, there is a very clear set of steps in calculating and embedding each watermark. Therefore, this project will benefit greatly from pipelining with multiple smaller stages. Also because multiple calculations can occur in parallel such as the calculation of DCT coefficients for a segment, generation of subsegments and fragments for a given segment, and generation of mean values such as p and q parallelism will greatly increase throughput.

**Experimental Results**

TBD

**Testing Strategy**

A test bench is being devised for the hardware implementation. The test bench will provide registers to push inputs to the encoder as well as registers to store output from the decoder and encoder. In this way determining whether the encoding was successful becomes very simple as the encoder output can be compared to the input or decoder output.

Another test bench framework will be devised to input audio to the encoder system and to play the audio output from the decoder system. This framework will be designed using Port Audio.

**Conclusion**

This project still requires heavy testing on our part, but given our knowledge of the algorithm and the hardware optimizations, the work ahead seems very doable. As we went through implementing the code in C, we found that previous work such as the CORDIC assignment will be very helpful to calculate cosines for the Discrete Fourier Transform. Given the nature of this assignment, creating an audio framework is a must to properly test as well as demo our work. In summary, this assignment will be a great challenge to us.

**References**

Natgunanathan, I.; Yong Xiang; Yue Rong; Wanlei Zhou; Song Guo, "Robust Patchwork-Based Embedding and Decoding Scheme for Digital Audio Watermarking," Audio, Speech, and Language Processing, IEEE Transactions on , vol.20, no.8, pp.2232,2239, Oct. 2012

Karthigaikumar, P.; Baskaran, K.; Kirubavathy, K.J., "Hardware implementation of audio watermarking - covert communication," Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference on , vol.5, no., pp.588,592, 26-28 Feb. 2010

De Strycker, L.; Termont, P.; Vandewege, J.; Haitsma, J.; Kalker, A.; Maes, M.; Depovere, G., "Implementation of a real-time digital watermarking process for broadcast monitoring on a TriMedia VLIW processor," Vision, Image and Signal Processing, IEE Proceedings - , vol.147, no.4, pp.371,376, Aug 2000

**Appendix**

Listing 1: C source code of patchwork algorithm (1st revision, not tested, no value of k)

#include <math.h>

#include <stdlib.h>

#define L (32)

#define LENGTH (1024)

#define NUM\_SEGMENTS (32)

#define NUM\_SEGMENTS\_OVER\_TWO (16)

#define ONE\_OVER\_NUM\_SEGMENTS (0.03125)

#define NUM\_COEFFICIENTS (16)

#define INT\_HI\_MASK (4294901760)

#define INT\_LO\_MASK (65535)

#define L\_OVER\_TWO (16)

#define ONE\_OVER\_L (0.0625)

#define ONE\_OVER\_SQRT\_L (0.25)

#define SQRT\_TWO (1.41421356237)

#define R (4)

#define ONE\_OVER\_R (0.25)

#define M (2) /\* M must be a power of 2 \*/

#define TWO\_M (4)

#define ONE\_OVER\_M (0.5)

#define ALPHA\_ONE (1.41421356237)

#define THRESHOLD (6) /\* TODO: figure out a good value empirically \*/

#define ALPHA\_TWO (0.70710678118)

#define PI (3.14159265358979323846264338327950288419716939937510)

/\* Function for embedding a watermark within a given audio input using the

\* pathwork algorithm decribed in [Natgunanathan 2012]. The paper can be found

\* at http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6198872 \*

\*

\* It is only possible to encode NUM\_SEGMENTS bits per input.

\*/

void embedWatermarK(int\*\* inputStream, int\* watermarK, int watermarKLength, int\*\* &outputStream, int\* &secretKey)

{

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Step 1: DCT coefficient and fragment generation

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

double coefficientsF[NUM\_SEGMENTS][NUM\_COEFFICIENTS];

double coefficientsR[NUM\_SEGMENTS][NUM\_COEFFICIENTS];

double dctMultiplier;

int coefficientF, coefficientR, inputStreamF, inputStreamR;

/\* Generate the DCT coefficients for each segment \*/

for (int segmentIdx = 0; segmentIdx < NUM\_SEGMETNS; ++segmentIdx)

{

coefficientF = 0.0;

coefficientR = 0.0;

for (int coefficientIdx = 0; coefficientIdx < L\_OVER\_TWO; ++sampleIdx)

{

for (int sampleIdx = 0; sampleIdx < L\_OVER\_TWO; ++idx)

{

inputStreamF = inputStream[segmentIdx][sampleIdx];

inputStreamR = inputStream[segmentIdx][sampleIdx + L\_OVER\_TWO];

dctMultiplier = (sampleIdx) ? 2 \* ONE\_OVER\_SQRT\_L : SQRT\_TWO \* ONE\_OVER\_SQRT\_L;

coefficientF += dctMultiplier \* inputStreamF \* cos(PI \* (2 \* sampleIdx + 1) \* coefficientIdx) \* ONE\_OVER\_L;

coefficientR += dctMultiplier \* inputStreamR \* cos(PI \* (2 \* sampleIdx + 1) \* coefficientIdx) \* ONE\_OVER\_L;

}

coefficientsF[segmentIdx][coefficientIdx] = coefficientF;

coefficientsR[segmentIdx][coefficientIdx] = coefficientR;

}

}

/\* Split the DCT coefficients into R frames of length 2M \*/

double framesF[NUM\_SEGMENTS][R][TWO\_M];

double framesR[NUM\_SEGMENTS][R][TWO\_M];

int frameIdx = 0;

int kIdx = 0;

for (int segmentIdx = 0; segmentIdx < NUM\_SEGMENTS; ++segmentIdx)

{

for (int coefficientIdx = 0; coefficientIdx < NUM\_COEFFICIENTS; ++idx)

{

framesF[segmentIdx][frameIdx][kIdx] = coefficientsF[segmentIdx][coefficientIdx];

framesR[segmentIdx][frameIdx][kIdx] = coefficientsR[segmentIdx][coefficientIdx];

kIdx++;

if (kIdx == TWO\_M)

{

kIdx = 0;

frameIdx++;

}

}

}

/\* Split each frame into M fragments using a PN sequence \*/

int pnSequence[TWO\_M];

for (int idx = 0; idx < TWO\_M; ++idx)

{

pnSequence[idx] = idx;

}

/\* Generate random indices for the pnSequence \*/

int tmp;

int rnd;

for (int idx = 0; idx < TWO\_M; ++idx)

{

rnd = rand() % TWO\_M;

tmp = pnSequence[rnd];

pnSequence[rnd] = pnSequence[idx];

pnSequence[idx] = tmp;

}

/\* Assign the PN sequence to the secret key \*/

secretKey = pnSequence;

/\* Generate the fragments for each frame \*/

double fragmentsF1[NUM\_SEGMENTS][R][M]

double fragmentsF2[NUM\_SEGMENTS][R][M];

double fragmentsR1[NUM\_SEGMENTS][R][M];

double fragmentsR2[NUM\_SEGMENTS][R][M];

/\* This array is not needed in the updated code, but I feel liKe it should be Kept around \*/

/\*

double k[R] // TODO: figure out how to reference k

for (int idx = 0; idx < R; ++idx)

{

k[idx] = 0;

}

\*/

for (int segmentIdx = 0; segmentIdx < NUM\_SEGMENTS; ++segmentIdx)

{

for (int frameIdx = 0; frameIdx < R; ++frameIdx)

{

for (int fragmentIdx = 0; fragmentIdx < M; ++fragmentIdx)

{

/\* This method doesn't worK in the updated code, but I feel liKe it should be Kept around \*/

/\*

fragmentsF1[segmentIdx][frameIdx][fragmentIdx] = framesF[k[frameIdx] + pnSequence[fragmentIdx]][frameIdx];

fragmentsF2[segmentIdx][frameIdx][fragmentIdx] = framesF[k[frameIdx] + pnSequence[M + fragmentIdx]][frameIdx];

fragmentsR1[segmentIdx][frameIdx][fragmentIdx] = framesR[k[frameIdx] + pnSequence[fragmentIdx]][frameIdx];

fragmentsR2[segmentIdx][frameIdx][fragmentIdx] = framesR[k[frameIdx] + pnSequence[M + fragmentIdx]][frameIdx];

\*/

/\* TODO: verify this is the correct fragment implementation \*/

fragmentsF1[segmentIdx][frameIdx][fragmentIdx] = framesF[segmentIdx][frameIdx][pnSequence[fragmentIdx]];

fragmentsF2[segmentIdx][frameIdx][fragmentIdx] = framesF[segmentIdx][frameIdx][pnSequence[M + fragmentIdx]];

fragmentsR1[segmentIdx][frameIdx][fragmentIdx] = framesR[segmentIdx][frameIdx][pnSequence[fragmentIdx]];

fragmentsR2[segmentIdx][frameIdx][fragmentIdx] = framesR[segmentIdx][frameIdx][pnSequence[M + fragmentIdx]];

}

}

}

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Step 2 : Select DCT frame pairs for watermarking

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

/\* Compute the expected value for frames across all segments \*/

double p[NUM\_SEGMENTS][R];

double p1[NUM\_SEGMENTS][R];

double p2[NUM\_SEGMENTS][R];

double pTilde[NUM\_SEGMENTS][R];

double q[NUM\_SEGMENTS][R];

double q1[NUM\_SEGMENTS][R];

double q2[NUM\_SEGMENTS][R];

double qTilde[NUM\_SEGMENTS][R];

char useFrame[NUM\_SEGMENTS][R];

double pVal, p1Val, p2Val, pTIldeVale, qVal, q1Val, q2Val, qTildeVal;

char useFrameVal, isSilent;

for (int segmentIdx = 0; segmentIdx < NUM\_SEGMENTS; ++segmentIdx)

{

pVal = 0.0;

p1Val = 0.0;

p2Val = 0.0;

qVal = 0.0;

q1Val = 0.0;

q2Val = 0.0;

for (int frameIdx = 0; frameIdx < R; ++frameIdx)

{

/\* Compute the expected value for fragments across all segments \*/

for (int fragmentIdx = 0; fragmentIdx < M; ++fragmentIdx)

{

p1Val += abs(fragmentsF1[segmentIdx][frameIdx][fragmentIdx]);

p2Val += abs(fragmentsF2[segmentIdx][frameIdx][fragmentIdx]);

q1Val += abs(fragmentsR1[segmentIdx][frameIdx][fragmentIdx]);

q2Val += abs(fragmentsR2[segmentIdx][frameIdx][fragmentIdx]);

}

p1Val \*= ONE\_OVER\_M;

p2Val \*= ONE\_OVER\_M;

q1Val \*= ONE\_OVER\_NUM\_SEGMENTS;

q2Val \*= ONE\_OVER\_NUM\_SEGMENTS;

/\* Compute the expected value of this frame \*/

pVal = 0.5 \* (p1Val + p2Val);

qVal = 0.5 \* (q1Val + q2Val);

/\* Test statistic used to determine if this frame should be used for

watermarKing. \*/

pTildeVal = abs(p1Val - p2Val) - ALPHA\_ONE \* pVal;

qTildeVal = abs(q1Val - q2Val) - ALPHA\_ONE \* qVal;

/\* Boolean indicating whether of not this frame should be used for

watermarKing. \*/

isSilent = (pVal - qVal >=0) ? qVal < THRESHOLD : pVal < THRESHOLD; /\* THRESHOLD will be determined emperically.

isSilent is used to maKe sure we do not

insert sound during a silent portion of

the host audio segment

\*/

useFrameVal = pTildeVal <= 0.0 && qTildeVal <= 0.0 && !isSilent;

/\* Store the values so they can be accessed later \*/

p[segmentIdx][frameIdx] = pVal;

p1[segmentIdx][frameIdx] = p1Val;

p2[segmentIdx][frameIdx] = p2Val;

pTilde[segmentIdx][frameIdx] = pTildeVal;

q[segmentIdx][frameIdx] = qVal;

q1[segmentIdx][frameIdx] = q1Val;

q2[segmentIdx][frameIdx] = q2Val;

qTilde[segmentIdx][frameIdx] = qTildeVal;

useFrame[segmentIdx][frameIdx] = useFrameVal;

}

}

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Step 3 : For every selected frame pair, embed a bit of the watermarK

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

int curWatermarKBit = 0;

double pPrime, p1Prime, p2Prime, qPrime, q1Prime, q2Prime, frameValF, frameValR;

/\* Calculate the modified expected value for each segment \*/

for (int segmentIdx = 0; segmentIdx < NUM\_SEGMENTS && curWatermarKBit < watermarKLength; ++segmentIdx)

{

/\* Each watermarK bit can be encoded in multiple frames, so we encode it in

\* every valid frame pair

\*/

for (int frameIdx = 0; frameIdx < R; ++frameIdx)

{

frameValF = 0.0;

frameValR = 0.0;

if (useFrame[segmentIdx][frameIdx])

{

/\* Encode a '0' bit \*/

if (!watermarK[curWatermarKBit])

{

/\* Check what p1' and p2' should be \*/

if (p1[segmentIdx][frameIdx] - p2[segmentIdx][frameIdx] >= ALPHA\_TWO \* p[segmentIdx][frameIdx])

{

p1Prime = p1[segmentIdx][frameIdx];

p2Prime = p2[segmentIdx][frameIdx];

}

else

{

p1Prime = (1.0 + 0.5 \* ALPHA\_TWO) \* p[segmentIdx][frameIdx];

p2Prime = (1.0 - 0.5 \* ALPHA\_TWO) \* p[segmentIdx][frameIdx];

}

/\* Check what q1' and q2' should be \*/

if (q2[segmentIdx][frameIdx] - q1[segmentIdx][frameIdx] >= ALPHA\_TWO \* q[segmentIdx][frameIdx])

{

q1Prime = q1[segmentIdx][frameIdx];

q2Prime = q2[segmentIdx][frameIdx];

}

else

{

q1Prime = (1.0 - 0.5 \* ALPHA\_TWO) \* q[segmentIdx][frameIdx];

q2Prime = (1.0 + 0.5 \* ALPHA\_TWO) \* q[segmentIdx][frameIdx];

}

}

/\* Encode a '1' bit \*/

else if (watermarK[curWatermarKBit])

{

/\* Check what p1' and p2' should be \*/

if (p2[segmentIdx][frameIdx] - p1[segmentIdx][frameIdx] >= ALPHA\_TWO \* p[segmentIdx][frameIdx])

{

p1Prime = p1[segmentIdx][frameIdx];

p2Prime = p2[segmentIdx][frameIdx];

}

else

{

p1Prime = (1.0 - 0.5 \* ALPHA\_TWO) \* p[segmentIdx][frameIdx];

p2Prime = (1.0 + 0.5 \* ALPHA\_TWO) \* p[segmentIdx][frameIdx];

}

/\* Check what q1' and q2' should be \*/

if (q1[segmentIdx][frameIdx] - q2[segmentIdx][frameIdx] >= ALPHA\_TWO \* q[segmentIdx][frameIdx])

{

q1Prime = q1[segmentIdx][frameIdx];

q2Prime = q2[segmentIdx][frameIdx];

}

else

{

q1Prime = (1.0 + 0.5 \* ALPHA\_TWO) \* q[segmentIdx][frameIdx];

q2Prime = (1.0 - 0.5 \* ALPHA\_TWO) \* q[segmentIdx][frameIdx];

}

}

/\* Update the coefficients stored in the fragment \*/

for (int fragmentIdx = 0; fragmentIdx < M; ++fragmentIdx)

{

fragmentsF1[segmentIdx][frameIdx][fragmentIdx] \*= (p1Prime / p1[segmentIdx][frameIdx]);

fragmentsF2[segmentIdx][frameIdx][fragmentIdx] \*= (p2Prime / p2[segmentIdx][frameIdx]);

fragmentsR1[segmentIdx][frameIdx][fragmentIdx] \*= (q1Prime / q1[segmentIdx][frameIdx]);

fragmentsR2[segmentIdx][frameIdx][fragmentIdx] \*= (q2Prime / q2[segmentIdx][frameIdx]);

}

/\* Update this frame's DCT coefficients \*/

framesF[segmentIdx][frameIdx] = (p1Prime + p2Prime) \* 0.5;

framesR[segmentIdx][frameIdx] = (q1Prime + q2Prime) \* 0.5;

/\* TODO: update the coefficients properly: the paper uses references,

but I don't thinK that's a good idea for hardware \*/

}

}

curWatermarKBit++;

}

/\* TODO: Compute the inverse discre cosine transform \*/

double idctMultiplier, outputStreamF, outputStreamR;

for (int segmentIdx = 0; segmentIdx < NUM\_SEGMENTS; ++segmentIdx)

{

for (int sampleIdx = 0; sampleIdx < L\_OVER\_TWO; ++sampleIdx)

{

for (int coefficientIdx = 0; coefficientIdx < NUM\_COEFFICIENTS; ++coefficientIdx)

{

idctMultiplier = (coefficientIdx) ? 4 \* ONE\_OVER\_SQRT\_L : SQRT\_TWO \* ONE\_OVER\_SQRT\_L;

outputStreamF += idctMultiplier \* coefficientsF[segmentIdx][coefficientIdx] \* cos(PI \* (2 \* segmentIdx + 1) \* coefficientIdx) \* ONE\_OVER\_L;

outputStreamR += idctMultiplier \* coefficientsR[segmentIdx][coefficientIdx] \* cos(PI \* (2 \* segmentIdx + 1) \* coefficientIdx) \* ONE\_OVER\_L;

}

outputStream[segmentIdx][sampleIdx] = outputStreamF;

outputStream[segmentIdx][sampleIdx + L\_OVER\_TWO] = outputStreamR;

}

}

}