Tartan Threads: A Method for the Real-time Digital Recognition of Secure Documents in Ink-Jet Printers

by

Fernando J. Paiz

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 21, 1999

Author_____
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by_____
Walter Bender
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Tartan Threads: A Method for the Real-time Digital Recognition of Secure Documents in Ink Jet Printers

by

Fernando J. Paiz

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 1999

in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Thanks to today's digital imaging technology, any ten year old child with basic computer skills can create convincing counterfeit currency. It comes as no surprise that as output quality and costs have improved in ink-jet printers, there has been a corresponding surge in digital counterfeiting of security documents. The design of a system, through which a printer can recognize a security or other protected document though identification of a watermark, presents a challenge for the application of information hiding techniques. The marking should be strong enough to provide certainty that a document was intentionally marked and robust enough to withstand the transformations inherent in the scanning and printing process. Using an extended spread-spectrum technique, a small one-dimensional thread encoded with a known multi-bit signature is generated. If the printer detects a match, printing halts and a warning message is output to the user. By applying several such threads at varying orientations, this can provide an effective first line of defense against the casual digital counterfeiter.

Thesis Supervisor: Walter Bender
Title: Senior Research Scientist, MIT Media Laboratory

INTRODUCTION

In 1998 the U.S. Treasury estimated that the percentage of US currency in circulation that is counterfeit had grown to over 4 percent from only 1 percent in 1995 [8]. The U.S. Secret Service Counterfeit Division reported that in their 1997 fiscal year, 19 percent of the $40 million worth of counterfeit U.S. currency seized domestically was produced on ink-jet printers [7]. This number represents an 805 percent increase from the percentage of counterfeiters using ink-jet printers in 1995. In the first five months of 1998 over 43 percent of seized currency came from ink-jet devices [7]. These figures are illustrative of a disturbing trend: as digital imaging technology becomes better and more affordable, the problem of casual counterfeiting has correspondingly grown. To make matters worse, similar trends are starting to be witnessed with other security documents (e.g. bank checks, driver's licenses, airline tickets) [9]. It is possible today to buy an ink-jet printer and scanner each capable of resolutions of more than 600dpi for less than $300 combined. With high quality images of world currencies becoming available on the Internet and 1200dpi and higher quality printers heading for the mainstream consumer market in coming years, this problem is only likely to grow.

The U.S. Treasury Department has taken several measures to deal with modern imaging technologies since the 1970s when color photocopiers were first introduced. Specifically, most color photocopiers in the United States today contain circuitry that attempts to recognize if the original to be copied is a bill. Also, as an added precaution, many high quality copiers encode their serial number onto any continuous-tone color image that is printed so that all copies can be traced back to that specific machine. Furthermore, today's more valuable currency notes (20's, 50's and 100's) contain some

additional copy protection in the form of watermarks, embedded plastic strips, very fine print and expensive color-changing inks [11].

Even the addition of these sophisticated precautions has not been able to curb the growth of ink-jet counterfeiting. This is mainly due to the marked difference in the cost of the technology. Since a user does not require an initial investment of tens of thousands of dollars or incur a cost of several dollars per printed page, ink-jet counterfeiters can afford to print less valuable bills (1's, 5's and 10's) which lack the more sophisticated copy protection. These bills can be printed as needed and then spent in locations where it is less likely that their authenticity will be doubted. Other documents worth up to hundreds of dollars, such as airline tickets, checks, stock certificates and tickets to sports or entertainment events are also at risk. The inexpensive technology also means that many more people may be tempted to try their hand at counterfeiting. No longer does counterfeiting require sophisticated knowledge or equipment (offset presses, copy cameras, etc.). Potential counterfeiters are likely to own or have access to scanners and printers already. They may start out of curiosity to see whether they can create a convincing replica of a bill, but if they manage to spend it, they may not want to stop.

Unfortunately, it isn't possible to simply install the systems currently embedded in copiers into consumer ink-jet printers. A device which can reasonably be placed in a copier costing $20,000 to $40,000 could unreasonably affect the cost of a device which retails for under $300. Even if this weren't so, since ink-jet printers print a single line at a time, efficient decoding requires a spatially contiguous encoding which occupies only a small portion of the document. A color copier has the full image of a document to analyze in order to recognize it. An ink-jet printer, on the other hand, can only "see" one

print head width at a time (typically ~0.25"). As a result, any proposed solutions must deal with this limited data space constraint.

We will propose a steganographic system to address this growing problem and its specific constraints. Building upon an existing data-hiding technique, we have designed a system which we believe has an adequate balance of robustness and bandwidth to serve as a first line of defense against the casual counterfeiter. We have created a prototype implementation of the Tartan Thread system and we will present the results of our preliminary evaluation of several effectiveness tests.

PREVIOUS WORK

The Tartan Threads method was first suggested by Gruhl and Bender in [1]. Noting the increased threat posed by the "casual counterfeiter," the authors suggest two data hiding methods which could be applied to help curb this trend: Patch Track and Tartan Threads. Both of these encoding methods are currently being examined for their individual effectiveness. Patch Track is a method which alters the statistics of an image so that a small amount of information can be redundantly encoded over its entire area. Due to its robustness and low perceptibility, this method is suggested as a way of encoding a printer's serial number onto continuous-tone color images as they print. Tartan Threads, on the other hand, is designed to hold more bits of information in a small linearly contiguous space to allow for time-efficient decoding with a high degree of certainty. Gruhl suggested the use of linear Direct Sequence Spread Spectrum as the underlying encoding method, but subsequent innovations as well as results of a fully implemented system are first presented here.

Marvel *et al.* have implemented a blind digital steganography system called SSIS built upon a two dimensional spread spectrum method [3]. Through this method, the authors are able to embed a large amount of data into an image file which can be recovered without any need for the original image file. Through the use of image restoration techniques, an estimate of the original is recreated and subtracted from the encoded document to reveal the encoded information. In order to ensure flawless recovery of embedded data, the SSIS method is combined with Error-Control Coding (ECC). As compared to Tartan Threads, SSIS yields a higher encoding bandwidth and a lower perceptibility. The encoding, however, is not intended to survive a printing and scanning image path or for quick decoding.

Alexander Herrigel *et al.* [4] and Fridrich *et al.* [5] both describe image watermarking methods built upon two-dimensional spread-spectrum techniques combined with a Public Key encryption system for authentication of the part of the author and purchaser of a digital image. Herrigel's technique, like Tartan Threads, encodes several small areas of the image with local identical watermarks for redundancy. Here, however, the protection is intended to survive cropping as all the areas are tiled and encoded in the same orientation. Rotation and scaling transformations are handled through the analysis of these encoded blocks in polar space. By taking a Fourier transform at each block it is then possible to determine what rotation and scaling has been done upon the image and undo it. Fridrich presents both a global and local encoding schemes. In order to provide greater security against attackers trying to destroy the watermark, encoding patterns are generated using a secret key. Since these techniques

involve two-dimensional encoding methods, decoding requires extensive processing times for larger images.

All of these methods, however, focus primarily on the marking of images to be distributed in their digital form. They may resist several lossy image paths, but are not intended to survive the many sampling errors introduced by printing and scanning. The Tartan Threads encoding, on the other hand, is intended for images which will be distributed as printed documents. Our encoding survives even at very low scanning resolutions, and can be decoded efficiently without complicated analysis of the encoded image and using only small contiguous areas of the protected document.

STATEMENT OF PROBLEM

Much of the design of the Tartan Thread method is dictated by the challenges and priorities of the ink-jet counterfeiting problem. First, because we are dealing with printed documents, we must be able to produce robust encodings, even if this results in a lower bandwidth. Any encoding on an actively circulated document such as currency must be detectable even after some standard wear to the original.[1] Since different users with different equipment will potentially be scanning the protected document, the encoding must also survive any non-geometric transformations and lossy image paths that may result from being saved into different image file formats (i.e. compression methods), slight rotations, imperfect color sampling and being re-sampled at different pixel resolutions. Ideally, the encoding should also survive any transformation the user is likely to apply to the digitized image which does not call attention to the human eye.

[1] An interesting area for further work would be a study to create a model for how such documents typically wear over time, as it is unknown if such model publicly exists.

Secondly, because an ink-jet printer renders images line by line and often only has enough memory to buffer a few of these image lines (typically between 16K to a few MB of buffer memory for more sophisticated models [15,16,17]), all decoding of our data must be able to occur efficiently using only a small section of the document. Ideally, a decoder would be created with inexpensive hardware (e.g. a PIC chip), which is capable of searching for encoded information in every print-line image output by the printer. However, any such system must also have extremely low probabilities of false-triggering to prevent the disruption of consumers using their printer for legitimate purposes.

THE UNDERLYING ENCODING

A version of linear Direct Sequence Spread Spectrum (DSSS), was chosen as one encoding method which sufficiently meets the criteria of the problem [12]. Traditional linear DSSS involves creating a carrier wave of length *data rate* at a known frequency and phase, multiplying it by a *chip* signal and adding onto another signal. For the Tartan Threads method, a carrier is created in the brightness plane of an image and summed with a row of *data rate* pixels in the original target image. The phase of the carrier wave in this region is set to 0 or 180 degrees to encode a single bit of information. The *chip* signal is a pseudo-randomly generated sequence with value of -1 or 1 at alternating at a given *chip rate*. Multiplying our carrier sine-wave by the *chip* makes spreads the signal so it looks like random noise on the image

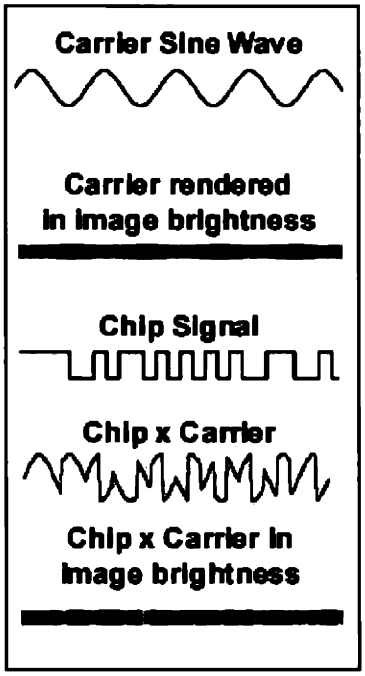

Figure 1. Linear DSSS image encoding creation.

(See Figure 1). Decoding requires the recovery of the phase of the encoding in a given area of the image. First, the brightness values for the encoded part of the image are once again multiplied by an identical chip signal, reproducing the structure of the original carrier wave and making the original image information behave like random noise on it. A Fast Fourier Transform is then taken to check the phase at the carrier frequency. This technique can be used to hide information into digital images with high bandwidth (1 bit for every one dimensional sequence of 8 to 16 pixels, or about 2Kbytes in a 512 x 512 image) when using a *chip rate* of 1 chip per pixel. However, to successfully recover that encoding, one must be able to accurately sample pixel by pixel the original encoded area. While it provides a good encoding density, it lacks the resistance to alignment errors required for steganography in printed documents.

We must trade some of the bandwidth potential of the linear DSSS to provide sufficient robustness in the Tartan Threads encoding. Using a higher *chip rate*, allows for alignment errors in the horizontal axis. Furthermore, an identical carrier signal encoding is repeated for every *chip rate* lines for vertical alignment errors. This creates a simple two-dimensional extension of our one dimensional encoding. Decoding is performed, by sampling once every *chip rate* by *chip rate* pixels. In effect, this modification creates a low resolution encoding which can be overlaid on a high resolution image. The decoding process is now

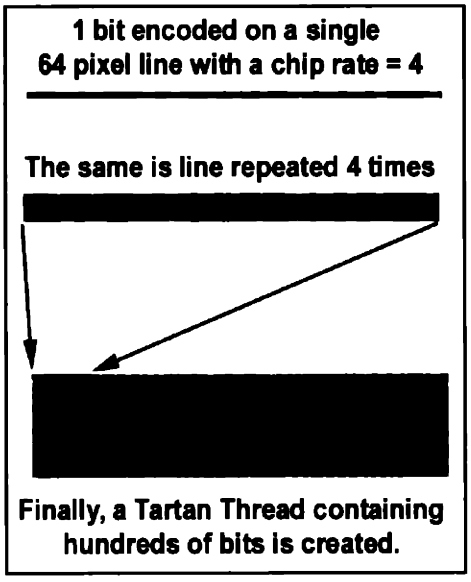

Figure 2. Multi-bit Tartan Thread encoding creation.

tolerant to the slight alignment errors as well as small rotational variances often introduced in scanning and printing. Also, it becomes resistant to resampling, since the encoding exists at a low enough resolution that any sampling likely to create a convincing image of the original must also capture the encoding detail. This is a necessity as printed documents will be digitized with unknown parameters (e.g. dpi, orientation and smoothing filters).

In addition to lower bit density, a higher *chip rate* also has the drawback of higher visibility. Research has shown that the human perceptual system has trouble perceiving noise present in high frequency areas of images it receives [13]. Therefore, in order to make our encoding as unnoticeable as possible, we rely on it having the characteristics of high frequency noise. Higher *chip rates*, however, achieve less spreading of the carrier signal since the carrier signal remains intact for *chip rate* pixels and is repeated for *chip rate* lines. This results in the lowering of the frequency of the noise added to the image. Furthermore, since a certain number of samples is required to accurately sample a sine wave, higher *chip rates* also require lower frequency carrier waves for any given *data rate*. Final encoding parameters need to balance robustness, bandwidth and visibility considerations.

THE VISIBILTY MASK

To prevent the encoding from becoming overly noticeable, a visibility mask of the image is created and used to scale the signal amplitude. Since in the human visual system allows for an encoded signal to be masked by the presence of other high frequency signals, effective data hiding requires the identification of high frequency areas

in the image. Vision scientist have researched a number of contrast sensitivity functions which attempt to measure points in an image where changes in luminance become visible to human observers [13]. The visibility mask is provides an estimate of contrast sensitivity by plotting the relative amounts of high frequency activity existing in each area of the picture. Visbility mask creation involves subtracting low-frequency values from the image and then re-scaling the result from zero to one (See Figure 2). When the encoding is applied to the image the amplitude of the signal is scaled by this visibility mask value. In this way we make the encoding as strong as we can in each part of an image without making it too noticeable.

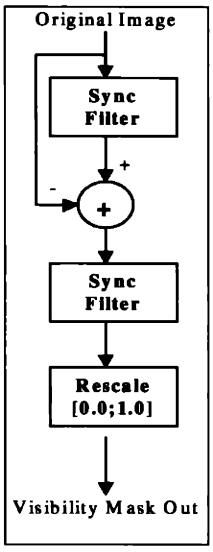

Figure 3. Visibility Mask Creation Flow Chart

Once the Thread creation application was written, the visibility mask was further optimized. In order to improve the decoding of individual bits, the visibility mask was made to use one visibility value for every data rate pixels. This, however, created bleed over from areas with dense printing into nearby sparsely printed areas, not fit for high amplitude encoding. Since this was only noticeable in these cases, the visibility mask was made to use individual values if the bit-encoding area contained any areas with visibility less than or equal to .1 (in a scale from 0 to 1 of high-frequency activity). The data-rate average value was used for all other bits. Finally, since scaling amplitude past a certain point, no longer helps the encoding, but makes the encoding more perceivable, a maximum amplitude value was set past which the carrier wave could not be scaled. This was typically set to 70, with an original carrier amplitude of 140 (on a scale from 0 to 255).

ONE TARTAN THREAD

Decoding the modified DSSS signal is a challenge due to the distortions introduced by the scaling of the visibility mask. There are always areas of the image where the encoding would be visible if encoded with high amplitude. Adding further redundancy to each bit by encoding it over a larger area of the image implies further sacrificing bandwidth. Furthermore, this kind of redundancy is less effective with linear DSSS than other encoding methods due to its spatially contiguous nature. That is to say, if one area of the image is unsuitable for encoding it is also likely that the area immediately surrounding its shares similar characteristics. All of these limitations mean that we can not expect lossless recovery of any bit signature encoded into an image one hundred percent of the time. However, by encoding sufficient bits, it is still possible to identify an image as being marked with very high probability. To this end we performed extensive characterizations to find an optimal balance of encoding parameters for the Tartan Threads method.

There are five parameters which define our Tartan Threads encoding. The first is the *data rate*, which is the number of pixels in one line of the image we use to encode a single bit. For ease of implementation, the *data rate* is always chosen to be a power of two. Next is the *amplitude* of the carrier wave and its frequency, Θ, which in our experiments is measured in number of full cycles per *data rate*. The *chip rate*, as described above, adds robustness by repeating the encoding in *chip rate* by *chip rate* squares. Finally, the spatial frequency of the encoding in the printed document, (essentially the *dpi* at which the image was encoded), must be known in order to

successfully decode it. All of these parameters must be permanently set for decoding, with the exception of the *amplitude* which can vary depending on the specific target image characteristics.

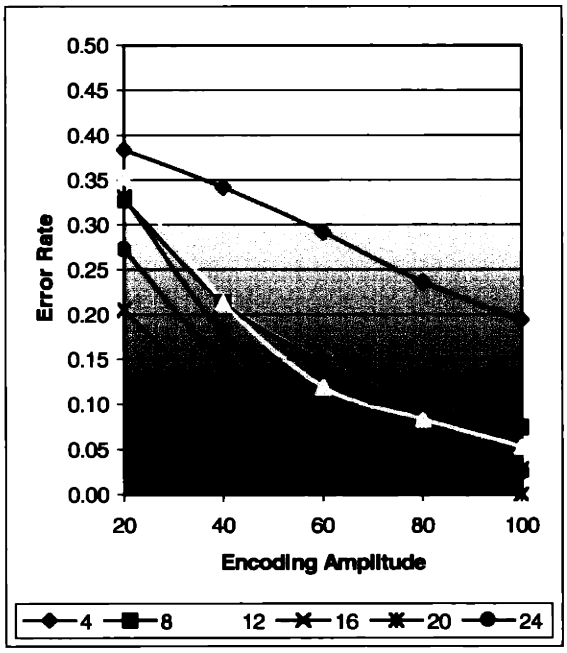

Figure 4. Characterization response for varying theta values. Chip Rate = 4; Data Rate = 128.

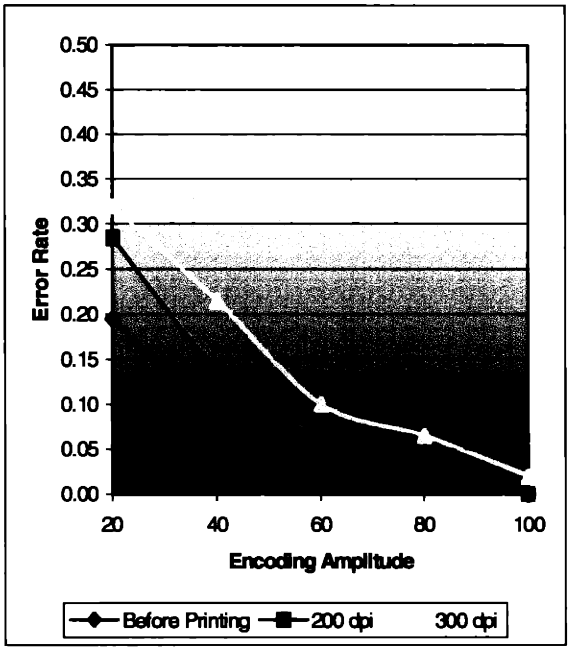

Figure 5. Characterization response, before printing and after 200 & 300 dpi scans.

In order to ensure robustness to resampling and lossy image file formats, we chose a low spatial resolution of 200 dpi for the Tartan threads encoding. 200 dpi also provided better results after printing and scanning (see Figure 5). Our experiments show that when encoding at this resolution, a *chip rate* of 4 is adequate for accurate sampling. The choice of higher *chip rate* values is not necessary, since sufficient robustness is achieved and higher values result in higher error rates and increased visibility. Since, carrier waves require two samples per cycle for accurate rendering, Θ was chosen simply as one half the number of *chips* in one *data rate* (see Figure 4). Choosing an appropriate *data rate*, involves balancing the need for accurately decoding each bit with the overall amount of certainty provided by the total Thread encoding. Encoding space at these low

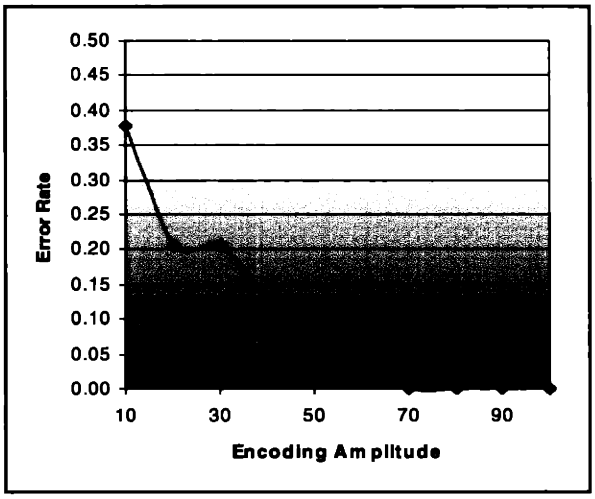

Figure 6. Characterization response with final encoding parameters on a blank image.

Table 1. Error rates for varying chip rates and data rates with maximized Theta value and # of samples. Encoding amplitude = 40.

Varying Chip Rate - Data Rate = 256			
Samples	Chip rate	Theta	Error Rate
128	2	64	4.58%
64	4	32	5.00%
32	8	16	11.67%
16	16	8	16.67%
8	32	4	20.00%
Varying Data Rate - Chip rate = 4			
Samples	Data rate	Theta	Error Rate
128	512	64	2.00%
64	256	32	50.00%
32	128	16	7.50%
16	64	8	15.25%
8	32	4	21.38%

resolutions is limited and very high certainty is desired. Higher *data rates* decode with fewer errors (since they have room for more sampling of the carrier wave), but take up exponentially increasing amounts of space. Table 1 shows error responses for varying *data rates* and *chip rates*. Using a 2.24" x .5" space for each thread, a *data rate* of 64 pixels was chosen. Even though bit error rates of 15% or more are common in the encoded images, the low *data rate* allows for 175 bits of information to be encoded per Thread and thus provides a high level of certainty of identification.

How many bits is enough to provide adequate identification certainty? That question is the

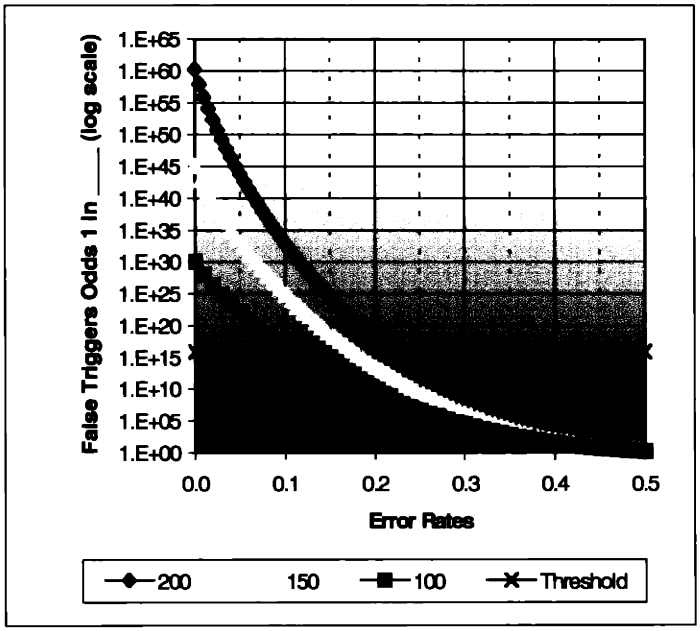

Figure 7. Tartan Thread false triggering odds per number of bits

overarching consideration in designing the encoding parameters. If this system is to be installed in all ink-jet printers, it is important that it behaves in a way where it does not prevent consumers from using their printers for a legitimate purpose. The probability of false-triggering occurring should be infinitesimal. Assuming that an unencoded image is equally likely to decode a 1 or 0 in any bit position (an assumption that is made all the more reasonable by the fact that we multiply by a pseudo-random chip signal in decoding), we can then analyze false triggering as a series of Bernoulli trials with probability (P) of .5 [14]. When attempting to decode an n-bit thread in an unencoded image, the probability of decoding exactly k_0 bits correctly is $P_k(k_0)=\binom{n}{k_0}P^{k_0}(1-P)^{n-k_0}=\binom{n}{k_0}\left(\frac{1}{2}\right)^n$. The probability of false-triggering would occur with a threshold of k_0 equals the probability of decoding k_0 or more bits correctly, which is $P_{k\geq k0}(k_0)=\sum_{j=k_0}^{n}\binom{n}{j}\left(\frac{1}{2}\right)^n$. Figure 3 shows the odds of false triggering for a given accepted error rate tolerance for a single Thread with 200, 150 or 100 bits of encoding. The threshold of for acceptable false triggering probability of approximately 1 in 10^{15}, is derived by calculating the number of placement possibilities of the Thread on an ink-jet printer and keeping the expected probability of triggering in the page to below 1 in one billion (see Table 2). With a 175 bit Tartan Thread, a 20.0% error rate (35 bit errors) or less is sufficient to show reasonable mathematical certainty that the image is marked.

Table 2. Necessary odds of false triggering per thread.

Maximum Printable Area			
8.3	x	10.8	89.64
by number of pixels (dpi)			
200	x	200	3585600
Desired odds of false-trigger			
1 in		1E+09	
Necessary threshold			
triggering odds per thread =			3.5856E+15

180 DEGREE ROTATION

Since efficiency of decoding is a crucial aspect of the Tartan Threads method, the encoding was designed to be decoded with the same procedure at either 0 or 180 degrees from the horizontal. This was accomplished by forcing a symmetry in the chip signal. For every line in the upper or lower half of a thread, the corresponding line that is equidistant from the midpoint in the other half has a chip signal generated from an identical seed that is placed in reverse order. If there is an odd number of encoded lines, the center line chip signal is made to be symmetric from either side. Naturally, this requires that the data being encoded in each bit be symmetric around the middle bit as well. Since rotation adds a phase shift, decoding for threads at 180 degree rotation, simply involves checking for abnormally high error rates as well as abnormally low ones.

MULTIPLE THREADS – ROTATION RESISTANCE

Because of the limited image area available in an ink-jet printer, Tartan Threads as described above, can only be decoded in a limited range of orientations. In order to be decoded, the thread has to fit entirely in a space designated to buffer a few print lines. Since the Threads are about 2 inches long, any small can force the sampling to misalign (see Figure 8). In fact, without adding more complicated orientation searches to the decoder (which would require extreme optimization and/or more complicated hardware), the only orientations which can be decoded are those threads aligned near 0 and +/- 180 degrees from the print line horizontal. In order to trigger with counterfeiters printing at other orientations, multiple threads are imbedded throughout a protected image.

How many threads should we place in a document in order to hope to get an optimal level of protection? And how effective can we hope for this protection to be? As noted above, our modified linear DSSS technique has some built in robustness for rotation variances. It is important to note, however that this is on a very small scale. If the *chip rate* is 4 with a 2" wide Thread on a 200 dpi image for example, then the our DSSS can decode with a 4 pixel offset in that 2 inch stretch of pixels. That amounts to only a .02" offset, or 1% of the Thread length. That is the equivalent of only slightly more than .5 degrees of tolerance in either direction or a range only 1 degree wide. That is enough to cover most alignment errors on a scanner, but not enough to resist a deliberate rotation. This would imply that to completely protect an image, we would require 180 Threads! Needless to say, with the size of our current threads, there isn't enough space to place that many threads on any document at varying orientations.

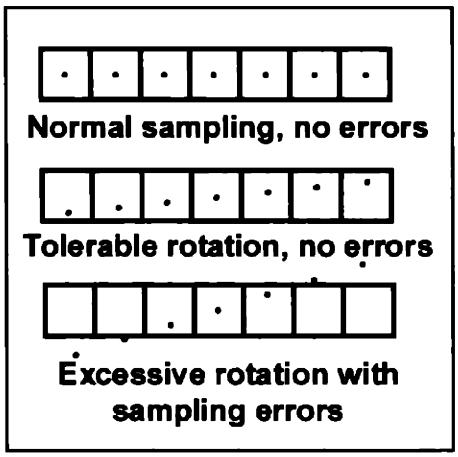

Figure 8. Encoding survives rotation with no errors while sampling points remain within *chip rate* x *chip rate* squares for the length of the thread.

In practice the rotation tolerance we observed was closer to one degree in either direction for a total range of two degrees. The reason for this increased resistance is that our encoding is strong enough to trigger below our specified error rate threshold even with some additional errors. However, to maximize rotation resistance, several modifications to the *chip signal* generator were attempted to help increase the likelihood of *chip* alignment even with some rotation. One attempt used an identical chip signal for all lines in a thread. This provided for much improved rotation handling up to 3.5

Table 3. Resulting Error Rates for Rotation Resistance Tests using Three Different Chip Signals

Degrees of rotation	Standard Chip	Chip Repeated For line pairs	Chip Repeated For All Lines
0.0	2.29%	2.29%	0%
0.5	5.71%	4%	0%
1.0	21.14%	15.43%	0%
1.5	30.85%	23.42%	2.29%
2.0	38.86%	31.43%	8%
2.5	38.86%	38.29%	12.57%
3.0	41.71%	38.29%	15.43%
3.5	42.29%	41.14%	18.29%
4.0	42.86%	40.57%	20.57%
4.5	40.57%	38.86%	22.28%

degrees in each direction. Unfortunately, this also made threads more visible since they now resembled contiguous streaks which were being added onto the image. As a compromise, each chip line was set to repeat twice, resulting in rotation tolerance up to 1.5 degrees in either direction (See Table 3). The total rotation range covered by a single Thread, including 180 degree rotations, is approximately 6 degrees.

Even with this improvement, it is still impossible to completely protect a document using a simple linear decoding search. To guarantee against false triggering and ensure robustness, the physical size of each thread ended up being larger than initially intended. This compounded with sparse printing in many of our target documents (currency, for example), leads to limited placement options. Also, due to the small amount of encoding space used for each bit, high amplitude signals are typically needed for the encoding to survive being embedded into an existing image. This means that Threads can't overlap without disrupting one another and rendering their intersection plainly visible. For these reasons the number of Tartan Threads that can be placed onto typical security documents is limited. However, it is still easy and viable to protect reproduction of larger documents and provide warnings to users attempting to reproduce documents in one of the standard portrait or landscape orientations.

Like all current watermarking methods, Tartan Threads has its weaknesses. Rather than being an unbeatable lock, therefore, a watermarking method need only

provide a first line of defense and a warning to users attempting to replicate protected documents of the illegality and potential penalties of their actions. Ideally, if a Patchwork serial number marking method was also implemented in the printer, a warning could inform the user that any attempted prints would be traceable back to them. In a sense, the idea here is to protect people from themselves, and prevent them from claiming ignorance. For this task, a small number of Tartan Threads at varying angles (definitely including standard portrait and landscape orientations) is appropriate. Also, for two-sided security documents, Thread placement on either side can cover different ranges of angles, since potential counterfeiters must align both sides to create convincing copies. Finally, the level of protection offered by a few Tartan Threads on an image could be increased over time. If a first generation of today's printers was only required to search for Threads in the horizontal direction, it would not be unreasonable to expect that in 4 or 5 years microprocessors and memory technologies will have progressed to the point where printers were could search larger areas of an image for threads within a 30 degree range of the horizontal in the same amount of time and with hardware that presents no greater cost to the consumer. Thus, in time the same decoding could be searched for more carefully, making it effectively cover all orientations. Furthermore, with greater care taken in the design of securities more Threads could be placed in documents and with reduced error rates.

IMPLEMENTATION

For our final implementation we have created a C++ program which reads grayscale PGM images analyzes them and attempts to optimally place a given number of

Threads. A generic thread encoding is calculated and stored in memory. The next step is to calculate the visibility mask. Then, a Monte Carlo sampling approach tests potential Thread locations for each desired angle from the horizontal. The Monte Carlo approach is chosen for its simplicity and time efficient running. In our tests placing 2.24" X .5" Threads in a 200dpi scan of a US Dollar bill, target locations are converged to optimal areas within 50,000 random trials. That is to say, the same area was consistently chosen with several different random seeds. Testing for six threads with these parameters was completed within a two minutes on an Intel Pentium Pro 200 machine running Linux. As Thread positions are chosen, the visibility mask is modified so that the area of the placed Thread has a visibility of zero at all points. This prevents subsequent threads from overlapping.

The decoding application currently remains unoptimized. Quite simply, every possible position of an area the size of a Thread is decoded in a linear sequence. Initially, thread positions are sampled every *chip rate* pixels. However, if an error rate less than 35% or greater than 65% is found, a search of all surrounding pixels is initiated. If one is sufficiently close to our target encoding (within 20%), a response is triggered. Decoding a given area involves generating the corresponding chip signal for each line of a Thread. Multiplying image brightness values by this signal and performing an FFT for each *data rate* to check the phase. Searching an entire page for threads could take several minutes, but a small amount of image analysis to eliminate areas with no content can reduce this search time to under one minute for small securities such as single currency notes. Specialized hardware capable of performing FFT's several times faster, should allow this

simple decoding to occur in less time than it takes to print a page on an ink-jet printer (typically 5-15 seconds).

RESULTS

To test the encoding, Tartan Threads were embedded in three different documents: a "Bender Buck" (fictitious Media Lab currency), an airline ticket and a scanned image of a plain text document. The bill was encoded with 2 threads at the

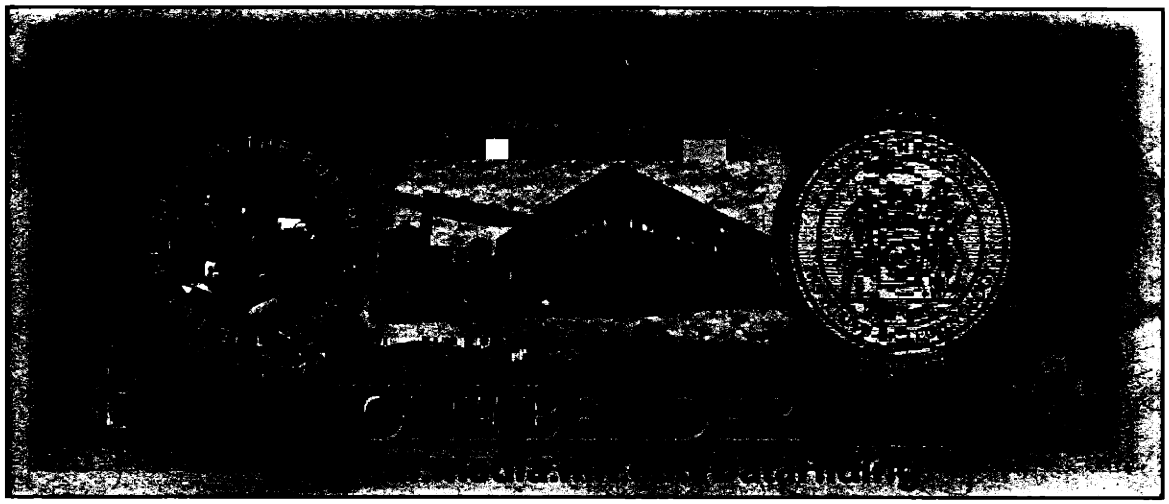

Figure 9. A Bender Buck encoded protected with 2 Tartan Threads

in digital counterfeiting of security documents. The design of a system, through which a printer can recognize a security or other protected document, presents an interesting challenge for the application of information hiding techniques. We propose a method for the marking of printed security documents which allows for their real-time recognition using inexpensive hardware which could be embedded in consumer ink-jet printers and is robust enough to withstand the transformations inherent in the scanning and printing process. Using an extended spread spectrum technique, a small one-dimensional thread encoded with a known multi-bit signature is generated. If the printer detects a match to the signature, printing of the image halts and a warning message is output to the user. By applying several such threads at varying orientations throughout the target image, this can provide an effective first line of defense against the casual digital counterfeiter.

Figure 10. Scanned text with one Tartan Thread applied

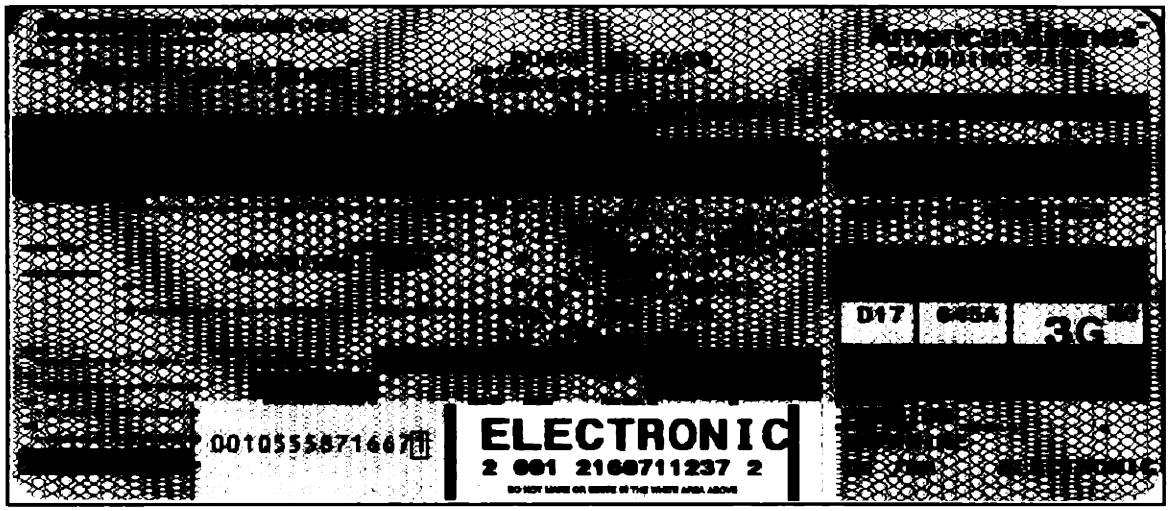

Figure 11. An airline ticket protected with one Tartan Thread

portrait and landscape orientations (see Figure 9). The landscape thread (in the lower left below the NIF seal), was better hidden due to the fact that is embedded in an area with a denser and darker etching pattern. The portrait thread (in the MIT seal) calls attention to itself somewhat in the areas with the least print density. Similarly, the airline ticket was only imbedded in the landscape orientation, because there wasn't a part of the pattern that clustered in the vertical direction (Figure 11). This is a perfect example of a document where a simple pattern could be devised for the background so it was more conducive to the Tartan Threads encoding. Both of these documents, were easily detected in our scanning tests with error rates below 15%. The text document (Figure 10) with its relatively sparse printing, however, could not be encoded strongly enough to be securely marked. Surprisingly, though, it came close. Before printing and scanning an error rate of 24% was found. After scanning, the error rate was 26%. Since this is

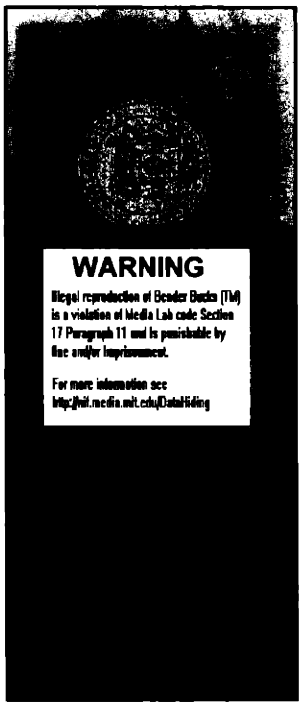

Figure 12. Sample trigger response output

so close to our threshold of 20%, it might be feasible to encode cetrain text documents with tartan threads if desired, provided that the texts was laid out with very tight clustering. Figure 12 shows the output of our prototype identification triggering.

Table 4. Results from image resampling tests on an encoded bill.

Scanned dpi	Error Rate
300	14.86%
200	14.86%
100	16.50%
72	20.00%
50	18.28%
37	20.57%
25	27.43%

The encoding performed well in all robustness tests. For a resampling test we scanned a printed version of a marked bill at varying resolutions. The scans where subsequently resized to the 200 dpi resolution and decoded. Table 3 shows the resulting error rates. The encoding survives successfully even at very low resolutions. Including 72 dpi (standard postscript resolution) and 50 dpi (the effective chip resolution). We believe this to be an adequate level of protection since lower resolutions will result in images which are pixilated or blurred enough to call attention to themselves as counterfeit reproductions.

The next test was the jpeg encoding test. The Joint Pictures Experts Group encoding is a common lossy perceptual encoding method for image files. When saving a user can set a desired quality level, which results in a corresponding image rendering precision and compression ratio. In our tests, the encoding survived being saved at quality levels from 100% to 1%. Figures 13 and 14 show

Table 5. Results from Jpeg encoding resistance tests on a marked bill.

Jpeg Quality	Error Rate
100%	14.86%
75%	17.71%
50%	15.43%
25%	16%
1%	19.40%

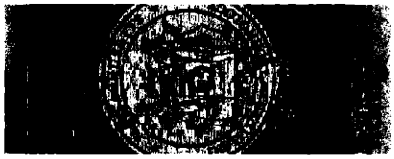

Figure 13. JPG of encoded area of a marked bill at 100% quality

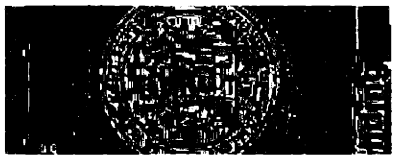

Figure 14. JPG of encoded area of a marked bill at 1% quality

the corresponding encoded sections of each image. Table 4 shows the observed error rates.

One type of transformation that Tartan Threads does not survive through very well is geometric scaling of the protected document. We assume that counterfeiters will try to make their copies of identical size to the original. However, it is possible that they may try to defeat the copy protection by adding small scaling differences. Table 5 shows decoded error rates with a clean gray encoded thread at slightly varied scaling factors. A change in scale of more than 1% pushes the error rate beyond the triggering threshold. Here, again, the decoding routine could search for Tartan Threads at varied scales. However, because even a small amount of scaling can dramatically affect error rates, the decoding routine would have to either happen a few orders of magnitude faster, or several processor chips would have to be employed to search different orientations in parallel. Again, as microchip technology continues to improve and become more affordable this kind of search could easily be implemented to search at scale factors within 10 percent or less of the original.

Table 6. Results from scaling response tests on a marked bill.

Scale	Error Rate
98.50%	38.86%
99%	29.71%
99.50%	19.42%
100%	12.57%
100.50%	17.71%
101%	26.28%
101.50%	37.71%

FUTURE WORK

One area which remains to be fully explored is the optimization and hardware implementation of the decoding system. A prototype for such a system must meet a number of important requirements. Firstly, the implementation must require hardware costing no more than a few dollars. Secondly, decoding must happen quickly enough so

as to not slowdown the printing process. As printers become more and more efficient, keeping up presents an increasing challenge. Furthermore, it is always advantageous to decode more quickly since any extra time could be spent by the printer searching for threads at skewed orientations or slight scale factors. One reasonable approach would be to take areas suspected of containing an encoding (i.e. those whose decoded error rate is above a certain threshold), and search threads within 10 or 15 degrees of the horizontal and within a 5% scaling in any direction. This type of decoding search would allow for complete protection of a document with only 6 Tartan Threads.

Another major limiting factor in the current implementation is that Threads are being added to an existing image which was not intended to hold them. The front of current U.S. Dollar bills, for example, have many areas with sparse printing, which limits the area available for encoding. In the future, it would be possible to design security documents around the fact that they must contain the patterns of several Tartan Threads in several orientations. A computer program could be written to analyze Thread Positions in a given area and create a background "etching" pattern which retains the same structure within its design. This would address two important problems in the current system. Firstly, since the background itself would contain all the encoding, consumers would be unable to identify the Tartan Thread locations even with careful scrutiny. Attacking the watermark, would therefore be that much more difficult. Also, many more threads could be fit into the document area to cover more decoding orientations, with considerations so that they don't interfere with one another. Second, since there would be less noise sharing in the image space, error rates should be considerably lower. Correspondingly, the encoding would be more robust to any kind of

transform it should endure. Although, watermarking is typically approached as the embedding of hidden information into an existing document that does not have to be the paradigm we use for security documents. It would be easy for the parties interested in keeping these documents secure (national treasuries, banks, airlines, etc.) to reverse-engineer the designs to provide a truly secure designs.

Finally, another area to explore which might yield interesting results is the creation of radially symmetrical encoding patches. The benefit of such a system, is obviously to address Tartan Thread's susceptibility to being undermined by rotation. A system which was retained the robustness to other transforms, and could be decoded in any orientation would be a valuable tool in marking printed documents. It might be possible to implement such a system by creating a radially symmetrical patch which merely contains a pointer to another area of the image. This would allow relatively few bits to be stored in the patch, while still providing a way to decode enough information to have mathematical certainty that a document is marked.

CONCLUSION

Protecting copyrighted materials and security documents is a growing concern that needs to be addressed soon. Digital imaging technology will only continue to improve and become more affordable. While several watermarking methods for copyright protection are currently being researched, they are not being tailored to the needs of protecting security documents. Their aims are more in tracking distribution of a copyrighted image and not preventing its reproduction. With printed security documents, their identification before they are counterfeited, could help the government win a fight it

is currently losing. This type of identification requires a method that is robust, provides a high level of certainty and can be decoded inexpensively in hardware with a limited viewable area of a document. Tartan Threads aims to provide this functionality, using an adapted version of a common information hiding technique.

A full implementation of the Tartan Threads method has been presented. The Tartan Threads encoding provides a robust and certain watermarking method that can be detected quickly and inexpensively. The watermark easily survives non-geometric transforms, resampling and lossy image paths it is likely to be subject to when the printed document is digitized. Although the initial results show several limitations of the encoding, these could be addressed by improving the designs and print quality of the original documents and by optimizing decoding procedures to allow for a search of a greater space of translations the encoding may have suffered. With further development of some supporting technologies, Tartan Threads could provide an important first line of defense against the casual counterfeiter.

BIBLIOGRAPHY

1. D. Gruhl, and W. Bender, "Information Hiding to Foil the Casual Counterfeiter," *Information Hiding: Second International Workshop* (1998).
2. W. Bender, D. Gruhl, N. Morimoto and A. Lu, "Techniques for Datahiding," *IBM Systems Journal 35 3 & 4* (1996).
3. L. M. Marvel, C. G. Boncelet, and C. T. Retter, "Reliable Blind Information Hiding for Images," *Information Hiding: Second International Workshop* (1998).
4. A. Herrigel et al. "Secure Copyright Protection Techniques for Digital Images," *Information Hiding: Second International Workshop* (1998).
5. J. Frisrich, "Robust Digital Watermarking Based on Key-Dependent Basis Functions," *Information Hiding: Second International Workshop* (1998).
6. W. Bender, D. Gruhl, and N. Morimoto, *Method and Apparatus for Data Hiding in Images,* U.S. Patent No. 5,689,587 (1996).
7. "Ink-jet Counterfeiting on the Rise," *Reuters*, http://www.zdnet.com/zdnn/content/reut/0401/ 302907.html (April 1, 1998).

8. M. Kotadia, "US in Counterfeit Crisis," *ZDNet UK*, http://www.zdnet.co.uk/news/news1/ns-3952.html (March 17, 1998).
9. S. Silverthorne, "Counterfeit Computing," *ZDTV*, http://www.zdnet.com/zdtv/cybercrime/features/story/0,3700,2000033,00.html (1996)
10. "Genuine or Counterfeit?," *Federal Reserve Bank of Atlanta*, http://www.frbatlanta.org/ publica/brochure/counter/counterf.htm (1996).
11. "Your Money Matters," *U.S. Treasury*, http://www.ustreas.gov/currency/hundred.html
12. M. K. Simon et al, *Spread Spectrum Communications Handbook*. McGraw-Hill, New York (1994).
13. A. N. Netravali and B. G. Haskell, *Digital Pictures: Representation, Compression, and Standards (Applications of Communications Theory)*. Plenum Publishing Corp, New York (1995).
14. A. Drake, *Fundamentals of Applied Probability Theory*. McGraw-Hill, New York (1967)
15. "HP Personal Printers Page," *Hewlett Packard Inc.*, http://www.pandi.hp.com/pandi-db/dds_product_list.show2?p_prod_catgy_id=1&p_prod_type_id=6&p_family=PersonalPrinters
16. "Epson Printer Products," *Epson Inc.*, http://www.epson.com/printer/
17. "Color Bubble Jet Printers," *Canon Computer Systems, Inc.*, http://www.ccsi.canon.com/goto.shtml?/bjc/index.html

ACKNOWLEDGEMENTS

I would like to thank Walter Bender for the opportunity to participate in the data hiding project and for his guidance and support.

Likewise, Daniel Gruhl has been an important source of knowledge, innovation, humor and support in the development of this technique.

Raymond Hwang collaborated in the development of early image processing and input/output function and developed the code for image rotation.

Finally, I would like Jessica Yeh and Walter Holland for their contributions to characterization experiments.

APPENDIX – C++ CODE

settings.h

```
#define DRATE 64   /* data rate */
#define CRATE 4     /* chip (pseudo-random number) rate */
#define THETA 8.0
#define FREQ THETA*M_PI*2/DRATE
#define AMP 40
#define BLUR1 3
#define BLUR2 5
```

util.hh

```
struct pgm{
  long columns;
  long rows;
  double maxdepth;
  double * data;
};


//RAY's ROTATION CODE

double* makeshift(double xyshift[2]);

double* makerot(double angle);

double* mulmatmat(double* b, double* a);

double* mulmatvec(double* b, double* a);

double* invmtx(double* T);


int index(pgm* image, int x, int y);


void transform(pgm* image, pgm* output, double* I);



//TARTAN THREADS PEICES

double* makeChip(const unsigned long length, const unsigned long rate,
const int seed);
void makeChip2(double * chip, const unsigned long length, const
unsigned long rate, const int seed, const int line, const int h);
double* makeData(const unsigned long length, const unsigned long rate,
int* data);
double* makeCarr(const unsigned long length, const double freq);

double* makeSignal(unsigned long length, const unsigned long rate,
double* data);
void makeSignal2(double* Sig,unsigned long length, const unsigned long
rate, double* data);
```

util.cc

```
#include "util.hh"
#include <iostream.h>
extern "C" {
#include <stdlib.h>
#include <math.h>
}

double*
makeChip(const unsigned long length, const unsigned long rate, const
int seed){
  srand(seed);

  double* rval = new double[length];
  for (long i=0; i<length; i++){
    if (i % rate == 0)
      rval[i] = (double) (2 * (rand() % 2) - 1);
    //rval[i] = rand() % 2;
    else
      rval[i] = rval[i-1];
  }

  return rval;
}


//assumes existing chip array of size length
//Makes symetric for 180 degree rotation encoding....
//makes own seed

void makeChip2(double * chip, const unsigned long length, const
unsigned long rate, const int seed, const int line, const int h){

  if ((h % 2 == 1)&&(line== (h+1)/2)) {
    if (line % 2 ==1)
      srand(seed+line-2);
    else
      srand(seed+line-1);
    for (long i=0; i<length/2; i++)
      if (i % rate == 0){
      chip[i] = (double) (2 * (rand() % 2) - 1);
      chip[length-i-1] = chip[i];
      } else {
      chip[i] = chip[i-1];
      chip[length-i-1] = chip[length-i];
      }
  } else if (line <= h/2){

    if (line % 2 == 1)
      srand(seed+line);
    else
      srand(seed+line-1);
    //cout << line << " " << h <<" " << seed+line << endl;
    for (long i=0; i<length; i++)
      if (i % rate == 0)
```

```
      chip[i] = (double) (2 * (rand() % 2) - 1);
      else
      chip[i] = chip[i-1];
  } else {
    if ((h-line+1) % 2 == 1)
      srand(seed+h-line+1);
    else
      srand(seed+h-line);
    //cout << line << " " << h <<" " << seed+h-line+1 << endl;
    for (long i=0; i<length; i++)
      if (i % rate == 0)
      chip[length-i-1] = (double) (2 * (rand() % 2) - 1);
      else
      chip[length-i-1] = chip[length-i];

  }
  return;
}



double*
makeData(const unsigned long length, const unsigned long rate, int*
data){
  int offset = -1;

  double* rval = new double[length];
  for (long i=0; i<length; i++){
    if (i % rate == 0){
      offset++;
      //cout << "Creating " << data[offset] << endl;
    }
    rval[i] = (double) data[offset];
  }
  return rval;
}

double*
makeCarr(const unsigned long length, const double freq){
  int offset = -1;

  double* rval = new double[length];
  for (long i=0; i<length; i++){
    rval[i] = sin( (double) i * freq );
  }
  return rval;
}

double*
makeSignal(unsigned long length, const unsigned long rate, double*
data){
  int offset = (int)(rate/2)-1;

  length = (long)(length/rate);
  double* rval = new double[length];

  for (long i=0; i<length; i++){
```

```
        rval[i] = (double) data[offset];
        //cout << offset << endl;
        offset += rate;
    }
    return rval;
}

//assumes existing sig array of size length
void makeSignal2(double* Sig,unsigned long length, const unsigned long
rate, double* data){
    int offset = (int)(rate/2)-1;

    length = (long)(length/rate);

    for (long i=0; i<length; i++){
        Sig[i] = (double) data[offset];
        //cout << offset << endl;
        offset += rate;
    }
    return;
}



//Ray's Rotation Code
//(Raymond Hwang)
//shift matrix

double* makeshift(double xyshift[2])
{
    double* S= new double[9];

    S[0] = 1;
    S[1] = 0;
    S[3] = 0;
    S[4] = 1;
    S[6] = 0;
    S[7] = 0;
    S[8] = 1;
    S[2] = xyshift[0];
    S[5] = xyshift[1];

    return S;
}


//rotation matrix cw

double* makerot(double angle)
{
    double* R = new double[9];

    R[0] = (double)cos((double)angle);
    R[1] = (double)-sin((double)angle);
    R[2] = 0.0;
    R[3] = (double)sin((double)angle);
    R[4] = (double)cos((double)angle);
```

```
  R[5] = 0.0;
  R[6] = 0.0;
  R[7] = 0.0;
  R[8] = 1.0;

  return R;
}


//multiply 3x3 matrix by matrix a*b

double* mulmatmat(double* b, double* a)
{
  double* temp = new double[9];

  temp[0] = a[0]*b[0]+a[1]*b[3]+a[2]*b[6];
  temp[1] = a[0]*b[1]+a[1]*b[4]+a[2]*b[7];
  temp[2] = a[0]*b[2]+a[1]*b[5]+a[2]*b[8];
  temp[3] = a[3]*b[0]+a[4]*b[3]+a[5]*b[6];
  temp[4] = a[3]*b[1]+a[4]*b[4]+a[5]*b[7];
  temp[5] = a[3]*b[2]+a[4]*b[5]+a[5]*b[8];
  temp[6] = a[6]*b[0]+a[7]*b[3]+a[8]*b[6];
  temp[7] = a[6]*b[1]+a[7]*b[4]+a[8]*b[7];
  temp[8] = a[6]*b[2]+a[7]*b[5]+a[8]*b[8];

  return temp;

}


//multiple a 3x3 matrix by a vector a*b

double* mulmatvec(double* b, double* a)
{
  double* temp = new double[3];

  temp[0] = a[0]*b[0]+a[1]*b[1]+a[2]*b[2];
  temp[1] = a[3]*b[0]+a[4]*b[1]+a[5]*b[2];
  temp[2] = a[6]*b[0]+a[7]*b[1]+a[8]*b[2];

  return temp;

}

//inverse 3x3 matrix

double* invmtx(double* T)
{
  double* I = new double[9];

  I[0] = T[8]*T[4] - T[5]*T[7];
  I[1] = T[2]*T[7] - T[8]*T[1];
  I[2] = T[5]*T[1] - T[2]*T[4];
  I[3] = T[5]*T[6] - T[8]*T[3];
  I[4] = T[8]*T[0] - T[2]*T[6];
  I[5] = T[2]*T[3] - T[5]*T[0];
  I[6] = T[7]*T[3] - T[4]*T[6];
```

```
  I[7] = T[1]*T[6] - T[7]*T[0];
  I[8] = T[4]*T[0] - T[1]*T[3];

  double tmp = T[0]*T[4]*T[8] + T[1]*T[5]*T[6] + T[2]*T[3]*T[7] -
T[2]*T[4]*T[6] - T[0]*T[5]*T[7] - T[1]*T[3]*T[8];

  //cerr << tmp;

  for (int i = 0; i < 9; i++)
    I[i] /= tmp;

  return I;

}


int
index(pgm* image, int x, int y){
  return image->columns * y + x;
}


//transforms image and interpolates

void transform(pgm* image, pgm* output, double* I)
{
  double* in = new double[3];
  double* out;

  for (int i=0; i<output->columns; i++){
    for (int j=0; j<output->rows; j++){

      in[0] = (double) i;
      in[1] = (double) j;
      in[2] = 1.0;

      out = mulmatvec(in, I);

      int x0 = (int) (floor((double) out[0]));
      int x1 = (int) (ceil((double) out[0]));
      if (x0 == x1) x1++;

      int y0 = (int) (floor((double) out[1]));
      int y1 = (int) (ceil((double) out[1]));
      if (y0 == y1) y1++;

      double xscale = out[0] - (double) x0;
      double yscale = out[1] - (double) y0;

      if (x0 < 0 || x1 >= output->columns ||
        y0 < 0 || y1 >= output->rows){
      output->data[index(output, i, j)] = 0.0;
      } else {
      double dx0y0 = image->data[index(output, x0, y0)];
      double dx0y1 = image->data[index(output, x0, y1)];
      double dx1y0 = image->data[index(output, x1, y0)];
      double dx1y1 = image->data[index(output, x1, y1)];
```

```
      double tmp1 = xscale * (dx1y0 - dx0y0) + dx0y0;
      double tmp2 = xscale * (dx1y1 - dx0y1) + dx0y1;
      double lev =  yscale * (tmp2- tmp1) + tmp1;

      output->data[index(output, i, j)] = lev;
      //image->data[index(image, ii, jj)];
      }

      delete [] out;
    }
  }


  delete [] in;
}
```

ttplace.cc

```
/*C++ libraries*/

#include <iostream.h>
#include <fstream.h>  /*for file output*/
#include <iomanip.h>  //for setw()//
#include <stdlib.h>
#include <time.h>  /*for rand()*/
#include <string>
#include <strstream>


extern "C" {
#include <stdio.h>
#include <math.h>
}
#include "settings.h"
#include "util.hh"

/*error message function*/

void error(const char* s, const char* s2 = "") {
  cerr << s << ' ' << s2 << '\n';
  exit(1);
}

// Counts comment lines gets image type
int clines(const char* filename, int& img_type)
{
  int count = 0;
  char* ch = new char[1];

  ifstream from(filename);

  from.get(ch[0]);
  from.get(ch[0]);

  img_type = atoi(ch);
```

```
  do{
    from.get(ch[0]);
  } while (ch[0] != '\n');

  from.get(ch[0]);

  while (ch[0] == '#')
    {
      count++;
      do{
      from.get(ch[0]);
      } while (ch[0] != '\n');
      from.get(ch[0]);
    }

  return count;
}


/*load picture data into structure*/
void get_pgm(const string filename, pgm* image) {

  ifstream from(filename.c_str());
  if (!from) error("cannot open input file ", filename.c_str());

  char ch;
  long temp;
  long size;

  int img_type;

  int number = clines(filename.c_str(),img_type);

  if (img_type ==2) {
    for (int lines = 0; lines <= number; lines++)
      do{
      from.get(ch);
      } while (ch != '\n');

    from >> temp;
    image->columns = temp;
    from >> temp;
    image->rows = temp;
    from >> temp;
    image->maxdepth = (double)temp;

    cout << image->columns << "x" << image->rows << " ascii PGM file
read." << endl;


    size = (image->columns) * (image->rows);
    image->data = new double[size];

    for (int index = 0; index < size; index++) {
    from >> temp;
    *((image->data)+index) = (double)temp;
```

```
    }

    return;
  }else if (img_type ==5) {

    ifstream is(filename.c_str());

    char buffer[128];

    is.getline(buffer, 128);

    is.getline(buffer, 128);
    while(buffer[0] == '#'){    is.getline(buffer, 128); }


    istrstream str(buffer, 128);

    long width, height;

    str >> width;
    str >> height;
    is.getline(buffer, 128);
    cout << width << "x" << height << " binary PGM file read." << endl;

    image->columns = width;
    image->rows = height;
    image->maxdepth = (double)255;

    long size = (image->columns) * (image->rows);
    image->data = new double[size];


    unsigned char mybuf[1024];
    long num;
    long index = 0;
    //    while(is.read(mybuf, 1024)){
    while (1) {
      is.read(mybuf, 1024);
      for (long i=0;i<1024; i++){
      long tlong = mybuf[i];

      image->data[index] = (double) mybuf[i];
      index++;
      //cout << index << " to " << size << endl;
      if (index >= size) return;
      }
    }

    return;
  } else {
    cerr << "ERROR! Unknown PGM file format: " << img_type << endl;
    exit(1);
  }
}


/*send image to file*/
```

```
void write_pgm(const string& filename, pgm* image)
{
  ofstream to(filename.c_str());
  if (!to) error("cannot open output file", filename.c_str());

  to << "P2" << '\n' << "# " << filename << '\n';
  to << image->columns << " " << image->rows << '\n';
  to << image->maxdepth << '\n';

  double size = (image->columns) * (image->rows);

  for (int rindex = 0; rindex < image->rows; rindex++)
    {
      for (int cindex = 0; cindex < image->columns; cindex++) {
      long val = (long)(*((image->data)+cindex+(rindex*(image-
>columns))));
      if (val > 255) val = 255;
      else if (val < 0) val =0;
      to << val << setw(4);
      }
      to << '\n';
    }

  return;
}

#include "settings.h"

#define PI M_PI
#define PI2 PI*2

int*
loadData(const string& filename, const int r, const int c, long& bits){
  // Open files for input
  ifstream dfile(filename.c_str());
  if (!(dfile)){cerr << "Could not open datafile for reading" << endl;
exit(1);}

  int FILELEN = 0;
  int bperrow = c / DRATE;
  int isize = bits;
  int extra = 0;
  int* data;
  char ch = 'f';
  dfile.get(ch);
  //cout << ch << endl;

  while (!dfile.eof()) {
    dfile.get(ch);
    if ((ch = '0') || (ch = '1')) FILELEN++;
  }

  dfile.close();
  cout << "I think I have " << FILELEN << " bits of data" << endl;
  cout << "Fitting " << bperrow << " bits/row every "
       << CRATE << " rows."  << endl;
```

```
  if ((bperrow *= (r/CRATE)) < FILELEN) FILELEN = bperrow;
  else extra = bperrow - FILELEN;

  dfile.open(filename.c_str());
  cout << "Encoding " << FILELEN << " bits into " << c << "x" << r << "
pixel area of image." << endl;

  //ADD extra datapoints for end of imagefile file
  data = new int[FILELEN+extra+2];
  if (data == NULL) error("data space not allocated\n\n");

  for (long i = 0; i < FILELEN; i++) {
    char c;
    dfile.get(c);
    data [i] = (c == '1') ? 1 : -1;
  }
  data[FILELEN] = 1;
  data[FILELEN+1] = 1;

  while (extra > 0) {
    data[FILELEN+1+extra] = 1;
    extra--;
  }

  bits = FILELEN;
  return data;
}



void initializepgm(pgm* simage, pgm* timage)
{
  timage->columns = simage->columns;
  timage->rows = simage->rows;
  timage->maxdepth = simage->maxdepth;
  long size = (simage->columns) * (simage->rows);
  timage->data = new double[size];

  return;
}


void point_sub(pgm* simage, pgm* timage)
{
  long size = (simage->columns) * (simage->rows);

  for (int index = 0; index < size; index++)
    *((timage->data)+index) = *((simage->data)+index) - *((timage-
>data)+index);
  return;
}


void abs_image(pgm* timage)
{
  double current_max = 0;
  long size = timage->columns * timage->rows;
```

```
  for (int index = 0; index < size; index++)
    {
      if (*((timage->data)+index) < 0)
      *((timage->data)+index) = -(*((timage->data)+index));

      if (*((timage->data)+index) > current_max)
      current_max = *((timage->data)+index);
    }

  timage->maxdepth = current_max;

  return;
}


void rescale_image(pgm* timage, double scale)
{
  double temp;
  double current_max = *((timage->data)+0);
  long size = timage->columns * timage->rows;
  double sum = 0;

  for (int index = 0; index < size; index++)
    {
      if (*((timage->data)+index) > current_max)
      current_max = *((timage->data)+index);
    }
  cout << "Image rescaled to 0.0 to " << scale << "\n";
  if (current_max == 0) {
    current_max = 1;
    sum = size/2;
    cout << "Visibility = 0; Forcing .5 for all\n";
    for (int index = 0; index < size; index++)
      {
      *((timage->data)+index) = .5;
      }
  }else{
    for (int index = 0; index < size; index++)
      {
      temp = *((timage->data)+index) / current_max;
      *((timage->data)+index) = temp * scale;
      sum += temp * scale;
      }
  }
  timage->maxdepth = current_max;
  cout << "Mean depth = " << sum/size << "\n\n";

  return;
}

void blur_image(pgm* simage, pgm* timage, int size)
{
  double new_depth;
  double pt_total;
  long yinit, xinit;
  long currenty, currentx;
```

```
  int max_dist = (size - 1)/2;
  long max_index = simage->columns * simage-> rows;

  for (int yindex = 0; yindex < simage->rows; yindex++)
    for (int xindex = 0; xindex < simage->columns; xindex++)
      {
      yinit = yindex - max_dist;
      xinit = xindex - max_dist;
      pt_total = 0;

      for (int syindex = 0; syindex < size; syindex++)
        {
          currenty = yinit + syindex;

          if (currenty < 0)
            currenty = -currenty;

          if (currenty > simage->rows)
            currenty = 2*simage->rows - currenty;

          if (currenty == simage->rows)
            currenty = currenty - 1;

          for (int sxindex = 0; sxindex < size; sxindex++)
            {
            currentx = xinit + sxindex;

            if (currentx < 0)
              currentx = -currentx;

            if (currentx > simage->columns)
              currentx = 2*simage->columns - currentx;

            if (currentx == (simage->columns))
              currentx = currentx - 1;

            pt_total += *((simage->data)+currentx+(currenty*(simage-
>columns)));
            }
        }

      new_depth = pt_total/(size*size);
            *((timage->data)+xindex+(yindex*(timage->columns)))=
new_depth;
      }
  return;
}

// Create a blank thread to be added to image later
void makethread(pgm* thread, int w, int h, int Amp, const string&
filename)
{
  //INIT Thread
  long size = w * h;
  thread->columns = w;
  thread->rows = h;
  thread->data = new double[size];
```

```
    //INIT all extra parts of thread to zero
    for (int ii=0; ii < size; ii++)
      thread->data[ii] = 0.0;


    int *data;

    int bperrow = w / DRATE;
    int Lines2Code;

    long isize = h * bperrow / CRATE;
    long FILELEN = isize;
    data = loadData(filename, h, w,FILELEN);

    if (FILELEN == isize)
      Lines2Code = h;
    else {
      Lines2Code =(int) (FILELEN*CRATE / bperrow);
      if (((double)(CRATE%bperrow) > 0) && ((double)(CRATE%bperrow) <
(double)bperrow/2.0))
        Lines2Code += (CRATE - CRATE%bperrow);
    }
    cout << "Using " << Lines2Code << " lines of " << h << ".\n";
    double count = 0.0;

    int seed = 77;  //Any random Seed must match Decode seed... duh

    // NEED to init this to random thing so Delete won't barf
    double* Data = new double[1];
    int linecount = 0;

    double * Chip = new double[w];

    //cout << Lines2Code << " = l2c" <<endl;

    for (long scanline = 0; scanline < Lines2Code; scanline++){
      //cerr << "encoding line " << scanline << endl;
      if (scanline % CRATE == 0) {
        delete [] Data;
        Data = makeData(w, DRATE, data+(linecount*bperrow));
        linecount++;
      }

      makeChip2(Chip,w, CRATE, seed,linecount,(int)(Lines2Code/CRATE));

      double * Carr = makeCarr(w, FREQ);

      double * Threadline = &(thread->data[thread->columns*scanline]);

      for (long i=0; i < w; i++)
        Threadline[i] = Amp * Carr[i] * Data[i] * Chip[i];
      delete [] Carr;
    }
    delete [] Chip;
    return;
}
```

```
//  Tries to thread visibility at Random location with given theta.
//  Returns location and total cum vis.  (x,y,vis)
double * trythread(int w, int h, pgm *vm, double theta)
{

  double sint = sin(theta);
  double cost = cos(theta);

  const double wincr = 60;
  const double hincr = 5;

  double wx = (w/2)/wincr * cost;
  double wy = (w/2)/wincr * sint;

  double hx = (h/2)/hincr * sint;
  double hy = (h/2)/hincr * cost;

  double * pos = new double[3];
  pos[0] =rand()%vm->columns;
  pos[1] =rand()%vm->rows;
  pos[2] = vm->data[(int)(pos[0]+pos[1]*vm->columns)];

  int cx,cy;

  for (int i = 1; i < wincr; i++)
    for (int j = 0; j < hincr; j++) {
      cx = (int) (pos[0] + i * wx + j * hx);
      cy = (int) (pos[1] + i * wy + j * hy);
      if ((cx >= 0)&&(cy >= 0)&&(cx < vm->columns)&&(cy < vm->rows))
      pos[2] += vm->data[cx+cy*vm->columns];
      cx = (int) (pos[0] - (i * wx + j * hx));
      cy = (int) (pos[1] - (i * wy + j * hy));
      if ((cx >= 0)&&(cy >= 0)&&(cx < vm->columns)&&(cy < vm->rows))
      pos[2] += vm->data[cx+cy*vm->columns];
    }
  if ((pos[0]-w/2<0)||(pos[1]-h/2<0)||(pos[0]+w/2>vm-
>columns)||(pos[1]+h/2>vm->rows))
    pos[2] -= 10;
  return pos;
}


//Scales Vismask values where a thread will be placed by a
//factor of alpha; Rotation code due to my man, Ray Hwang
(rwhwang@mit.edu)

void placethreadvm(int w,int h, pgm *vm, double theta, double* pos)
{

  double alpha = 0.1;

  int max = w;
  if (h > max) max = h;
  //Create New image to rotate thread
  pgm rot;
  rot.columns = (int)(max* 1.5);
```

```
rot.rows = rot.columns;
int isize = rot.rows*rot.rows;
rot.data = new double[isize];
double* center = new double[2];
center[0] = rot.columns /2;
center[1] = rot.rows /2;
int cx,cy;
for (int x = 0; x < rot.columns; x++)
  for (int y = 0; y < rot.rows; y++) {
    cx = (int) (x + w/2 - center[0]);
    cy = (int) (y + h/2 - center[1]);
    if ((cy<0)||(cy>h)||(cx<0)||(cx>w))
    rot.data[x+y*rot.columns] = 1.0;
    else
    rot.data[x+y*rot.columns] = alpha;
  }

//shift needed to move center to 0,0
double* initial = new double[2];
initial[0] = - center[0];
initial[1] = - center[1];

//Init matrix vars
double* M = new double[9];
double* S = new double[9];
double* R = new double[9];
double* B = new double[9];
double* T = new double[9];
double* I = new double[9];
double* X = new double[9];

//  cerr << "Calculating Transformation\n";

//shift to 0,0
S = makeshift(initial);

//rotate by correct angle, either cw or ccw
R = makerot((double)theta);

//shift back to original position
B = makeshift(center);

//multiply S and R
M = mulmatmat(S, R);

//multiply M and B
T = mulmatmat(M, B);

//invert for transformation of rotated image
double* SS = invmtx(T);

pgm postrot;
initializepgm(&rot, &postrot);
transform(&rot, &postrot, SS);

delete [] SS;
//PLACE rotated thread to image
```

```
  for (int x = 0; x < rot.columns; x++)
    for (int y = 0; y < rot.rows; y++) {
      cx = (int) (x + pos[0] - center[0]);
      cy = (int) (y + pos[1] - center[1]);
      if (!((cy<0)||(cy>vm->rows)||(cx<0)||(cx>vm->columns)))
      vm->data[cx+cy*vm->columns] = vm->data[cx+cy*vm->columns] *
postrot.data[x+y*rot.columns];
    }
  return;
}

//Generates angles for thread placement
double * MakeTheta(const unsigned int l) {

  double * theta = new double[l];

  theta[0] = 0.0;
  if (l > 1) theta[1] = 90.0;
  double incr = 180.0 / (double)(l);
  double count = 0.0;
  for (int i = 2; i < l; i++) {
    if ((i % 2) == 0) count += incr;
    if ((i % 2) == 0) theta[i] = count;
    else theta[i] = count + 90.0;
  }
  return theta;
}

//Places thread to target image rotated by theta radians centered
//  at x0,yo....  Rotation code due to my man, Ray Hwang
(rwhwang@mit.edu)

void placethread(pgm *thread, pgm *vm, pgm *target, int x0, int y0,
double theta)
{

  int max = thread->columns;
  if (thread->rows > max) max = thread->rows;

  //NEW VisMask!  Use same val for DRATE pixels
  //Limimit value to MaxAmp

  double MaxAmp = 70;

  double * NewVm = new double[thread->columns*thread->rows];

  for (int y = 0; y < thread->rows; y++)
    for (int x = 0; x < thread->columns; x++)
      NewVm[x+y*thread->columns] = 0;

  double sint = sin(theta);
  double cost = cos(theta);

  double wx = thread->columns*cost;
  double wy = thread->columns*sint;

  double hx = thread->rows*sint;
```

```
  double hy = thread->rows*cost;

  int cx,cy;

  //DEBUG  vm->maxdepth = 255;
  for (int y = 0; y < thread->rows; y++)
    for (int x = 0; x < thread->columns; x+=DRATE) {
      double avg =0;
      int zeroed = 0;
      for (int iii=0; iii<DRATE;iii++) {
      cx = (int)(x0 -wx/2 -hx/2 + (x+iii)*cost + y*sint);
      cy = (int)(y0 -wy/2 -hy/2 + (x+iii)*sint + y*cost);
      if (!((cx<0)||(cy<0)||(cx>target->columns)||(cy>target->rows))){
        avg += vm->data[cx+cy*target->columns];
        if (vm->data[cx+cy*target->columns] < 0.1)
          zeroed = 1;
        //DEBUGvm->data[cx+cy*target->columns] = 255;
      }
      }
      avg /= DRATE;
      //Use half average and half true vismask val (to soften effect)
      for (int iii=0; iii<DRATE;iii++) {
      cx = (int)(x0 -wx/2 -hx/2 + (x+iii)*cost + y*sint);
      cy = (int)(y0 -wy/2 -hy/2 + (x+iii)*sint + y*cost);
      if (!((cx<0)||(cy<0)||(cx>target->columns)||(cy>target->rows))) {
        //NewVm[x+iii+y*thread->columns] = avg*0.5 + vm-
>data[cx+cy*target->columns]*.5;
        if (zeroed)
          NewVm[x+iii+y*thread->columns] = vm->data[cx+cy*target-
>columns];
        else
          NewVm[x+iii+y*thread->columns] = avg;

        vm->data[cx+cy*target->columns] *= 0.1;
      }else
        NewVm[x+iii+y*thread->columns] = avg;
      }
    }

  //Create New image to rotate thread
  pgm rot;
  rot.columns = (int)(max* 1.5);
  rot.rows = rot.columns;
  int isize = rot.rows*rot.rows;
  rot.data = new double[isize];
  double* center = new double[2];
  center[0] = rot.columns /2;
  center[1] = rot.rows /2;
  float avgamp = 0.0;
  for (int x = 0; x < rot.columns; x++)
    for (int y = 0; y < rot.rows; y++) {
      cx = (int) (x + thread->columns/2 - center[0]);
      cy = (int) (y + thread->rows/2 - center[1]);
      if ((cy<0)||(cy>thread->rows)||(cx<0)||(cx>thread->columns))
      rot.data[x+y*rot.columns] = 0.0;
      else {
```

```
      double val= thread->data[cx+cy*thread-
>columns]*NewVm[cx+cy*thread->columns];
      if (val > MaxAmp) val = MaxAmp;
      rot.data[x+y*rot.columns] = val;
      avgamp+=NewVm[cx+cy*thread->columns];
      }
    }

      //shift needed to move center to 0,0
  double* initial = new double[2];
  initial[0] = - center[0];
  initial[1] = - center[1];

  //Init matrix vars
  double* M = new double[9];
  double* S = new double[9];
  double* R = new double[9];
  double* B = new double[9];
  double* T = new double[9];
  double* I = new double[9];
  double* X = new double[9];

  //shift to 0,0
  S = makeshift(initial);

  //rotate by correct angle, either cw or ccw
  R = makerot((double)theta);

  //shift back to original position
  B = makeshift(center);

  //multiply S and R
  M = mulmatmat(S, R);

  //multiply M and B
  T = mulmatmat(M, B);

  //invert for transformation of rotated image
  double* SS = invmtx(T);

  pgm postrot;
  initializepgm(&rot, &postrot);
  transform(&rot, &postrot, SS);

  delete [] SS;

  //PLACE rotated thread to image
  for (int x = 0; x < rot.columns; x++)
    for (int y = 0; y < rot.rows; y++) {
      cx = (int) (x + x0 - center[0]);
      cy = (int) (y + y0 - center[1]);
      if (!((cy<0)||(cy>target->rows)||(cx<0)||(cx>target->columns)))
      target->data[cx+cy*target->columns] +=
postrot.data[x+y*rot.columns];
    }
  cerr << ".";
```

```
  cout << "Avg Vis Amp = " << avgamp/(thread->rows*thread->columns)
<<endl;
  return;
}


int main(int argc, char* argv[])
  /*usage: imgcode datafile errorfile.pgm outfile */
{

  if ((argc != 9)) error("Wrong number of arguments\nusage: ttplace
data.txt image.pgm threadwidth threadheight num amp search out.pgm");

  // Sdepth = 50000 seems like a good num

  pgm noise;

  get_pgm(argv[2], &noise);
  int w = atoi(argv[3]);
  int h = atoi(argv[4]);
  int tts = atoi(argv[5]);
  int Amp = atoi(argv[6]);
  int sdepth = atoi(argv[7]);

  //CREATE A VISIBILITY MASK FOR INPUT
  pgm temp;
  pgm vismask;
  cout << "Generating Visibility Mask\n";
  initializepgm(&noise, &temp);
  blur_image(&noise, &temp, BLUR1);
  point_sub(&noise, &temp);
  abs_image(&temp);
  initializepgm(&temp, &vismask);
  //vismask is now the current altered version.
  blur_image(&temp, &vismask, BLUR2);
  //THROW OUT TEMP VERSION?
  delete [] temp.data;
  //write_pgm("vm.pgm",&vismask);
  rescale_image(&vismask, 1.0);
  //normalize image to max_depth = scale

  //Copy vismask to other version
  temp.data = new double[temp.rows*temp.columns];
  for (int jj= 0; jj < temp.rows*temp.columns; jj++)
    temp.data[jj] = vismask.data[jj];

  cout << "Calulating thread angles" << endl;
  double * theta =  MakeTheta(tts);
  //srand48(clock());
  srand((unsigned) time (NULL));

  cout << "Testing Thread Positions" << endl;
  double * tpos = new double[2*tts]; // X followed by Y for each point


  for (int i = 0; i < tts; i++) {
    double * best = new double[3];
```

```
    best[2] = 0.0;
    cerr << theta[i] << "\t";
    if (sdepth ==0){
      best[0] =w/2;
      best[1] = h/2;
    }else
      for (int j = 0; j < sdepth; j++) {
      double * curr = trythread(w,h, &temp, (theta[i]*PI/180));
      if (curr[2] > best[2]) {
        delete best;
        best = curr;
      } else delete curr;
      if ((j % (sdepth/10))==0) cerr << ".";
      }
    cerr << endl;

    tpos[2*i] = best[0];
    tpos[2*i+1] = best[1];
    //    cerr << best[0] << " " << best[1] <<endl;
    placethreadvm(w,h, &temp, (theta[i]*PI/180),best);
    delete best;

    //FIX!
    //Don't use temp; instead place thread on the fly...

  }

  for (int i = 0; i < tts; i++) cout << theta[i] << " " << tpos[2*i] <<
" "<< tpos[2*i+1] << endl;


  pgm thread;
  cout << "\nCreating Tartan Thread\n";
  makethread(&thread,w,h,Amp,argv[1]);
  cout << "\nApplying threads.\n";
  for (int i = 0; i < tts; i++)

placethread(&thread,&vismask,&noise,(int)tpos[2*i],(int)tpos[2*i+1],(th
eta[i]*PI/180));

  //write_pgm("vm.pgm",&vismask);
  cout << "\nWriting outputfile: " << argv[8] << endl;
  write_pgm(argv[8],&noise);
  cout << "Done.\n";

}
```

linedecode.cc

```
/*C++ libraries*/

#include <iostream.h>
#include <fstream.h>  /*for file output*/
#include <stdlib.h>
#include <time.h>  /*for rand()*/
#include <string>
```

```
#include <stdio.h>
#include <math.h>
#include <strstream>

#include "util.hh"
#include "settings.h"

/*error message function*/

void error(char* s, char* s2 = "") {
    cerr << s << ' ' << s2 << '\n';
    exit(1);
}

// Counts comment lines
int clines(const char* filename, int& img_type)
{
  int count = 0;
  char* ch = new char[1];

  ifstream from(filename);

  from.get(ch[0]);
  from.get(ch[0]);

  img_type = atoi(ch);

  do{
    from.get(ch[0]);
  } while (ch[0] != '\n');

  from.get(ch[0]);

  while (ch[0] == '#')
    {
      count++;
      do{
      from.get(ch[0]);
      } while (ch[0] != '\n');
      from.get(ch[0]);
    }

  return count;
}


/*load picture data into structure*/
void get_pgm(const string& filename, pgm* image) {

  ifstream from(filename.c_str());
  if (!from) error("cannot open input file ", filename.c_str());

  char ch;
  long temp;
  long size;

  int img_type;
```

```
  int number = clines(filename.c_str(),img_type);

  if (img_type ==2) {
    for (int lines = 0; lines <= number; lines++)
      do{
      from.get(ch);
      } while (ch != '\n');

    from >> temp;
    image->columns = temp;
    from >> temp;
    image->rows = temp;
    from >> temp;
    image->maxdepth = (double)temp;

    cout << image->columns << "x" << image->rows << " ascii PGM file
read." << endl;


    size = (image->columns) * (image->rows);
    image->data = new double[size];

    for (int index = 0; index < size; index++) {
    from >> temp;
    *((image->data)+index) = (double)temp;
    }

    return;
  }else if (img_type ==5) {

    ifstream is(filename.c_str());

    char buffer[128];

    is.getline(buffer, 128);

    is.getline(buffer, 128);
    while(buffer[0] == '#'){    is.getline(buffer, 128); }


    istrstream str(buffer, 128);

    long width, height;

    str >> width;
    str >> height;
    is.getline(buffer, 128);
    cout << width << "x" << height << " binary PGM file read." << endl;

    image->columns = width;
    image->rows = height;
    image->maxdepth = (double)255;

    long size = (image->columns) * (image->rows);
    image->data = new double[size];
```

```
    unsigned char mybuf[1024];
    long num;
    long index = 0;
    //    while(is.read(mybuf, 1024)){
    while (1) {
      is.read(mybuf, 1024);
      for (long i=0;i<1024; i++){
      long tlong = mybuf[i];

      image->data[index] = (double) mybuf[i];
      index++;
      //cout << index << " to " << size << endl;
      if (index >= size) return;
      }
    }

    return;
  } else {
    cerr << "ERROR! Unknown PGM file format: " << img_type << endl;
    exit(1);
  }
}

int*
loadData(const string& filename, const int r, const int c, long& bits){
  // Open files for input
  ifstream dfile(filename.c_str());
  if (!(dfile)){cerr << "Could not open datafile for reading" << endl;
exit(1);}

  int FILELEN = 0;
  int bperrow = c / DRATE;
  int isize = c * r;
  int* data;
  char ch = 'f';
  dfile.get(ch);
  //cout << ch << endl;

  while (!dfile.eof()) {
    dfile.get(ch);
    //ASSUME datafile starts with sequence of 1's and 0's
    if ((ch = '0') || (ch = '1')) FILELEN++;
    //cout << ch << endl;
    // THIS ISN'T WORKING, BUT NOT IMPORTANT
  }

  dfile.close();
  if ((bperrow *= (r/CRATE)) < FILELEN) FILELEN = bperrow;
  dfile.open(filename.c_str());

  cout << "Decoding " << FILELEN << " bits from " << c << "x" << r << "
Tartan thread." << endl;

  data = new int[FILELEN];
  if (data == NULL) error("data space not allocated\n\n");
```

```
  for (long i = 0; i < FILELEN; i++) {
    char c;
    c = dfile.get();
    data [i] = (c == '1') ? 1 : -1;
  }

  bits = FILELEN;
  return data;
}


#define PI 3.141592653589793
#define PI2 6.28318530717959

#define SWAP(a,b) tempr=(a); (a)=(b); (b)=tempr

void four1(double data[], unsigned long nn, int isign)
{
    unsigned long n,mmax,m,j,istep,i;
    double wtemp,wr,wpr,wpi,wi,theta;
    double tempr,tempi;

    n=nn << 1;
    j=1;
    for (i=1;i<n;i+=2) {
      if (j > i) {
          SWAP(data[j],data[i]);
          SWAP(data[j+1],data[i+1]);
      }
      m=n >> 1;
      while (m >= 2 && j > m) {
          j -= m;
          m >>= 1;
      }
      j += m;
    }
    mmax=2;
    while (n > mmax) {
      istep=mmax << 1;
      theta=isign*(6.28318530717959/mmax);
      wtemp=sin(0.5*theta);
      wpr = -2.0*wtemp*wtemp;
      wpi=sin(theta);
      wr=1.0;
      wi=0.0;
      for (m=1;m<mmax;m+=2) {
          for (i=m;i<=n;i+=istep) {
            j=i+mmax;
            tempr=wr*data[j]-wi*data[j+1];
            tempi=wr*data[j+1]+wi*data[j];
            data[j]=data[i]-tempr;
            data[j+1]=data[i+1]-tempi;
            data[i] += tempr;
            data[i+1] += tempi;
          }
          wr=(wtemp=wr)*wpr-wi*wpi+wr;
          wi=wi*wpr+wtemp*wpi+wi;
```

```
      }
      mmax=istep;
    }
}
#undef SWAP

int main(int argc, char* argv[])

{

  if ((argc != 5)) error("Wrong number of arguments\nusage: linedecode
datafile imgfile.pgm w h");

  pgm noise;

  get_pgm(argv[2], &noise);

  int w = atoi(argv[3]);
  int h = atoi(argv[4]);
  double* res1 = new double[w*2/CRATE];
  double* Chip = new double[w];
  double* Sig1 = new double[w];
  long FILELEN = 0;
  int* data = loadData(argv[1], h, w,FILELEN);
  int best = FILELEN/2;
  int bx,by,bitcount,x1,y1;
  int seed = 77;
  int bperrow = w / DRATE;
  int lastline = (int)h;
  int prog = (noise.rows-h)/10;
  int proginc = prog;
  for (int yoff = 0; yoff <= noise.rows-h; yoff += CRATE){
    if (yoff > prog) {cerr << ".";prog+=proginc;}
    for (int xoff = 0; xoff <= noise.columns-w; xoff += CRATE-1) {
      int errorcount = 0;
      bitcount = 0;
      int linecount = 0;
      int targetline = CRATE/2 - 1;
      for (long scanline = 0; scanline < lastline; scanline++){

      if (scanline % CRATE == 0) {
        linecount++;
      }

      if (scanline == targetline) {

        makeChip2(Chip,w, CRATE, seed,
linecount,(int)(lastline/CRATE));

        makeSignal2(Sig1,w, CRATE,&(noise.data[noise.columns *
(scanline+yoff)+xoff]));
        for (long i=0; i<w/CRATE; i++){
          res1[i*2] = ((Sig1[i])+1) * Chip[i*CRATE];
          res1[i*2+1] = 0.0;
        }
        for (long rb = 0; rb < bperrow; rb++) {
          long rinc = rb *2 * DRATE/CRATE;
```

```
          //Take fourier trasform for one DRATE/CRATE length
          four1(res1+rinc-1, DRATE/CRATE, 1);
          int Freq2 = (int)THETA * 2;
          int Freq3 = DRATE*2/CRATE - Freq2;
          int Freq4 = Freq3 +2;
          if (bitcount < FILELEN){
            double phase1 =
(atan2((res1+rinc)[Freq3+1],(res1+rinc)[Freq3]));
            double phase = phase1;
            int last;
            if ((phase > PI/2) || (phase < -PI/2)){
            if (data[bitcount] == 1) errorcount++;
            last = 0;
            } else {
            if (data[bitcount] == -1) errorcount++;
            last = 1;
            }
            bitcount++;
          }
        }

        targetline += CRATE;
      }
      }

      // If error rate low enough trigger
      if ((((float)errorcount / (float)bitcount) < -0.20)  ||
        (((float)errorcount / (float)bitcount) > 0.80)) { //SHOULD be
<= .2
      if (((float)errorcount / (float)bitcount) > 0.80)
        cout << "\nError Rate = " << 1.0-(float)errorcount /
(float)bitcount << " with 180 rotation." << endl;
      else
        cout << "\nError Rate = " << (float)errorcount /
(float)bitcount << endl;
      cout << "Encoding detected at (" <<xoff<<","<<yoff<<")." << endl;
      delete [] res1;
      delete [] Chip;
      delete [] Sig1;
      return(1);
      //ELSE update best var and check if good enough for thorough
search loop
      } else if (errorcount < best) {
      best = errorcount;
      bx = xoff;
      by = yoff;
      }
      if ((errorcount < (int)(.35 * bitcount))||(errorcount > (int)(.65
* bitcount)))
      for (int y1 = yoff-CRATE; y1 < yoff+CRATE; y1++)
        for (int x1 = xoff-CRATE; x1 < xoff+CRATE; x1++)
          if ((y1 >=0)&&(x1 >=0)&&(x1+w
<noise.columns)&&(y1+h<noise.rows)){
          int errorcount = 0;
          bitcount = 0;
          int linecount = 0;
          int targetline = CRATE/2 - 1;
```

```
          for (long scanline = 0; scanline < lastline; scanline++){
            if (scanline % CRATE == 0) {
            linecount++;
            }

            if (scanline == targetline) {
            makeChip2(Chip,w,CRATE,
seed,linecount,(int)(lastline/CRATE));
            makeSignal2(Sig1,w,CRATE,&(noise.data[noise.columns *
(scanline+y1)+x1]));

            for (long i=0; i<w/CRATE; i++){
              res1[i*2] = ((Sig1[i])+1) * Chip[i*CRATE];
              res1[i*2+1] = 0.0;
            }
            for (long rb = 0; rb < bperrow; rb++) {
              long rinc = rb *2 * DRATE/CRATE;
              four1(res1+rinc-1, DRATE/CRATE, 1);
              int Freq2 = (int)THETA * 2;
              int Freq3 = DRATE*2/CRATE - Freq2;
              if (bitcount < FILELEN){
                double phase1 =
(atan2((res1+rinc)[Freq3+1],(res1+rinc)[Freq3]));
                double phase = phase1;

                int last;
                if ((phase > PI/2) || (phase < -PI/2)){
                  if (data[bitcount] == 1) errorcount++;
                  last = 0;
                } else {
                  if (data[bitcount] == -1) errorcount++;
                  last = 1;
                }
                bitcount++;
              }
            }
            targetline += CRATE;
            }
          }
          //Check for Trigger threshold
          if ((((float)errorcount / (float)bitcount) < 0.20)||
            (((float)errorcount / (float)bitcount) > 0.80)) { //SHOULD
be <= .2

            if (((float)errorcount / (float)bitcount) > 0.80)
            cout << "\nError Rate = " << 1.0-(float)errorcount /
(float)bitcount << " with 180 rotation." << endl;
            else
            cout << "\nError Rate = " << (float)errorcount /
(float)bitcount << endl;

            cout << "Encoding detected at (" <<x1<<","<<y1<<")." <<
endl;
            delete [] res1;
            delete [] Chip;
            delete [] Sig1;
            return(1);
```

```
            //otherwise update best
          } else if (errorcount < best) {
            best = errorcount;
            bx = x1;
            by = y1;
          }
        }
    }}
  cout << "\nEncoding not found." <<endl;
  cout << "Lowest Error Rate = " << (float)best / (float)bitcount << "
("<<bx<<","<<by<<")."<< endl;
}
```