2016

The efficient recovery of deleted data from NAND flash memory

Lisa Diercks
University of Northern Iowa

THE EFFICIENT RECOVERY OF DELETED DATA FROM NAND FLASH MEMORY

A Thesis Submitted

in Partial Fulfillment

of the Requirements for the Designation

University Honors with Distinction

Lisa Diercks

University of Northern Iowa

May 2016

This Study by: Lisa Diercks

Entitled: The Efficient Recovery of Deleted Data from NAND Flash Memory

has been approved as meeting the thesis or project requirement for the

Designation University Honors with Distinction

_____	_____
Date	Dr. Sarah Diesburg, Honors Thesis Advisor
_____	_____
Date	Dr. Jessica Moon, Director, University Honors Program

The Efficient Recovery of Deleted Data from NAND Flash Memory

NAND flash memory is used in flash drives, smart phones, and memory cards for digital cameras. While there are ways to recover data from deleted memory, there are no universal or efficient ways to recover data from NAND flash memory. When a user clicks “delete” on a file in this type of memory, the file is not necessarily deleted, but may just be hidden. This means these files are still on the chip, but are inaccessible through normal means. The ability to recover this lost data could help computer forensic examiners during investigations and corporations working to use secure deletion techniques of confidential files. This project will provide a potential solution to recovering data from NAND flash memory.

This research is a continuation of previous research where some code was written in Python using processes. The previous attempt to write this software resulted in software that was slow and would not scale well to large storage devices. This research explores whether software can be rewritten to take advantage of parallel computer execution on multiple processor cores to run and finish execution in a reasonable amount of time that would scale well to larger chip sizes, creating an efficient means of analyzing deleted data from NAND flash memory.

1. INTRODUCTION

The uses of NAND flash memory are growing in recent years. The cost to produce this type of data storage is falling, while the demand and uses for it are increasing [9]. NAND flash memory is often used in flash drives, cameras, and solid state drives. Flash memory can be used to back up files, transfer large files, or even run programs. Forensic examiners need access to them. Unfortunately, they are not

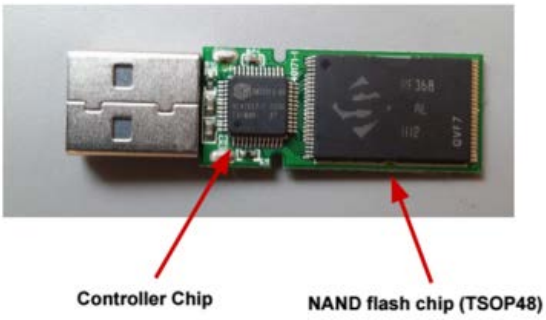

Figure 1

able to access all potentially useful data. This project created a framework to access this data in an easy-to-use manner, using common hardware and specially developed code. Specifically, this project concentrated on making the code more efficient with parallel techniques.

This research paper gives a background on NAND flash memory, followed by a relevant literature review. It then describes the previous work that was done, prior to personal involvement in the study of NAND flash memory. The methodology and detailed description of the code will then be given to provide information on how each section of the developed code works. Finally, this study describes the results that were found after the code was written and future work that can be done with the results.

2. BACKGROUND AND LITERATURE REVIEW

Background on NAND storage technology and parallel coding is followed by a relevant literature review.

a. Background

i. NAND flash memory

This study used flash drives because they are widely available and relatively cheap. Just like any other technology, NAND flash memory comes with its downsides. With its versatility and convenience comes the issue of it becoming a security risk. People can put large amounts of an organization's data on it and give it to a competitor, or install malicious programs on a network [6]. Not only can they transport data when people want them to, but they can also transport data unintentionally. Pressing the delete button for a file on flash memory does not erase the file. Diesburg et al.[5], states, "Typical file deletion only updates the file's metadata (e.g., pointers to the data)…while often leaving the file data intact." The file is not erased until something needs to be written to the area that file was in. Magnets, water, and even explosives have been used to

try to destroy flash drives and they have still had at least some data recovered [14]. This is great if someone wants to get their data back, but if someone wants confidential or sensitive information deleted, it can be difficult for an ordinary user. This means if someone gets ahold of a flash drive, there could be recoverable, sensitive information on it leading to unintentional organization or personal information disclosures.

NAND flash memory is a type of solid-state storage that is built out of transistors and does not have any moving parts like hard drives and retain information without power [1]. These flash drives are made up of a built-in circuit board, which is electronically connected to a TSOP48 NAND flash chip and a controller chip. TSOP is the Thin Small-Outline Package that holds the flash memory with 48 pins that allow the transfer of data or power. This houses the flash memory that is being researched in this thesis. Flash chips are divided blocks [1]. Data in these blocks are stored in pages ranging from 512B to 8KB, in multiples of 512B. This division is shown in Figure 2 [1]. These pages are not always stored consecutively. This structure is important for reading and writing in flash. To write to a page in a block in NAND flash memory, the entire block has to first be erased. This erase-write cycle means any other data in the pages of the block first have to be copied elsewhere before the block is erased [1]. Erase sets the memory to all 1s, while writing sets some bits to 0 [12]. This method, erasing only when a write is needed is what makes flash memory so special, and is why this type of memory is prone to have a lot of data left on it.

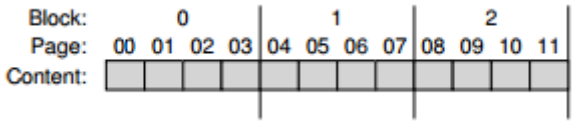

Figure 2: A Simple Flash Chip: Pages Within Blocks

When NAND flash memory stores data, it has to keep track of where it stores things. It does this by keeping an in-memory table. To keep this table even after it loses power, it keeps an out-

of-bound (OOB) area [1]. This OOB area is stored with each page in memory. If flash memory loses power, it has to reconstruct its memory table. It does this by scanning OOB areas [1]. This OOB area means that instead of a page being 512B, it might be 528B with the extra OOB area. This study took this OOB area into account, using it to get more accurate results. The code created in this study was made to be more efficient and universal.

ii. Parallelization

Efficiency will be improved by creating code that runs using parallelization techniques. When computers have multiple cores, they can run multiple processes at once. This means that if someone is trying to do a large task such as multiplying 1,000 numbers by 1,000 different numbers, the program can be divided into "threads" (one for each core) and each core can take a different chunk of numbers to get the work done much faster. Figure 3 [7] clearly illustrates this process where process A and process C are divided into pieces that can be run at the same time, allowing for faster processing.

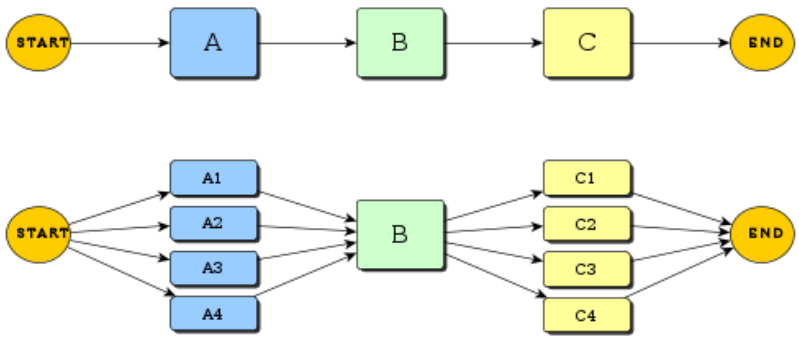

Figure 3

b. Literature Review

There has been little research done on the hidden areas of NAND flash memory that may house data that an owner attempted to delete. Breeuwma et al. [3] used a reverse engineering technique to recover data located in hidden areas of NAND flash memory. Although this does yield results, the methods are time consuming. Not only are the methods time consuming, but

each chip has its own file system and wear algorithm, which means each reverse engineering technique will have to be specific to one type of chip. With hundreds of different chips out there, this approach is not plausible for a general analysis or algorithm to read and analyze NAND flash memory. Although the approach to analyzing chips is not feasible, this article does provide a solid basis on the characteristics of NAND flash memory and how to remove NAND flash memory chips by de-soldering them.

Recovering data from the hidden areas of NAND flash memory not only requires computer science knowledge, but also a bit of mathematics. Billard et al.'s [2] research, includes groundwork for an algorithm for recovering data. Although Billard et al. [2]includes a potential algorithm for recovering data, there are significant limitations to the algorithm and research. One of the most prevalent issues of Billard et al. [2] is that it does not apply the algorithm to a realistic, working data set. The algorithm that is created would have to operate on an unrealistic data set for it to function properly. One issue that would occur in a realistic data set is the issue of bit errors. Bit errors can be created through the microcontroller or other means. It is unfeasible to have a data set without bit errors. Another limitation is that this research is file focused instead of focusing on all the data in the hidden areas. Focusing on the files means missing other data that could be truly important and still left in hidden areas. Finally, Billard et al. [2] admits that another limitation would be duplicate files. This would also occur on a realistic data set, since many files are, "found several times, either as a regular or erased file" [2]. Applying this study's research and methods to a realistic data set (a set of used flash drives from various users) will eliminate these major issues that occur in the current research.

Diesburg et al. [5] gives this research a solid foundation on how the drives were collected and what people believe concerning deletion techniques. This is an important step in the research

that has been previously completed. This research determines that there is no correlation between beliefs concerning deletion and actual deletion effectiveness. In the study “over 60% of the drives tested had recoverable sensitive data,” which included personally identifiable information, corporate commercial data, corporate confidential data and illicit data [5]. This study does not only look at the types of information left on NAND flash memory, but also looks at potential deletion techniques. This piece of the research is an important part of analyzing the known data image against the raw data image. Depending on the deletion technique, it may be easier, harder, or even impossible to recover data in the hidden areas of NAND flash memory.

3. PREVIOUSLY COMPLETED WORK

This research began with a buyback program run by Diesburg et al. [5] that collected various flash drives by trading students new flash drives for old. With this buyback program she conducted a survey about students’ thoughts and beliefs about deletion, and analyzed the actual effectiveness of the methods. This study “found no correlation between users’ perceived versus actual effectiveness of deletion” [5]. Diesburg et al. [5] also bought flash drives from Amazon and eBay to compare against the students’ flash drives. These flash drives bought from Amazon and eBay were found not have any statistical difference in terms of deletion methods or presence of sensitive data. This helps illustrate how little knowledge there is about effective deletion.

During Diesburg et al.’s [5] study, one researcher removed all the NAND flash memory chips from the flash drives. The researcher did so using a heat gun and desoldering the TSOP48 NAND flash chip from the circuit board it is connected to. Previously two chip readers had been purchased, a Dataman 48Pro-2 [4] and a TNM 5000 [10], and each chip was fed into a reader. The reader would then attempt to read the memory and if it was successful, the raw memory dump was saved. This raw memory dump would contain a file with pages of 1’s and 0’s (binary

code). This route was chosen for extracting the raw image of NAND flash memory because of its “guarantee that no data is written in flash memory” and “a complete forensic image can be produced” [3]. These two chip readers were able to read approximately 61% of the flash drives purchased. The data taken from drives using the chip readers included data in the “hidden” areas, which would include data without pointers that users may believe to be deleted. This data will be referred to as the “raw” data or image. Metadata was collected from all of the drives, such as basic characteristics, manufacturer, model and size. Unmodified binary level images of the data on the drives were taken using the Linux dd command. This method of copying data from the drives collects the data that has not been “deleted” by users. This data will be referred to as the “known” data or image.

After readers attempted to read all the chips, a student employed by Dr. Diesburg worked to write code that would compare the known image to the raw image that was read by the readers. The code that was written, was done in Python. Python is a useful language to write code, especially since it is easy to read and uses built in data structures. However, it is a slow language and is very difficult to parallelize. Parallelization of the code once it is translated into the faster, but more complex C, was one of this research’s main goals. Depending on the size of the chip, the current Python code took hours or even sometimes days to do comparisons on the data. Although the code worked sometimes, it also sometimes gave irregular results. This was another goal of this research, to see if there were any flaws in the previous code and attempt to see what the issue(s) might have been. The hypothesis was as follows:

i) Can deleted data in erase blocks (hidden areas) of NAND flash memory be recovered by obtaining a raw dump of the NAND flash memory and analyzing it against any documents currently seen in memory?

ii) Can recovering deleted data in erase blocks be done in a reasonable amount of time so that the method is universal and cost-effective, where the program would be useful to professionals investigating deleted data?

4. METHODOLOGY

a. New Chip Reader

Research for this project began with the installation of a new chip reader, a Xeltek SuperPro 6100 [13]. A third chip reader was purchased in an attempt to improve the number of flash chips that are able to read. This began with the installation of Windows 7 on a wiped hard drive, which allowed for a blank slate to install and write a program on. This computer was connected to a protected server, which stopped other people from accessing or editing any data. From here, the new chip reader was installed using the disk and instructions provided by the company. After the chip reader was installed Dr. Diesburg provided all of the chips that the other two chip readers had been unable to read. These included many SanDisk chips. Unfortunately, this chip reader could not read the TSOP48 SanDisk chips used in this research. This chip reader could read SanDisk chips as advertised, but not TSOP48 chips that this study used. After testing all the chips, this chip reader was not successful in reading any of the chips that the other two chip readers could not read. Unfortunately this study was unable to increase the number of flash chips that were able to read with this extra reader.

b. General Code Design

Rewriting this code in C instead of Python was done to improve the efficiency of the code that will analyze the known (intact) image against the raw image that was received through the chip readers. This was done through a specific type of “fuzzy matching” algorithm that is commonly used to detect similarity and plagiarism in written essay documents. A matching

algorithm was chosen because it would determine which documents are in both the raw and the known image. Once the documents on both are identified, it will be easy to identify the unmatched ones on the raw device. These documents would be the ones without pointers that were left on the raw image after people tried to delete them. An example of this is shown in Figure 4 where it is clear to see that Doc4 does not have a match. Doc4 lost its pointer in the known image, but still exists in the raw image. The device storage was broken down into word (8 byte) and document (512-528) byte segments and comparisons were made between each word in each document in the raw read and each word in each document in the known read. The code was rewritten in a form where processes can run in a parallel form.

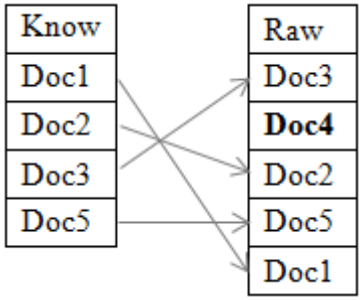

Figure 4

c. General Code Implementation

i. *NAND flash memory*

When this program first starts, the user has to give it some information. The program needs to know what two files need to be compared, so it takes in two arguments, the raw filename (a binary file created by using the chip reader on the flash memory) and the known filename (a binary file created by using the dd command in Linux from the same flash memory).

ii. Data Structures

The main function in this program begins by creating an empty hash table. The structure and code for these hash tables comes from the widely used, open-source, uthash (pronounced U-T-hash), created by Troy D. Hanson [11]. C does not have a built in hash table structure, like Python or other languages. Instead a user has to build the structure themselves. A hash acts very similarly to a dictionary, using key:value pairs. In a dictionary, these pairs would be

word:definition, but this study uses a doc_id:word pairs and then each word also corresponds to a count in a word:count pair. Hanson's uthash allows users to add/replace, find, delete, count, and iterate through items, which will all be utilized in the code created in this research.

iii. Reading Known and Raw Files

This empty table is then used in the read_file_create_hash function. This function reads one of the given binary files. It pulls in a specific number of bytes from the file that is determined by the size of a doc, which is referred to as doc_size. This doc_size contains multiple chunks of data that are referred to as "words" in this study. This function then loops through each "word" in a "doc," counting how many times each word appears in that doc. This counting is done using increment_doc_hash, which takes in what hash table it needs to use, the doc_id the word is in, and what word needs to be incremented. As an example:

doc with ID 0 that contained "hello hello world doc"

doc with ID 1 that contained "hello I'm document one"

a hash table using the format in this study would look like Figure 5. Now the words would actually be binary in the form of 1's and 0's, but this gives an understandable example of the hash table format. The code for this resembles:

Doc_ID	Word	Count
0	hello	2
	world	1
	doc	1
1	hello	1
	I'm	1
	document	1
	one	1

Figure 5

```
for(doc_id = 0; doc_id<num_docs; doc_id++)
{
fread(buffer, doc_size, 1, in_file);
buf_pointer = buffer;
printf("doc_id: %i \n", doc_id);
total_words_read = 0; //reset for each doc_id
for (total_words_read = 0; total_words_read<num_words_in_doc; total_words_read++)
       {
       memcpy(word, buf_pointer, WORD_SIZE); //copy word from the buffer into word
       buf_pointer = buf_pointer + WORD_SIZE; //set buffer pointer read next word
       if ((memcmp(word, stop_word0, WORD_SIZE) == 0) || (memcmp(word, stop_word1,
WORD_SIZE) == 0))
              {
                     //do  nothing for stop word
              }
              else
```

```
                {
                        increment_doc_hash(&whatever_hash, word, doc_id);
                }
        }
}
fclose(in_file);
return whatever_hash;
```

Another important piece of this function is that it is leaving out stop words. These are unimportant, very common words; in this case they are words “0000 0000” and “1111 1111” in binary. Stop words are similar to words like “the” and “a” in English. They are common and do not distinguish one document from another since almost all documents contain them. These areas full of 0’s or 1’s may just be blank areas of leftover space that was not written to, therefore does not contain words that need to be counted. Leaving out these stop words would help to get a more accurate calculation when normalizing the counts of each word.

iv. Normalization

Once all of the words in each doc are put into the hash table, the next step is to normalize each of the counts. This is done through a “bag of words” approach where the number of times a word appears is important, but not the order. When a document mentions a word a lot, that word is more important compared to others. Each word is assigned a “weight” that depends on the number of occurrences of a word in a document [8]. This means that if a doc has 100 words and has “hello” in it five times, that hello would not be as important (or carry as much weight) as a document that only has twenty words and has “hello” in it five times. However “it seems unlikely that twenty occurrences of a term in a documented truly carry twenty times the significance of a single occurrence” [8]. This program normalizes words by going through each doc in the hash table that has already been created. It goes through each word in the doc using HASH_ITER it takes the count of each word and changes it to 1 + log(count). It then adds up each of these new values into a total, and once done, it square roots this total. Finally it goes back

through the counts in the doc and divides each count by the square root of the total it previously found. An example of these calculations is given in Figure 6 below.

Doc_ID	Word	Count	1 + log(count)	Normalized
0	hello	2	1.30	.68
	world	1	1	.52
	doc	1	1	.52
Sqrt(Sum(counts^2))			1.92	
1	hello	1	1	.5
	I'm	1	1	.5
	document	1	1	.5
	one	1	1	.5
Sqrt(Sum(counts^2))			2	

Figure 6

In the code the process looks like:

```
for (doc_id = 1; doc_id  <= 20; doc_id++) //loop through doc IDs
{
	HASH_FIND_INT(whatever_hash,&doc_id,found_doc);
	total_norm = 0;
	//There is no doc 1 hash entry.
	HASH_ITER(hh,found_doc->doc_hash,found_word,tmp_word)
	{
		found_word->count = 1 + log(found_word->count);
		total_norm = total_norm + pow(found_word->count,2);
	}
	sqrt_total = sqrt(total_norm);
	HASH_ITER(hh,found_doc->doc_hash,found_word,tmp_word)
	{
		found_word->count = found_word->count / sqrt_total;
	}
}
```

From the example initially used, the normalization would change the hash table to have the new normalized values:

v. Comparison between Documents to Generate Scores

After normalizing all of the values, the code finally compares the words. Each document in the raw image has to be looked at and each document in the known image has to be compared to it. This is done with the first "for" loops that iterate through each doc_id. Then it has to go through all the words. Initially this was done by using HASH_ITER to run through all the words in a raw doc id and then use HASH_ITER to run compare each of those words to each of the

words in a known doc id. If the words match, the normalized value of one word is multiplied by the normalized value of another word and added to a total. This method caused nested “for” loops that were four loops deep. In addition, a running total had to be kept, but reset with each new raw doc and known doc comparison. This method was not compatible with a parallel paradigm for the program. The issue was that the total the threads would reset the total at times when others were using them, causing incorrect scores.

Since this method was not compatible with parallelization, a new method was devised, which used temporary arrays to store data. These arrays allow the code to be parallelized because of the way they store data. When the matrixes are made, data is stored by index, with the first index being 0 so if the word “World” is being looked for in the top array of Figure 7 it would be written as array[1] in the code.

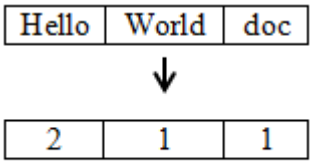

Figure 7

The code still operate with “for” loops, with the iterations and comparisons taking place in a for loop that goes through each raw_id and a for loop within that for each known_id. These loops cause each raw_id to be compared to each known_id. The new matrixes that are created in the parallel method include a matrix of all the words in a raw document (raw_word_matrix), another for all the words in a known document (known_word_matrix), one for all the counts that correspond to the words in the raw document (raw_count_matrix) and one for all the counts that correspond to all the words in the known document (known_count_matrix). Each number raw or known matrix of counts matches to the word in its corresponding matrix. An example of what a word matrix and corresponding count matrix would look like can be seen in Figure 6 above with the top and bottom matrices respectively. This matrix creation can also be seen below:

```
for (raw_id = 0; raw_id<rnum_docs; raw_id++)
{
	for (known_id = 0; known_id<knum_docs; known_id++)
```

```
{
        score = 0;

        raw_num_words = HASH_COUNT(raw_hash);
        known_num_words = HASH_COUNT(known_hash);
        raw_iter = 0;
        HASH_FIND_INT(raw_hash,&raw_id, raw_found_doc);
        if (raw_found_doc == NULL)
                continue;
        HASH_ITER(hh, raw_found_doc->doc_hash, raw_found_word, raw_tmp_word)
        {
                raw_word_matrix[raw_iter] = raw_found_word->w;
                raw_count_matrix[raw_iter] = raw_found_word->count;
                raw_iter++;
        }
        known_iter = 0;
        HASH_FIND_INT(known_hash,&known_id, known_found_doc);
        if (known_found_doc == NULL)
                continue;
        HASH_ITER(hh, known_found_doc->doc_hash, known_found_word, known_tmp_word)
        {
                known_word_matrix[known_iter] = known_found_word->w;
                known_count_matrix[known_iter] = known_found_word->count;
                known_iter++;
        }
```

Once these arrays are filled in for the raw_id and known_id of the current iteration, a temporary array (tmp_array) has to be created for the math to take place in. This array is a 2D array that is initially filled with 0s. This can be imagined just like a table filled with 0s. The number of rows the table has is determined by the number of words in the raw doc and the number of columns is determined by the number of words in the known doc. If a word in the raw doc matches a word in the known doc, their normalized values are multiplied. Using example docs from the previous figures and imagining that one is from the raw image and the other is from the known, tmp_array would look something like Figure 8. It can be seen in Figure 8 that

	Hello	World	Doc
Hello	.68 * .5 = .34	0	0
I'm	0	0	0
Doc	0	0	.52 * .5 = .26
One	0	0	0

Figure 8

all of the words that do not match contain 0s. Only the spots where the words match have products in them. These products will then be summed and put into an output array. In the code, this is done by looping through each word in a specific raw doc ID and comparing the word to each word in a known doc ID. The code then checks if the words are the same with and "if" statement, and if they are the same, it records the product of their normalized scores. This is shown in the code below.

```
#pragma omp parallel for private(j,i,match) shared(tmp_array)
for(i = 0; i < known_num_words; i++)
{
	for (j = 0; j < raw_num_words; j++)
	{
		match = memcmp (known_word_matrix[i], raw_word_matrix[j], WORD_SIZE);
		if(match)
		{
			//don't match
		}
		else
		{
			tmp_array[i][j] = known_count_matrix[i]*raw_count_matrix[j];
		}
	}
}
```

The first line in this set of code is a pragma. This is where this research was finally able to parallelize the code, using all of the arrays that were created. These parallel sections are what was expected to speed up the code. Instead of having the billions of compares getting done one at a time, threads should cut down on the amount of time it takes to complete this section.

The output array is the final array that contains scores for the comparison of each raw doc to each known doc. Each spot in the table corresponds to an index so output_array for raw doc 2 and known doc 3, would be row 3, column 4 (the array indexes begin at 0 instead of 1). Each of these index holds the sum of the tmp_array that was created for that same doc comparison. If the tmp_array in Figure 8 is a comparison of raw doc ID 0 and known doc ID 0, it would be placed in the

	0	1	2
0	**.6**		
1			
2			

Figure 9

corresponding spot in the array. It can be seen in Figure 9, the 0,0 spot holds a sum of .6, which was obtained from adding the values in Figure 8 (.34 + .26). The closer this score is to 1, the more similar the docs are. This score and addition of scores from the tmp_arrays is done in the code below. The code loops through each spot in the tmp_array, adding it to the score. This score is then placed in the output matrix in the last line of this code. This code is also able to be run in parallel since all the threads can help add up the values in a tmp_array.

```
score = 0;
#pragma omp parallel private(y,z) shared(score)
{
#pragma omp for reduction(+:score)
for (y = 0; y < raw_num_words; y++)
{
	for (z = 0; z < known_num_words; z++)
	{
		score += tmp_array[y][z];
	}
}
}
output[raw_id][known_id] = score;
```

vi. Output

Finally scores from the output array are put into a csv file, which can be opened in Excel. This is only done after every raw doc has been compared to every known doc and the output array has finished being created. To output data faster, rows that were completely 0 were left out. This means that words in that raw doc ID had no matches with words in any known doc ID. To put all of these scores into the csv file, the code loops through each row and each columns with a check that is called match. If an output in a row is found not to be 0 then the row is written to the output. This process of writing the output is shown below.

```
FILE *outputp;
outputp = fopen("output.csv","w+");
match=0;
for (i = 0; i < rnum_docs; i++)
{
	for(j=0;j<knum_docs;j++){
		if(output[i][j]!=0)
		{
			match=1;
		}
```

```
        }
        if(match)
        {
                fprintf(outputp,"raw:%i ",i);
                for(j = 0; j < knum_docs; j++)
                {
                        if(match)
                                fprintf(outputp,"%f,",output[i][j]);
                }
                fprintf(outputp, "\n");
        }
        match=0;
}
fclose(outputp);
```

The last step in the code is to free up memory by deleting the hash tables that were created. This is done in the main function of the code using the delete_entire_doc_hash function that was created. The entire program that was created through this research can be seen in the Appendix.

5. RESULTS

This code was run on a University of Northern Iowa server that is only accessible to four people, including myself. This server was set up by one of the professors at UNI mainly for himself and a couple other professors to use. The server has twelve cores, which allowed the program to run twelve threads on it (one for each core). The machine also has 64G RAM.

The timings of the code are shown in Figure 10. This graph shows the time it takes for the previous student's Python code to run compared to what was created in this research. It is easy to

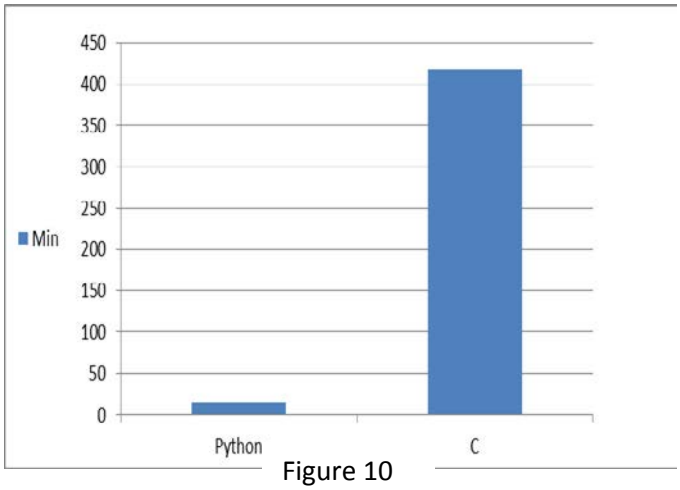

Figure 10

see that this C code did not turn out as well as expected. Python code took on average about fourteen minutes, while the code created in this research took approximately 420 minutes. These timings were derived from running each code five times on the same 64MB chip and taking the average. Writing to the output file was not included in either of the code timings.

Unfortunately this study was not able to create a large section of the code in parallel. When this research began pragmas were discussed as this line of code that would instantly parallelize code, but after running into many problems with them, it was discovered that a lot of work had to go into using them. From the discussion in the General Code Implementation section, it is easy to see that many arrays had to be created, just to allow a few things to run in parallel. It was hoped, that creating these arrays would allow all of the "for" loops to run in parallel. The outside "for" loop that runs through all the raw doc ID's was never able to be put into parallel. This study ran into many errors attempting to do this, and while the creation of arrays helped with cutting the time it takes for compares to be done, it also takes time to create and write to these arrays. Two professors and myself attempted to find a way to put this large loop into parallel without success.

a. Lessons Learned

As mentioned before, attempts to parallelize the code caused errors and incorrect output. This was because each thread needed to access the same hash table. Access to the hash table was done through a pointer and because the hash table was so large it was not stored all in one chunk of memory. Pieces of the hash table were stored in various areas of memory with pointers going to each one. When this study tried to make the outer "for" loop parallel, multiple threads would attempt to access the hash table at the same time. This concurrent access moved pointers to incorrect areas, or even causing Segmentation Faults (a memory access fault).

In an attempt to get the pragmas to work and parallelize the code, private variables were implemented. Private variables are variables where each thread has their own copy of the variable. This means if variable “j” starts as equaling 1 for all threads. When one thread changes its value to 2, it does not change the value the other threads have for that variable. The other threads continue to believe “j” equals 1. The hash table pointer was made private in an attempt to solve the Segmentation Fault issues. At first it seemed as though it did. The code ran in approximately two minutes and thirty seconds. This was very exciting, but upon closer inspection, the code was running this fast because the hash table pointer was pointing to NULL. This was because a pointer is always pointing the same place in memory. Copies of a pointer cannot be made because a copy of it cannot point to the same place, which means the pointers were not pointing to the hash table. This study still ended up using private variables for iterators, but they would not work for pointers.

Originally, the plan was also to parallelize the creation of the hash table, but the exact same issue came up with reading the files. When reading the files, pointers are moved to show where in the file the current iteration is at. If the code is put in parallel, this pointer would have to be private, but as discussed before, pointers cannot be private.

Locks, critical sections, and barriers are all different implementations of stopping one thread from essentially “breaking” something another thread did. A lock or critical section stops two threads from accessing something or doing something at the same time as another thread. These were attempted to be used around pointers, but since so much of the code is connected to those pointers, the locks had to be around huge sections of code. If a lock or critical section is around a huge section of code, that code essentially gets done sequentially just like before. Barriers stop

all threads from moving on before all are done, but this is done by default after “for” loops, so trying to implement them, usually meant they already existed in those spots by default.

6. FUTURE RESEARCH

a. Code Changes

Future research can continue to try and parallelize sections of the code. The loop to get working would be the outer for loop that was discussed before. Other people with more or even just different knowledge of parallelization might see something that the researcher for this study did not.

One idea to parallelize the code was to create multiple copies of the hash table. Since pointers cannot be made private (cannot have multiple copies of a pointer), different pointers would be made for each hash table, eliminating Segmentation Faults that were previously run into. This idea could be feasible, but would require quite a bit of editing. First, there would have to be a way to create multiple pointers with various variable names. This can be done through hard coding, but the goal of this code is to make it universal and efficient. The code should allow for users with different numbers of cores to run it with a different number of hash tables (one for each core/thread) without to edit the code. Not only this, but each thread somehow has to be told which hash table pointer it needs to use. The point is for each thread to use a different one, so this needs to be declared somehow. Other errors could arise with this implementation that have not been initially thought of here.

Another strategy that could be used, is using process instead of threads. Processes were used in the Python code, but threads are faster, which is why they were chosen for this research. Processes can be implemented in C, but it could require a major change in the code, even more than changing the number of hash tables. Processes themselves may be difficult to implement,

but along with processes, comes the need for queues. There is not a great way to implement queues in C.

Although this code does not use threads as much as initially hoped, it does use threads. As mentioned before, this code has been run on cores, but threads can also run on GPUs (Graphics Processing Units). They have a highly parallel structure that could speed up the code if it was to be run on GPUs instead of cores. This is something that the Python code would not be able to do.

Finally, arrays could be implemented instead of hash tables. However, arrays would severely limit the possible word size. Only words of size two would be able to be put into the arrays because of memory constraints. This would cause even more compares that would need to be done and there is a much higher chance of a match between two characters than eight so scores could be high even if they do not have longer strings in common, only small pieces.

b. Future Testing

If changes were made to this code to get the timing of it closer, or better than that of the Python code, other tests should be run on it besides the overall timings that were done here. Tests should be run to determine not only the amount of time it takes the code to run, as well as include the amount of time it takes to write similar output files.

Timing changes because of the number of threads could also be run. Code run with twelve threads would be expected to run faster than the code with four threads. Tests such as this could show benefits of adding new threads and if there was an optimal number of threads.

Changes in word size should also be tested. Large words might take longer to compare if they are the same, but it would cause less comparisons to occur in the code. There could also be an optimal word size. The current default word size is eight bytes. If this was increased it would decrease the number of times the code would write to tmp_array and add for the score. However,

too large of a word size could negatively impact results. Bit errors and other small changes in a document could create a score of 0 when they actually have much more in common.

c. Continuing Research

Even if the code in this research was completed, it would not be the end of this research. The end goal of this research is to be able to identify documents in the raw image that are not in the known image. This code identifies the documents that are in both. Code will need to be created to find the docs in the raw image that do not have matches. These would be the documents that were deleted, but still left in "hidden areas." After these docs were found, research would have to be done to analyze what types of files they are and what they contain.

7. CONCLUSION

The wide variety of uses of NAND flash memory makes it an optimal way to transfer many types of data. Being able to recover memory from these flash drives may be important to forensic investigators. This study found that deleted data in erase blocks (hidden areas) of NAND flash can be recovered by obtaining a raw dump of the NAND flash memory and analyzing it against documents that are currently seen in memory. This can be done using the methods above, finding matches, and then looking at the raw image of docs that do not have matches. Recovering deleted data in these hidden areas can be done in a reasonable amount of time, shown by the previous Python code, but has not been replicated or sped up in this study's C code. Unfortunately this reasonable amount of time only applies to small chips. If chips were 8G instead of the 64MB one that was used, the time would increase exponentially because of the exponential number of compares that would have to be done. Because of the faster nature of C, it is still hoped that this code can be edited to become faster than the original Python code. The tools that are being

created in this study could be invaluable to investigators or companies that are attempting to recover deleted data on NAND flash chips.

APPENDIX

```
#include "fuzzy.h"
#include "uthash.h"
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <sys/stat.h>
#include <time.h>
#include <math.h>
#include <omp.h>

struct doc *read_file_create_hash (char *filename, struct doc *whatever_hash, int
num_docs, int doc_size)
{
	FILE *in_file = fopen(filename,"r");
	int total_words_read = 0;
	char word [WORD_SIZE];
	char buffer [doc_size];
	int num_words_in_doc = doc_size / WORD_SIZE;
	char *buf_pointer=NULL;
	char stop_word0[WORD_SIZE];
	char stop_word1[WORD_SIZE];
	memset(stop_word0,0,WORD_SIZE);
	memset(stop_word1,1,WORD_SIZE); // set stopWord arrays, to set to all 0s or all 1s
	int doc_id = 0;
	for(doc_id = 0; doc_id < num_docs; doc_id++)
	{
		fread(buffer, doc_size, 1, in_file);
		buf_pointer = buffer;
		total_words_read = 0; //reset for each doc_id
		for (total_words_read = 0; total_words_read  < num_words_in_doc;
total_words_read++)
		{
			memcpy(word,buf_pointer,WORD_SIZE); //copy the word from the buffer
				into word
			buf_pointer = buf_pointer + WORD_SIZE; //set the buffer pointer to
				read the next word
			if ((memcmp(word,stop_word0,WORD_SIZE) == 0) ||
(memcmp(word,stop_word1,WORD_SIZE) == 0))
			{
				//do  nothing for stop word (don't add to hash)
			}
			else
			{
				increment_doc_hash(&whatever_hash,word,doc_id); //all hash
					table work, increments the count
			}
		}
	}
	fclose(in_file);
	return whatever_hash;
}
```

```
void normalize(struct doc *whatever_hash, int num_docs)
{
	float total_norm = 0;
	int doc_id = 0;
	float sqrt_total=0;
	struct doc *found_doc=NULL; // our word:count dictionary for each ID
	struct word *found_word=NULL;
	struct word *tmp_word=NULL;

	//Find the hash table for doc ID 1-20
	for (doc_id = 0; doc_id  < num_docs; doc_id++) //loop through doc IDs
	{
		HASH_FIND_INT(whatever_hash,&doc_id,found_doc);
		total_norm = 0;
		if(found_doc==NULL) //There is no doc # hash entry.
		{
		}
		else
		{
			HASH_ITER(hh,found_doc->doc_hash,found_word,tmp_word)
			{
				found_word->count = 1 + log(found_word->count);
				total_norm = total_norm + pow(found_word->count,2);
			}
			sqrt_total = sqrt(total_norm);
			HASH_ITER(hh,found_doc->doc_hash,found_word,tmp_word)
			{
				found_word->count = found_word->count / sqrt_total;
			}
		}
	}
}


void compare (struct doc *known_hash, struct doc *raw_hash, struct doc *out_hash,int
rnum_docs,int knum_docs,int rdoc_size,int kdoc_size)
{
	int raw_id; //down
	int known_id; //across
	int raw_iter = 0;
	int known_iter = 0;
	int allocate_iter1 = 0;
	int allocate_iter2 = 0;
	float **output;
	output = malloc(rnum_docs*sizeof(float *));
	for (allocate_iter1=0; allocate_iter1 < rnum_docs; allocate_iter1++)
	{
		output[allocate_iter1] = malloc(knum_docs*sizeof(float));
		memset(output[allocate_iter1],0,(knum_docs*sizeof(float)));
	}

	float score = 0;
	int rnum_words_in_doc = (rdoc_size / WORD_SIZE);
	int knum_words_in_doc = (kdoc_size / WORD_SIZE);
	float **tmp_array;
	tmp_array = malloc(rnum_docs * sizeof(float *));
```

```
for (allocate_iter2=0; allocate_iter2 < rnum_docs; allocate_iter2++)
{
       tmp_array[allocate_iter2] = malloc(knum_docs*sizeof(float));
}
int i = 0;
int j = 0;
int y = 0;
int z = 0;
int match=0;

int raw_num_words = 0;
int known_num_words = 0;

char *known_word_matrix[knum_words_in_doc];
float *known_count_matrix;
known_count_matrix = malloc(knum_words_in_doc*sizeof(float));
char *raw_word_matrix[rnum_words_in_doc];
float *raw_count_matrix;
raw_count_matrix = malloc (rnum_words_in_doc*sizeof(float));

struct doc *known_found_doc = NULL;
struct word *known_found_word = NULL;
struct word *known_tmp_word = NULL;

struct doc *raw_found_doc = NULL;
struct word *raw_found_word = NULL;
struct word *raw_tmp_word = NULL;


omp_set_num_threads(4);
/* NOTE: I don't think a pragma can work here */

/* For each raw doc id */
for (raw_id = 0; raw_id < rnum_docs; raw_id++)
{

       printf("raw_id: %i \n", raw_id);

       /* For each known doc id */
       for (known_id = 0; known_id < knum_docs; known_id++)
       {

              /* If either raw/known hash doesn't exist, skip!*/
              HASH_FIND_INT(raw_hash,&raw_id,raw_found_doc);
              if (raw_found_doc == NULL){
                     continue;
              }

              HASH_FIND_INT(known_hash,&known_id,known_found_doc);
              if (known_found_doc == NULL){
                     continue;
              }
              /* Fill in both word and count matrix for raw */
              raw_iter = 0;
              HASH_ITER(hh,raw_found_doc->doc_hash,raw_found_word,raw_tmp_word)
```

```
                        {
                                raw_word_matrix[raw_iter] = raw_found_word->w;
                                raw_count_matrix[raw_iter] = raw_found_word->count;
                                raw_iter++;
                        }
                        raw_num_words= raw_iter; /*Matrix is this big */

                        /* Fill in both word and count matrix for known */
                        known_iter = 0;
                        HASH_ITER(hh,known_found_doc-
>doc_hash,known_found_word,known_tmp_word)
                        {
                                known_word_matrix[known_iter] = known_found_word->w;
                                known_count_matrix[known_iter] = known_found_word->count;
                                known_iter++;
                        }
                        known_num_words = known_iter; /*Matrix is this big */
                        {
                        #pragma omp parallel for private(j,i,match) shared(tmp_array)
                        for(i = 0; i < known_num_words; i++)
                        {
                                for (j = 0; j < raw_num_words; j++)
                                {
                                        match = memcmp (known_word_matrix[i],
raw_word_matrix[j], WORD_SIZE);
                                        if(match)
                                        {
                                                tmp_array[i][j]=0; //don't match
                                        }
                                        else
                                        {
                                                tmp_array[i][j] =
known_count_matrix[i]*raw_count_matrix[j];
                                        }
                                }
                        }
                        }
                        score = 0;
                        #pragma omp parallel private(y,z) shared(score)
                        {
                        #pragma omp for reduction(+:score)
                        for (y = 0; y < raw_num_words; y++)
                        {
                                for (z = 0; z < known_num_words; z++)
                                {
                                        score += tmp_array[y][z];
                                }
                        }
                        }
                        output[raw_id][known_id] = score;

                }
        }
        printf("OUTPUT ARRAY \n");
        FILE *outputp;
        outputp = fopen("output.csv","w+");
        match=0;
```

```
	for (i = 0; i < rnum_docs; i++)
	{
		for(j=0;j<knum_docs;j++){
			if(output[i][j]!=0)
			{
				match=1;
			}
		}

		if(match)
		{
			fprintf(outputp,"raw:%i ",i);
			for(j = 0; j < knum_docs; j++)
			{
				if(match)
					fprintf(outputp,"%f,",output[i][j]);
			}
			fprintf(outputp, "\n");
		}
		match=0;
	}
	fclose(outputp);


int main (int argc, char *argv[])
{
	clock_t begin, end;
	double time_spent;
	begin = clock();
	struct doc *raw_hash = NULL;
	int rnum_docs = 0;
	int knum_docs = 0;
	char *rfilename;
	char *kfilename;
	rfilename = argv[1];
	kfilename = argv[2];

	int rfile_size;
	int kfile_size;
	struct stat st;
	stat(rfilename, &st);
	rfile_size = st.st_size;
	stat(kfilename, &st);
	kfile_size = st.st_size;

	int rdoc_size = 528; //528, 4 for test
	int kdoc_size = 512; //512, 4 for test

	printf("rfilesize: %i \n",rfile_size);
	printf("rdoc size: %i \n",rdoc_size);
	rnum_docs = (rfile_size / rdoc_size);
	printf("rnum_docs: %i \n",rnum_docs);

	printf("kfilesize: %i \n",kfile_size);
	printf("kdoc size: %i \n",kdoc_size);
	knum_docs = (kfile_size / kdoc_size);
```

```
    printf("knum_docs: %i \n",knum_docs);

    struct doc *known_hash = NULL;
    struct doc *out_hash = NULL;

    raw_hash = read_file_create_hash(argv[1], raw_hash, rnum_docs,rdoc_size);
    known_hash = read_file_create_hash(argv[2], known_hash, knum_docs,kdoc_size);

    normalize(raw_hash,rnum_docs);
    normalize(known_hash,knum_docs);

    compare(known_hash,raw_hash,out_hash,rnum_docs,knum_docs,rdoc_size,kdoc_size);
    end = clock();
    time_spent = (end - begin) / CLOCKS_PER_SEC;
    printf("TIME: %lf \n", time_spent);

    delete_entire_doc_hash(raw_hash);
    delete_entire_doc_hash(known_hash);
}
```

LITERATURE CITED

[1] Arpaci-Dusseau, R. and Arpaci-Dusseau, A. 2015. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books.

[2] Billard, D. and Hauri, R. 2010. Making Sense of Unstructured Flash-memory Dumps. *Proceedings of the 2010 ACM Symposium on Applied Computing* (New York, NY, USA, 2010), 1579–1583.

[3] Breeuwsma, M., Jongh, M.D., Klaver, C., Knijff, R.V.D. and Roeloffs, M. *Forensic Data Recovery from Flash Memory*.

[4] Dataman 48Pro2C Super Fast Universal ISP Programmer: *http://www.dataman.com/dataman-48pro2c-super-fast-universal-isp-programmer.html*. Accessed: 2016-04-13.

[5] Diesburg, S., Feldhaus, C.A., Fardan, M.A., Schlicht, J. and Ploof, N. 2015. Is Your Data Gone? Comparing Perceived Effectiveness of Thumb Drive Deletion Methods to Actual Effectiveness. *arXiv:1512.08986 [cs]*. (Dec. 2015).

[6] Goldsborough, R. 2006. Flash Drives: Latest and Greatest Gadget. *Tech Directions*. 66, 2 (Sep. 2006), 7–7.

[7] How to Parallel Programming with OpenMP: A Quick Introduction: *http://null-byte.wonderhowto.com/how-to/parallel-programming-with-openmp-quick-introduction-0165793/*. Accessed: 2016-04-30.

[8] Manning, C.D., Raghavan, P. and Schütze, H. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

[9] Preimesberger, C. 2015. No Flash in the Pan: Flash Memory in 2015 Hotter Than Ever. *eWeek*. (Aug. 2015), 1–1.

[10] TNM ELECTRONICS 5000+: *http://www.tnmelectronics.com/English/5000.html*. Accessed: 2016-04-13.

[11] uthash: *https://troydhanson.github.io/uthash/userguide.html*. Accessed: 2016-04-11.

[12] Wong, B. 2012. The Fundamentals of Flash Memory Storage. *Electronic Design*. 60, 4 (Mar. 2012), 34.

[13] Xeltek SuperPro 6100 Universal IC Chip Device Programmer: *http://www.xeltek.com/universal-programmers/superpro-6100-universal-ic-chip-device-programmer*. Accessed: 2016-04-18.

[14] 2014. Myths: Surrounding Usb Flash Drives. *Machine Design*. 86, 4 (Apr. 2014), 96–96.