Using GPUs to Forensically Recover Non-Addressable Data on Hardware-FTL-Backed NAND Storage

S. Diesburg and M. Fienup
Computer Science Department, University of Northern Iowa, Cedar Falls, Iowa, USA

Abstract - *We present work-in-progress towards a general, parallelized method to identify leftover data in non-addressable regions of NAND flash storage devices for which the storage interface does not allow access to all the data regions (such as with thumb drives and SSDs). Significant amounts of data can wind up in non-addressable regions through regular use of the NAND flash storage that cannot be accessed through normal methods. Our method uses a combination of commodity hardware and software optimized to run in parallel on GPUs to identify these spare areas for further forensics analysis without knowing the specific layout of the chip.*

Keywords: GPU; NAND flash; data forensics

1 Introduction

NAND flash storage devices are becoming more common due to their lowering cost, temperature tolerance, small package size, and lack of moving parts. NAND flash itself is commonly used as non-volatile storage in tablets, smart phones, SD cards, solid-state drives (SSDs), and USB thumb drives. Many NAND flash devices include more storage than advertised in the form of non-addressable storage areas that cannot be accessed through software methods alone (e.g. thumb drives, SSDs, and certain SD cards). Even if a user decides to overwrite all addressable areas of the device, data can still remain in the non-addressable areas [1]. These non-addressable areas may contain sensitive data that could be valuable to storage forensics operations. Once that data is recovered, it could then be processed further to reconstruct files.

This paper presents a work-in-progress towards a framework that uses inexpensive, commodity methods to recover data from non-addressable areas on the NAND TSOP-48 chip commonly found in current SSDs and thumb drives. Since used thumb drives are inexpensive, easy to purchase, and contain the same underlying NAND TSOP-48 flash package, we use a set of used thumb drives purchased in a prior study to test our framework.

Our framework consists of the following two components: 1) hardware tools necessary to desolder the NAND flash chip from the storage device as well as universal flash programmers to read all the storage areas from the raw NAND flash chip, and 2) software written for parallel GPU execution to compare data areas found on the intact storage device to data areas found on the raw NAND flash chip. By comparing and matching the two sets of data, any data found to be unmatched is determined to be the non-addressable data.

The software component must be written carefully to consider a number of potential pitfalls, including bit-flips/ECC errors, corruption, duplicated blocks of data, and the physical placement of non-data areas commonly found within the flash chip. Parallelism must be achieved to make the software efficient. The final goal of our framework is to provide a storage forensics investigator with a method to extract more potential data from SSDs or thumb drives after traditional forensics methods are applied without requiring her to be a NAND flash expert or know the specific layout of the chip in question.

Our paper is organized as follows. Section 2 provides background in NAND flash and related work; Section 3 outlines hardware and software methodology; Section 4 gives early results; and Section 5 discusses current work, future work and conclusions.

2 Background and related work

2.1 NAND flash background

NAND flash is a type of memory storage with an interface that accepts read, write, and erase requests [2]. Very generally, storage on NAND flash is divided into flash blocks, which are further divided into flash pages. These flash pages are further divided into data areas and flash-specific metadata areas which we call out-of-bound areas. (See Fig 1)

Data is read and written at the flash page granularity, but once a page is written, further updates are not allowed in the flash page data area until the entire block on which the flash page resides on is erased with a flash erase operation. This erase operation not only causes other valid flash page data to be copied out of the current block to avoid destruction, but it is also very slow. In addition, each flash block can only be erased a small number of times, so the erasures and therefore the wear must be evenly spread across the storage to avoid wearing out the device prematurely.

These challenges are solved through the addition of a ***flash translation layer (FTL)***. The FTL is located in hardware controllers (in the case of SSDs and thumb drives), and it exports a typical read/write hard drive interface instead of the read/write/erase flash interface described above. To avoid excess slow erasure operations, when flash receives a request to update or overwrite a flash page, the FTL remaps the write to a pre-erased flash page using a translation table, and marks the old page as invalid in the page out-of-bounds area to be erased later. This old page cannot be accessed through the exported interface, but will stay on the chip until block erasure time occurs (if it ever occurs).

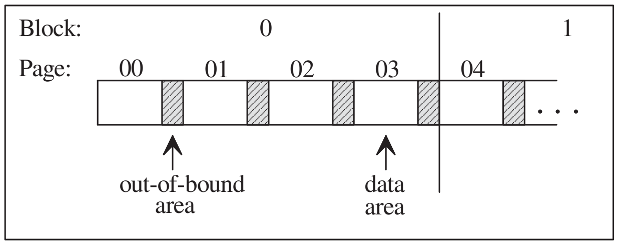

Fig. 1. Division of NAND flash data storage areas.

To assist in these FTL algorithms, NAND flash drives have extra non-addressable physical locations that may hold information on old pages. Some of that information may be sensitive or important [3]. One study found NAND flash to hold as much as 6%–25% of additional unreported storage area [4]. Due to the proprietary nature of the FTL algorithms, it is not known exactly when those pages will be erased.

2.2 Related work

Previous work has shown that information on both invalid and valid pages could be read by removing the NAND flash chip and placing it in a universal reader [5]. However, the primary goal of this method is to re-construct the original high-level file system. Billard et al [1] devises a scheme to compare intact images from devices with the extracted flash chip image. However, this method uses specific file system and file structure knowledge, does not handle duplicates data, does not handle errors and bit-flips, and does not discuss placement of out-of-bounds data. Other researchers have created specialty FPGA boards [6] to read a specific NAND TSOP-48 chip, but such a solution needs to be customized for each chip from each manufacturer and requires FPGA knowledge. Alternatively, the Netherlands Forensics Institute supplies a NFI Memory Toolkit [7] to recover potentially erased data from NAND flash devices. Unfortunately, the toolkit is pricey and the Netherlands Forensics Institute only delivers services to other governmental organizations.

3 Methodology

3.1 Initial steps

Step 1) Obtain devices and create USB image: For this study, we used 120 used USB thumb drives from eBay and Amazon Marketplace purchased for a prior study that investigates user deletion behavior [8]. These drives were purchased from a wide variety of sellers in hopes of increasing drive diversity. In both marketplaces, our primary differentiating factor was cost (cheapness) and small size of drive (mostly under 8GB). Once the drives were verified to work, we created binary images of the intact devices using the linux dd command. We call these images the ***USB images***.

Step 2) Desolder chips: After removing the protective covering of the USB thumb drive, we desoldered the NAND TSOP-48 chips from the boards using a hot air gun.

Step 3) Obtaining chip images: We purchased two universal flash programmers: TNM [9] and Dataman [10]. along with various TSOP-48 chip adapters. The primary use of a flash programmer is to write information to a flash chip to be used in an embedded device; however, each programmer also has read functionality to ***verify*** that the program (write) operation worked successfully. We only used the read functionality to create binary images of each chip, which we call the ***chip images***. These images contained both addressable and non-addressable data areas as well as out-of-bounds areas and data. These areas are typically not marked on the images. The images may also contain errors.

3.2 Software algorithm

The goal of our software is to ultimately output a list of non-addressable (invalid) pages found on the chip image that would contain data for forensics purposes. Technically, these invalid pages should be marked as such somewhere in the page out-of-bounds area. However, such markings are proprietary and potentially different with each chip and each controller. Therefore, we take the longer approach of matching every addressable page on the USB image to unique pages on the chip image. When finished, the non-matched pages on the chip image will be the non-addressable (invalid) pages.

We implemented and tested our software on a host machine with a 6-core Intel Xeon X5670 2.93GHz CPU and 22 GB memory with three NVIDIA Tesla C2070 GPU cards.

Several complicating factors must be considered when trying to match a USB-image page to a chip-image page:

- We would like to just skip the out-of-bounds areas to simplify the search. Unfortunately, we have empirically found that the out-of-bounds area in the chip image may not stay in the same position across flash planes, so we cannot assume where chip-image page out-of-bound areas begin and end
- Bit-flips and corruption of data areas on the chip prevent using a hash table of USB-image pages to match potential chip-image pages.

We first implemented a base-line brute-force algorithm on the GPU that compares every USB-image page to every possible chip page by sliding a page-size window down the whole of the chip image. This allowed us to find the true optimal USB-image to chip-image page match, and provided base-line timing data on our Tesla C2070 GPU cards.

The brute-force program utilized the GPU as a 1D grid of thread blocks with threads within a block also being 1D. Each block is responsible for matching a single USB-image page to the optimal page-size chunk across the whole chip image. For 4 GB or less USB flash drives, we were able to copy the entire chip image to the GPU's global memory. The USB-image pages are copied in grid-size chunks before performing the GPU kernel to optimally match these USB-image pages to the whole chip image.

A noteworthy feature of the GPU kernel is that it minimizes the GPU's global-memory to shared-memory bandwidth usage. The kernel block initially loads the USB-image page it is responsible for once and the first page-size bytes of the chip-image into shared-memory. After matching the first page-size bytes of the chip-image, only the next byte of the chip-image from global-memory is read into shared-memory replacing the byte dropping out of the page-size sliding window. (See Fig 2)

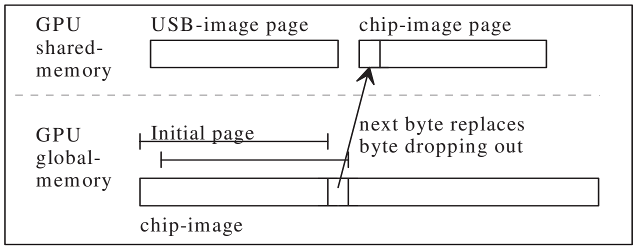

Fig. 2. Single byte read to slide window down chip-image.

Another noteworthy feature of the GPU kernel occurs when matching a USB-image page with a chip-image page to determine the number of byte matches. Each thread in a block is responsible for comparing a subset of bytes (e.g., each thread does 2 byte comparisons if the blockDim.x = 1024 and page-size is 2048) and counting the number of matching bytes. Instead of using a CUDA `atomicAdd` to a shared-memory `sum` variable, it is faster for thread *i* to place its count at index *i* of a `threadCounts` array in shared memory. After all threads are done matching their subset of bytes (i.e., `_syncthreads()`), a binary-tree reduction of the `threadCounts` array is performed with `threadCounts[0]` receiving the sum of matches between these pages. Thread 0 of each block is responsible for maintaining the byte offset within the whole chip image producing the optimal USB-image page match.

4 Preliminary results

Timings of the brute-force GPU implementation on our Tesla C2070 card using data from our smallest 16 MB USB flash drive required 15.78 hours. This is not surprizing since the brute-force implement performs $i \times c$ byte comparisons, where i is the number of bytes in the USB image and c is the number of bytes in the chip image. The details for this 16 MB USB drive are: 512 bytes per page, 16 bytes per OOB data, USB-image size 15,908,864 bytes, and chip-image size 17,301,504 bytes.

We made an easy optimization to the brute-force implementation by observing that out-of-bounds data on all the chip images we have analyzed is always a multiple of 8 bytes. Without loss of generality, this allowed us to slide the page-size window down the chip-image in 8 byte increments instead of a single byte. As expected processing of this 16 MB USB flash image dropped from 15.78 hours down to 2.10 hours for a speedup of 7.51. Table I shows estimated timings of this optimized brute-force implementation on "full" USB flash drives of several common sizes.

Table I. Estimated run-time of optimized brute-force on various USB drive sizes

USB Drive Size	Number of Hours Estimated
16 MB	2.10 hours
1 GB	8,602 hours (358.4 days)
4 GB	137,625.6 hours (15.7 years)

In our experience, flash drives are often not full so the USB image contains a large percentage of all-zero-bit pages. Throwing these all-zero-bit pages out greatly speeds processing, but clearly we cannot depend on flash drives not being full so we need a less computationally intensive algorithm than our brute-force approach.

5 Current and future work

5.1 Optimizations

Currently we perform some preprocessing of the USB-image and chip-image data similar to the BLAST (Basic Local Alignment Search Tool) algorithm used in Bioinformatics [11]. The general steps of this preprocessing are:

- Split USB-image pages into non-overlapping smaller size "words" (say 32-bytes in length). This set of USB-image words is used as keys to populate a hash table, we call this the ***USB-word table***.
- Process the chip image sequentially checking if word-size pieces are in the USB-word table, we call these word-size piece ***chip-words.*** If a chip-word is found in the USB-word table, then the chip-word's offset within the chip file is stored in the list of offsets associated with the corresponding USB-word table key. (See Fig. 3)

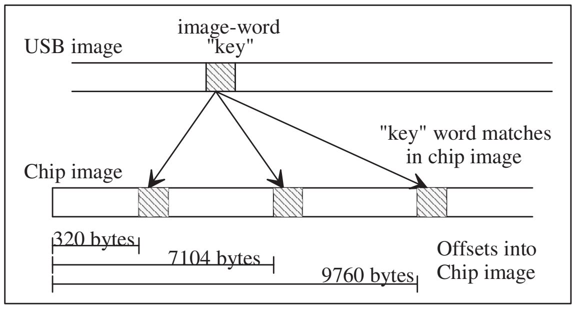

Fig. 3. USB-word table entry "key" : [320, 7104, 9760]

- Utilize the USB-word table information to drastically reduce the computation performed on the GPU when matching full USB-image and chip-image pages. The USB-word table information will show where 32-byte words in a USB-image page match in the chip-image. Most of the time all the words in a USB-image page will map to a whole page-size chunk of the chip-image, so we have found an exact page match. Thus, the GPU does not need to consider this USB-image page further. Occasionally, a chip-image page which should map to a USB-image page might contain bit-flips, so only some 32-byte words will match exactly. If enough multiple words in a USB-image page map to the same relative positions in page-size chip-image chunks, we can consider matching on a bit-level using the GPU. (See Fig. 4)

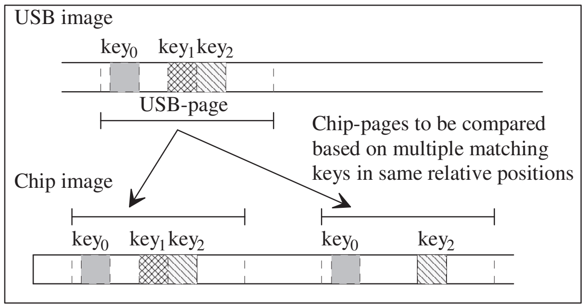
USB image
key$_0$
key$_1$
key$_2$
USB-page
Chip-pages to be compared based on multiple matching keys in same relative positions
Chip image
key$_0$
key$_1$
key$_2$
key$_0$
key$_2$

Fig. 4. Multiple USB-word table keys used to find potential matches

We currently have a sequential Python program which performs the above preprocessing steps to build and utilize the USB-word table. Table II shows the execution time for several USB flash drive sizes. For USB flash drives of greater than 1 GB, our current program thrashes because the hash table size grows too large. Part of our future work will be to solve the problem by processing the USB image in stages and combining their results. The USB-word table needs to store the keys, but the combined result only needs to store the integer offset in the USB-image where the key is located as well as the list of integer offsets in the chip-image corresponding to the location of key matches – the key is no longer needed. Additionally, we plan on porting this Python program to C and execute it on the host computer containing the GPU cards.

Table II. Time to build USB-word table on various USB flash drive sizes

USB Drive Size	**Time to Build USB-word Table**
16 MB	23.2 seconds
128 MB	170.0 seconds
1 GB	1546.4 seconds

After preprocessing, only USB-image pages without exact matches to chip-image pages need to be consider by the GPU kernel. The GPU kernel will assign each grid block a USB-image page and a list of chip-image offsets which matched multiple word keys during preprocessing. Unlike the brute-force algorithm, only these chip-image page-size chunks must be compared to the USB-image. The GPU kernel will populate an array in the global-memory containing the number of bit-wise matches for each chip-image offset, we call this array the ***bitwise-matches***.

After the GPU kernel executes, we copy the bitwise-matches back to the host computer. Post-processing of the bitwise-matches must be combined with the pre-processing full-page matches. The overall goal of this post-processing is to identify the non-matched pages on the chip image which corresponds to the non-addressable (invalid) pages on the USB flash drive.

5.2 Resilience to bit-flips and corruption

Our approach is moderately resilient to bit-flips within a chip-image page as discussed previously with Fig. 4. If bit-flips occur in every 32-byte word across the whole page, then no USB-word table matches are found, but we expect this to be a rare occurrence. We can experiment with a smaller word size, say 16-bytes, which would make this less likely.

5.3 Next steps

Once the invalid/non-addressable pages are identified by our software, the pages can be feed through common file carvers and data-identification software to re-create files and/or identify snippets of interesting data.

6 Acknowledgment

We would like to acknowledge Jonathan Schlicht for his work in desoldering and categorizing our flash chips, as well as early work in extracting data from the chips. We would also like to thank Lisa Diercks and Russel Rook for their early work in the parallelization software to extract non-addressable data.

7 References

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

[2] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating systems: Three easy pieces*, vol. 151. Arpaci-Dusseau Books Wisconsin, 2014.

[3] S. M. Diesburg, "Ghosts of Deletions Past: New Secure Deletion Challenges and Solutions," *ArXiv161104216 Cs*, Nov. 2016.

[4] M. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, "Reliably erasing data from flash-based solid state drives," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, Berkeley, CA, USA, 2011, pp. 8–8.

[5] M. Breeuwsma, M. De Jongh, C. Klaver, R. Van Der Knijff, and M. Roeloffs, "Forensic data recovery from flash memory," *Small Scale Digit. Device Forensics J.*, vol. 1, no. 1, pp. 1–17, 2007.

[6] T. Bunker, M. Wei, and S. J. Swanson, *Ming II: A flexible platform for nand flash-based research*. Department of Computer Science and Engineering, University of California, San Diego, 2012.

[7] M. van V. en Justitie, "NFI Memory Toolkit," 19-Aug-2016. [Online]. Available: https://www.forensicinstitute.nl/products_and_services/forensic_products/nfi-memory-toolkit.aspx. [Accessed: 10-Apr-2017].

[8] S. Diesburg, C. A. Feldhaus, M. Al Fardan, J. Schlicht, and N. Ploof, "Is Your Data Gone? Measuring User Perceptions of Deletion," in *ACM Socio-Technical Aspects in Security and Trust Workshop*, Los Angeles, California USA, 2016, p. 13.

[9] "TNM5000 Universal Programmer." [Online]. Available: http://www.tnmelectronics.com/English/5000.html. [Accessed: 10-Apr-2017].

[10] "Dataman 48Pro2C Super Fast Universal ISP Programmer." [Online]. Available: http://www.dataman.com/dataman-48pro2c-super-fast-universal-isp-programmer.html. [Accessed: 10-Apr-2017].

[11] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *J. Mol. Biol.*, vol. 215, no. 3, pp. 403–410, Oct. 1990.