Image forensics

NOTES

- Change this to recover PDF documents for *important business purposes*.

TL;DR

Implement a program to recover lost photos from a corrupted SD card.

Learning objectives

This exercise is designed to help you learn about (and assess whether you have learned about): * how to understand a problem. * how to write an algorithm for solving a problem. * how to write a simple program in Java using multiple classes and file i/o.

Lastly, this exercise will give you a better understanding and appreciation for filesystems and photo storage.

Background

Just the other day, I took a stroll around campus with a friend (Dan Armendariz of MIT, whose skills with a camera outshine my point-and-shoot tendencies) snapping photos, all of which were stored as JPEGs on a 1GB CompactFlash (CF) card. Rather than act like typical tourists, taking photos of John Harvard's foot (ugh) and squirrels (I mean, really), we opted to shoot identifiable but non-obvious persons, places, and things on campus.

Unfortunately, I somehow corrupted that CF card the moment I got home. Both my Mac and PC refuse to recognize the card now as having any photos, even though I'm pretty sure we took them. Both operating systems want to format the card, but, thus far, I've refused to let them, hoping instead someone can come to the rescue.

Write a program called recover that recovers these photos. (Please!)

OMG, what?

Well, here's the thing: JPEGs have "*signatures*", patterns of bytes that distinguish them from other file formats. A list of common file signatures can be found here. In fact, most JPEGs begin with one of two sequences of bytes. Specifically, the first four bytes of most JPEGs are either

0xff 0xd8 0xff 0xe0

or

0xff 0xd8 0xff 0xe1

from first byte to fourth byte, left to right. Odds are, if you find one of these patterns of bytes on a disk known to store photos (e.g., my CF card), they demark the start of a JPEG.

Fortunately, digital cameras tend to store photographs contiguously on CF cards, whereby each photo is stored immediately after the previously taken photo. Accordingly, the start of a JPEG usually demarks the end of another. However, digital

cameras generally initialize CF cards with a FAT file system whose block size is 512 bytes (B). The implication is that these cameras only write to those cards in units of 512 B. A photo that's 1 MB (i.e., 1,048,576 B) thus takes up $1048576 \div 512 = 2048$ blocks on a CF card. But so does a photo that's, say, one byte smaller (i.e., 1,048,575 B)! The wasted space on disk is called slack space. Forensic investigators often look at slack space for remnants of suspicious data.

The implication of all these details is that you, the investigator, can probably write a program that iterates over a copy of my CF card, looking for JPEGs' signatures. Each time you find a signature, you can open a new file for writing and start filling that file with bytes from my CF card, closing that file only once you encounter another signature. Moreover, rather than read my CF card's bytes one at a time, you can read 512 of them at a time into a buffer for efficiency's sake. Thanks to FAT, you can trust that JPEGs' signatures will be "block-aligned." That is, you need only look for those signatures in a block's first four bytes.

Realize, of course, that JPEGs can span contiguous blocks. Otherwise, no JPEG could be larger than 512 B. But the last byte of a JPEG might not fall at the very end of a block. Recall the possibility of slack space. Fortunately, I bought a brand-new CF card for my stroll about campus. Odds are, that CF card was "zeroed" (i.e., filled with 0s) by the manufacturer. Because I didn't outright delete any photos we took, the only bits on that CF card should belong to actual photos or be 0s. And it's okay if some trailing 0s (i.e., slack space) end up in the JPEGs your program spits out; they should still be viewable.

Since I've but one CF card, I've gone ahead and created a "forensic image" of the card, storing its contents, byte after byte, in a file called card.raw. So that you don't waste time iterating over millions of 0s unnecessarily, I've only imaged the first few MB of the CF card. Since you're only going to be reading it, you don't need your own copy of this forensic image. (Might as well save space!) Simply open our copy with fopen, as in the below.

```
FILE *fp = fopen("card.raw", "r");
```

You should find that this image contains a whole bunch of JPEGs.

It's up to you to create, at least, a Makefile and recover.c for this program. You probably don't need a recover.h, but you're welcome to create one. For simplicity, you may hard-code the path to card.raw in your program; your program need not accept any command-line arguments. When executed, though, your program should recover every one of the JPEGs from card.raw, storing each as a separate file in your current working directory. Your program should number the files it outputs by naming each ###.jpg, where ### is three-digit decimal number from 000 on up. (Befriend sprintf.) You need not try to recover the JPEGs' original names. To check whether the JPEGs your program spit out are correct, simply SFTP them to your own desktop, double-click, and take a look. If each photo appears intact, your operation was likely a success!

Odds are, though, the JPEGs that the first draft of your code spits out won't be correct. (If you open them up and don't see anything, they're probably not correct!) Execute the command below to delete all JPEGs in your current working directory.

```
rm *.jpg
```

If you'd rather not be prompted to confirm each deletion, execute the command below instead.

```
rm -f *.jpg
```

Just be careful with that -f switch, as it "forces" deletion.

The section whose students collectively identify the most photographs shall win an amazing prize. In the event of a tie, the section that submits the most photos first shall be declared the winner!

FAQs

None so far. Reload this page periodically to check if any arise.

Attribution