

Testing Forensic Hash Tools on Sparse Files

Harish Daiya

IIT Kharagpur, India

Maximillian Dornseif

Hudora GmbH, Germany

Felix C. Freiling

Laboratory of Dependable Distributed Systems

University of Mannheim, Germany

Abstract: Forensic hash tools are usually used to prove and protect the integrity of digital evidence: When a file is intercepted by law enforcement, a cryptographic fingerprint is taken by using a forensic hash tool. If later in a court of law the identical fingerprint can be computed from the presented evidence, the evidence is taken to be original. In this paper we demonstrate that most of the freely available forensic hash tools fail to support this conclusion at the file system level for *sparse files*, a particular class of files in Unix systems that contain holes. We describe an experimental setup by which existing and future hash tools can be easily tested for this border case. In conclusion, we argue that further efforts are necessary to test and validate common forensic hash tools so that the significance of their results can be better judged.

1 Introduction

Computer forensics deals with analysis and investigation of computer systems using scientific methods to test whether or not the system was used for unauthorized activities. One of the first steps in forensic investigations is the preservation of digital evidence. If this evidence comes in the form of a file system on a magnetic disk, *forensic imaging* is one of the first steps performed. Forensic imaging is the process in which exact copies of file systems are made. Forensic analysis can then be performed on the copy of the data and not on the original to preserve the integrity of the evidence. To be valid in a court of law, examiners must argue that the copy is identical to the original evidence. This is done today by using *forensic hash tools*, i.e. tools based on cryptographic hash functions. Roughly speaking, such tools compute a unique and unforgeable fingerprint of a file system. File systems with identical content should have the same fingerprint while different file systems should have a distinct fingerprint.

Since evidence collected during a forensic investigation can be used in a court of law, it is mandatory to have high confidence in the correctness of such forensic hash tools. One way to increase this confidence is to test the forensic tools under stress and using border cases as inputs. Such border cases should contain incomplete inputs, malformed file structures, corrupted directory structures etc. Only if the tools behave predictably and

in their intended manner in all these cases should they be acceptable in court.

1.1 Previous Work on Forensic Tool Testing

The *Computer Forensics Tool Testing Project* (CFTT) [10] is a joint project of the United States' National Institute of Justice, National Institute of Standards and Technology, and other agencies, such as the Technical Support Working Group. The objective of the CFTT project test the accuracy of the computer forensic tools and increase the confidence in them so that they can be used by the forensic community and legal organizations during digital investigations. The results of the tests done as a part of the project also provide useful information to the tool makers to improve the tools.

The activities of forensic investigations under CFTT is divided into categories, such as hard disk write protection, disk imaging, string searching, etc. They then develop a test methodology for each category and select a tool in each category. The test results are posted on NIJ's website. Disk imaging tools like dd, SafeBack, EnCase have been tested. Write block tools like RCMP HDL and PDBLOCK have also been tested. Currently the project is working on developing a test methodology for testing deleted file recovery tools.

Apart from the CFTT project, the *Digital Forensics Tool Testing Project* (DFTT) [3] has developed several small test cases for testing various forensic tools. The test images for all the tested tools are available on the website. Some such test cases include the keyword search test for Ext3, Fat and NTFS file systems. This test focuses on searching an ASCII string in a file. The goal is to identify which tool can find different types of strings in a file. For example, a string can cross between end of a file into the slack space of the file. Some tools can identify this others might not.

Forensic hash tools have neither been tested by DFTT nor has been any such test been posted on CFTT.

1.2 Contributions

In this paper we report on experiences in investigating forensic hash tools on the border case of a very simple and common file structure: *sparse files* which are also known as *files with holes*. Briefly spoken, sparse files result from using the `lseek` system call in Unix operating systems. Using hash tools in the context of sparse files can result in surprising results, as we now explain.

The experimental setup was as follows. On a given file system we created two files: The first file `withhole` was a sparse file that contained a hole, i.e., the beginning and the end of the file contained a fixed bitstring while the transition from beginning to end was performed using the `lseek` system call. The second file `withzero` contained the same bitstrings at beginning and end while the intermediate part of the file was filled with zeros. Obviously, these two files have different representations on the raw hard disc while many

operating systems will treat both files similarly at the system call interface, so it is not immediately clear how forensic hash tools should treat them. We tested a variety of forensic hash tools (`md5sum`, `md5deep`, `shasum`, `shaldeep`, `tigerdeep`, `sha256deep`, `whirlpooldeep`) on these files for a variety of file system types (`ext3`, `reiserfs`, `vfat`, `jfs`, `minix`, `ext2`, `msdos`). In summary all of the hash tools computed the same hash for both files. However, the file size of the two files (as reported by invoking the Unix command `du`) was different for all file systems except `vfat`, `minix` and `msdos`.

At first sight the above results may lead to the conclusion that all of the common forensic hash tools have problems when confronted with sparse files on most file systems. However, at second sight the problem is more complex: As pointed out by Carrier [2, p. 10f], data analysis can be performed at different levels of abstraction. For example, data can be analyzed directly on the *physical storage medium* (the raw hard disk sectors), within a *volume* (a collection of sectors accessible to an application), within a *file system* (a collection of data structures that allow an application to create, read, and write files). Forensic imaging, which is where hash tools are commonly used, is performed at the physical storage medium level or the volume level. Sparse files are a feature supported at the file system level and so the results of our study do *not* directly affect standard practices of digital forensic investigations. However, we feel that investigators should be aware of the pitfalls which arise from using the *same tools* on *different analysis layers*.

The set of sample file system images is publicly available for download [4].

1.3 Paper Outline

The paper is structured as follows. We first recall the principles of cryptographic hash functions and forensic hash tools in Section 2. We then explain the concept of sparse files in Section 3. Section 4 reports on the experiments we performed using forensic hash tools on different file systems. Finally, Section 5 summarizes our results and concludes the paper.

2 Hash Functions

A cryptographic hash function is a mathematical function which satisfies certain properties to make it suitable for use as primitive in various information security applications, such as authentication and message integrity. In this section we recall these special properties, explain their importance in forensic investigations and enumerate the most relevant hash algorithms used in practice.

2.1 Properties of Hash Functions

A cryptographic hash function H maps an arbitrary size input string x to a fixed size output string y . The output y is usually called the *message digest* or *fingerprint* of x .

The security properties which H must satisfy are as follows:

- H must be a *one way function*, i.e., for a given hash value y it is computationally infeasible to find a message $y' \neq y$ such that $H(x) = H(y)$.
- H must be *collision free*, i.e., it is computationally infeasible to find two different messages x and x' such that $H(x) = H(x')$.

2.2 Usefulness of Hash Functions

Since the input of a hash function can be of arbitrary length and the output is of fixed length, in principle there must exist distinct files that have the same fingerprint. However, because of the security properties of the hash function it is very hard to find such two files. Therefore it can be safely assumed that distinct files result in different fingerprints.

One of the most important purposes of hash functions is to check the integrity of files and file systems. For this, a hashing tool that implements the hash function initially computes the fingerprint of a file or file system. This fingerprint is recorded in a place where integrity can be assured (e.g. the investigator's paper notebook or a printed report). In case the integrity of the original files is questioned, an investigator or analyst can run the same hash tool on the files. In case the fingerprint is equal to the one initially recorded, the evidence is original with very high probability. The alternative to using hash functions is to store a complete copy of the original files in a protected place. Especially for large data sources this is very inconvenient.

2.3 Hash Functions and Tools

We now give an overview over the most common hash functions in use today.

MD5 The MD5 (Message Digest 5) family of hash functions was developed by Rivest [5]. The output size of MD5 is 128 bits. Many tools implement the MD5 algorithm, e.g., `md5sum`, `md5deep`. In contrast to `md5sum`, the tool `md5deep` can create MD5 hashes of whole directory trees and is extensively used for forensic purposes

SHA The SHA (Secure Hash Algorithm) was designed by the National Security Agency (NSA) and published as US government standard [6, 7]. There are several different instantiations of SHA, e.g., SHA-0, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512. The tools used were `sha1sum`, `sha1deep` and `sha256deep`.

TIGER The Tiger family of hash functions was designed by Anderson and Beham in 1995 [8]. This family can produce hashes of lengths 128 bits, 160 bits and 192 bits. The tool used to produce the message digest is called `tigerdeep`.

WHIRLPOOL The Whirlpool family was developed by Rijmen and Barreto [9]. The hash is 512 bits in length. We used the tool `whirlpooldeep` to produce the message digest.

3 Sparse Files

We now recall the concept of sparse files. A file is sparse if it has unallocated blocks i.e holes. The reported size of such files is always larger than the actual disk space consumed by them. The Unix disk utility `du` (disk usage) by default gives the real disk space consumed by a sparse file (i.e. the space used by disk blocks), while `du` with the option `--apparent-size` gives the size of a file after reading it through the system call interface. The holes are always read as zeros. All major Linux filesystems like `ext3/2`, `JFS`, `reiserfs`, etc. supports sparse files. But some file systems like `ISO 9600 CD-ROM` filesystem does not. Later in this paper we will see a program that can be used to create a sparse file.

Overall we can say that it is easy to create files with holes, but it is difficult to distinguish them at the system call interface by reading and comparing them. If read, the holes will be treated as zeros. Using the Unix utility `du` it is always possible to find whether a file is sparse or not. In this sense, sparse files are a hybrid concept: meant to be transparent at the system call interface, but distinguishable using other means.

4 Testing Forensic Hash Tools on Sparse Files

We now describe our experiments. First we explain the experimental setup, i.e., how we created different file systems and how we created test files (sparse and non-sparse). Then we enumerate the test results for different hash tools and different file systems.

4.1 Experimental Setup

All the analysis and tests are done on separate file system images of different types. To set up these images, we used the Unix `dd` command to create a disk partition within a file. Then we formatted the (partition) file to establish a file system of a particular type. Using the loopback device option of the `mount` command it is then possible to mount the file system like a regular disk partition into the directory tree and to create a number of test files.

A common sequence of the necessary commands looks like this:

- Create a 10 MB image containing zeros:

```
# dd if=/dev/zero of=image bs=1M count=10
```

- Format it as for e.g. the `ext3` file system:

```
# mkfs.ext3 image
```

- Mount the image using the loopback device:

```
# mount -o loop image /mnt
```

We created file system images for the following file system types: `ext3`, `reiserfs`, `vfat`, `jfs`, `minix`, `ext2`, `msdos`. On these file systems we created two different but similar files:

- A sparse file named `withhole` containing the following data:
 - The first 10 bytes of the file contain the string `beg_string`.
 - Then a fixed number n of bytes is skipped using the `seek` system call. In our experiments we used $n = 100000$.
 - The final 10 bytes of the file contain the string `end_string`.
- A non-sparse file named `withzero` containing the following data:
 - The first 10 bytes contain the string `beg_string`.
 - Then n null characters are written to the file (ASCII value 0).
 - The final 10 bytes of the file contain the string `end_string`.

We used the following program in Figure 1 to create both files.

Counting the number of bytes and skipped byte positions, both files have exactly the same length. Our aim now is to see the behaviour of hash tools while dealing these two files.

4.2 Testing of Tools

All the mentioned hash tools (`md5sum`, `md5deep`, `sha1sum`, `sha1deep`, `tigerdeep`, `sha256deep`, `whirlpooldeep`) were tested by comparing the hash or message digest produced by them for the two created files. The message digest of the two files was exactly the same for all the tools. We also tested whether the two files were distinguishable at the

```

#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<string.h>
#include<sys/stat.h>
#include<fcntl.h>

/* set up variables */
#define beg_str    "beg_string"
#define end_str    "end_string"
#define withhole  "withhole"
#define withzero  "withzero"
#define n          100000          /* size of hole */

int main()
{
    int offset=n, fd1, fd2, i;
    char *buff1 = beg_str;
    char buff2[offset];
    char *buff3 = end_str;

    fd1=creat(withhole, 0);
    fd2=creat(withzero, 0);

    /* create common header */
    write(fd1,buff1,10);
    write(fd2,buff1,10);

    for(i=0;i<offset;i++)
        buff2[i]=0;
    lseek(fd1,offset,SEEK_CUR);
    write(fd2,buff2,offset);

    /* create common trailer */
    write(fd1,buff3,10);
    write(fd2,buff3,10);
    return(0);
}

```

Figure 1: Program to create files used in experiment.

level of the file system. In most cases this was the case, i.e., the disk usage of the two files was different.

To test the disk usage of the two files, we used the Unix command `du` which shows the number of bytes allocated for the file at the file system level. This can give different results for sparse and non-sparse files as shown below. There is a special option `--apparent-size` of `du` that shows the apparent size of a file, i.e., a possibly larger value due to holes, internal fragmentation, indirect blocks etc. For example, a 3KB file with holes in it with `du --apparent size` will show the size as 98KB which is same if these holes were read as zeros. As explained before in many file systems there is no special provision for reading a hole. Whenever it is read it is interpreted as zero.

4.3 Test Cases

We now enumerate the investigated file systems. The hash values for the different hash tools as well as the MD5 hash of the complete file systems are given in the summary below.

Ext3 This test image is a raw partition image of an Ext3 file system. The size of the image is 16MB. For this file system it was possible to distinguish the two files from their file size.

Reiserfs This test image is again a raw partition image of a Reiser file system. For this file system it was possible to distinguish the two files from their file size.

vfat The test image is raw partition image of a FAT file system. The size of the image is 11MB. In contrast to the first two file systems, it was *not* possible to distinguish the file sizes using `du`.

JFS The test image is raw partition image of a JFS file system. The size of image is 17MB. For this file system it was possible to distinguish the two files from their file size.

Minix The test image is raw partition image of a Minix file system. The size of image is 5.1MB. For this file system it was possible to distinguish the two files from their file size.

Ext2 This test image is raw partition image of an Ext2 file system. The size of the image is 11MB. For this file system it was possible to distinguish the two files from their file size.

Msdos The test image is a raw partition image of a MS-Dos FAT file system. The size of the image is 5.1MB. Like the vfat file system, here it was *not* possible to distinguish the two files using `du`.

filesystem	img size	du		du -app size		msg digest identical
		withhole	withzero	withhole	withzero	
ext3	16MB	3KB	99KB	98KB	98KB	yes
reiserfs	76MB	8KB	100KB	98KB	98KB	yes
vfat	11MB	98KB	98KB	98KB	98KB	yes
jfs	17MB	8KB	100KB	98KB	98KB	yes
minix	5.1MB	99KB	99KB	98KB	98KB	yes
ext2	11MB	3KB	100KB	98KB	98KB	yes
msdos	5.1MB	98KB	98KB	98KB	98KB	yes

Table 1: Results of applying the tested tools on the two files.

hashtool	hash
md5sum	458b5ebc8c1bf7cacc4684e67eabb409
md5deep	458b5ebc8c1bf7cacc4684e67eabb409
sha1sum	6beb9d5bfe512fdf34aa33f0c258a72caeb64995
sha1deep	6beb9d5bfe512fdf34aa33f0c258a72caeb64995
tigerdeep	21aca239cefd99d2f441eb3dd45768989617582925158971
sha256deep	7bce318f4ce2833a02decbcde475c0b1164d1557567e55cc004f883276811d21
whirlpooldeep	d9ae3e0342558c1f1fc0eb5d8fcfcd97a4c932ddd869cba141d77367594660fb fefb292a895d97174757bbd85a1befe5d1d8e018b0f5d3bb0ceae05ded02f2f9

Table 2: Hash values for both files using different hash tools.

4.4 Summary and Discussion

Tables 1 and 2 show a summary of the results of the experiment done. It should be noted that the disk usage in vfat, minix and msdos file systems is the same. This is because these file systems do not seem to support sparse files. Overall the apparant sizes of the two files for all the file systems is the same and so is the message digest for all the hash tools. Table 3 show the reference values computed by all hash tools.

4.5 Attack Scenario

Can the observed behavior of hash tools be exploited by an attacker? One scenario we can envision is this context is an insider who wants to steal a large confidential file by copying it to a relatively small removable storage medium. The attacker prepares the file with holes and is able to copy it to the storage device and steal it. When accused in court of stealing the file, the attacker can claim that it is impossible to copy the file to the storage medium because of its size. Since only the hashes have been recorded, it is questionable whether the attacker is telling the truth or the evidence has been tampered with.

A second attack scenario occurs in the context of denial-of-service attacks. An attacker

File System	md5sum Hash
Ext3	e4eb6cc8f026f676ee7b1249251905bf
Reiserfs	07c7e4afbc788799b83f398614e09d49
vfat	4e32ebb9784a01c176a7f371e3a680c4
JFS	c7ac4f517683b4ee5dde89a54346cc1d
Minix	76a8da6e046ee1690a26212c13f5d100
Ext2	ed3f984761861d24fbaeab1c6b476d40
Msdos	56324b6d97230650e29a09d5c4dd6501

Table 3: File system hashes for different file systems.

may prepare a large file and place it on a disk device to consume sufficient disk space to cause service disruption. Later, when accused of denial-of-service, the attacker may claim of having placed a sparse file which consumed less disk space. Since only the hashes have been recorded, it is questionable whether the attacker is telling the truth or evidence has been tampered with.

5 Conclusion

In this work we investigated how forensic hash tools perform on sparse files (i.e., files with holes) in comparison to non-sparse files. We therefore analyzed the tools for this border case on different file systems. All tools gave the same hash for the two created files even if the disk usage of the two files was different. This is mainly because the hole in the sparse file when read by the tools is interpreted as a sequence of zero bytes. The results of the tests are as follows:

- All the tools resulted in same message digests in all the file systems.
- Disk usage was different in Ext2, Ext3, Reiser and Jfs file systems.
- Disk usage was exactly same in vfat, minix and msdos file system.
- The apparent sizes of the files however was same for all the file systems.

This points to a weak point in the usage of hash tools today if they are used on different analysis layers within the same investigation. If used on the file system layer, two files which have different physical representations on the disk had the same message digest. These differences become apparent if *the same* hash tools are used on the volume layer. The caveat here is that even on the file system layer (i.e., using `du`), the two files with identical hashes can be distinguished (at least in many file systems). This opens an applicability gap for hash tools and a need for a convincing argument in case the integrity of evidence is challenged in court.

Overall we feel that current tools should at least warn the user about such an inconsistency if they are used at the file system level. This is possible since the file sizes can be observed

using the `du` command. Overall we argue that all common hash tools need to be modified for dealing correctly with sparse files.

Acknowledgments

We wish to thank the anonymous reviewers for helpful feedback. In particular we acknowledge the comments by reviewer #3 who pointed out the difference of using hash tools at different levels of abstraction (i.e., on volumes and on file systems).

References

- [1] Dan Farmer and Wietse Venema: *Forensic Discovery*, Addison Wesley Professional, 2005.
- [2] Brian Carrier: *File System Forensic Analysis*, Addison-Wesley Professional, 2005.
- [3] Brian Carrier: *Digital Forensics Tool Testing Images*, <http://www.dftt.sourceforge.net>, last visited: Dec 2006.
- [4] Harish Daiya: *Sparse file testing images*. Available online at pi1.informatik.uni-mannheim.de/filepool/projects/hash-tool-testing/images.zip
- [5] Ronald Rivest: *The MD5 Message-Digest Algorithm*, RFC 1321, April 1992.
- [6] D. Eastlake, T. Hansen: *US Secure Hash Algorithms (SHA and HMAC-SHA)*, RFC 4364, July 2006.
- [7] D. Eastlake: *US Secure Hash Algorithm 1 (SHA1)*, RFC 3174, September 2001.
- [8] Ross Anderson and Eli Biham: *Tiger - A Fast New Hash Function*, *Proceedings of Fast Software Encryption 3*, Cambridge, 1996.
- [9] P.S.L.M. Barreto, V. Rijmen: *The Whirlpool Hashing Function*, *First Open NESSIE Workshop*, Leuven, 13-14 November 2000.
- [10] National Institute of Standards and Technology: *The Computer Forensics Tool Testing (CFTT) Project Homepage*, <http://www.cftt.nist.gov>, last visited: December 2006.