APPLICATION OF MESSAGE DIGESTS FOR THE VERIFICATION OF LOGICAL FORENSIC DATA

Pontjho M. Mokhonoana[1], Martin S. Olivier[2]

[1]Information and Computer Security Architectures Research Group
Department of Computer Science
South Africa

[1]pontjho@tuks.co.za, [2]martin@mo.co.za

ABSTRACT

A message digest is a fixed length output produced by applying a cryptographic algorithm on input binary data of arbitrary length. If the input data changes even by one bit, the generated message digest will be completely different from the original. This is used in digital investigations to verify that stored digital evidence has not been tampered with.
This technique has been applied successfully on physical disk images because there is only one continuous stream of data. However, this is not applicable to logical disk images where there is no obvious or standard method of concatenating the data to produce an output message digest. This paper describes the difficulties that complicate the computation of a message digest for logical data. In addition, a candidate process for calculating a verification value for computer forensic evidence for logical data, regardless of its underlying representation is given. This method is presented in the context of cellphone forensics.

KEY WORDS

Computer Forensics, Verification, Cryptographic Hash, Message Digest

APPLICATION OF MESSAGE DIGESTS FOR THE VERIFICATION OF LOGICAL FORENSIC DATA

1 INTRODUCTION

Current best practices for dealing with digital evidence advocate that when evidence is acquired, the first to the last bit of the data on the device be copied to create a physical disk image [Jansen and Ayers, 2006]. This has the advantage of allowing the recovery of deleted or partially overwritten data. The physical image also facilitates the simplicity of the process of the computation of the associated message digest of that image since the one way hash is based on a single stream of binary data.

It is however not always possible to obtain a physical image of a device's data; this is usually the case when extracting data from Small Scale Digital Devices [Harrill and Mislan, 2007] such as cellular phones or a Subscriber Identity Module (SIM) card. In other cases, it is preferable to obtain a logical image when only the logical data is required. Creating a physical image makes very little sense since that would add unnecessary additional processing and storage overheads.

When looking at logical data, the picture gets a little more complicated since a simple reordering of the logical data items will affect the produced hash. This complexity of verifying logical data makes it difficult to verify data retrieved from mobile devices using most of the acquisition methods [Mokhonoana and Olivier, 2007], since they produce a logical image.

For example, a phone book entry will consist of the name of the person, their phone number and usually other contact details such as the email or secondary number. There is no defined order in which such items should be read meaning that if a hash value were to be computed on that data, the output produced by different tools on the same content may be different depending on the order in which the data was read.

To give an idea of how the logical image from a smartphone would look like, figure 1 is given below. Each of the contact entries could in turn have the name, surname, office number, mobile number, email as well as other attributes.

One way to get around this is to compute the hash for each data item in the logical image. Such a solution has a number of drawbacks:

- It would complicate the process of verifying the authenticity of the log-

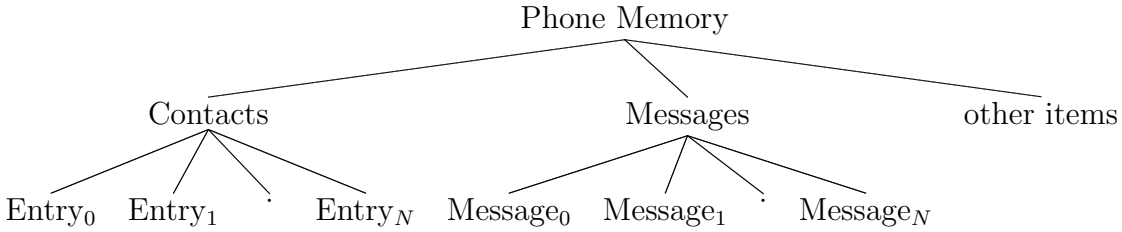

Figure 1: Sample Data From a Mobile Phone

ical image since it would increase the algorithmic complexity of comparing the hashes.

- If there is a large number of data items in the logical image, the hashes could take up a large amount of disk space.
- It does not take into account the attributes of the logical data items. If it does, then there is the question of the ordering and representation of the attributes.

To solve the above mentioned problems, the Sorted Vector Hashing (SVH) is porposed in this paper. The goal of the algorithm is to enable the computation of a hash value based on the logical content of a file rather than just its binary data. This will facilitate the comparison of the content of different data items such as an email, even though they are stored in different mail formats.

The aim of this paper is to present a method for producing a message digest for a logical image file that also takes into account the attributes of the data items and the potentially different representations of the attributes. The method achieves the goal by allowing the message digest computation to be performed on items at any granular level and putting it together with a simple algorithm which will be discussed in more detail later in the paper.

2 RELATED WORK

Most of the computer forensics tools employ the use of hashes to verify their images. However for logical data, different tools use different schemes to do the verification. *EnCase* [Guidance Software, 2008a] uses a proprietary L01 format for the storage of logical data. For that reason, it is difficult to determine how its verification works.

The *Forensic Toolkit* [Guidance Software, 2008b], which uses FTK Imager to create images employs the use of AD1 image containers to store logical data. The format does not have built-in mechanisms for the verification. Instead, another file (called a hash list) with the hashes of all the individual files is created. The hashes are only based on the filestream content and not their associated attributes.

TULP 2G [van den Bos and van der Knijff, 2005] is an open source tool for acquiring and decoding data from electronic devices. It uses an XML format for the storage of case related data. A sample of the data is given in figure 2. To calculate the hash of the case items or the entire case, the formulas in figure 3 are used.

In these formulas, the hash of an item is computed by concatenating its metadata with the string content. The hash is then performed on that resultant output. A similar process is used to compute the hash of the investigation and that of the case.

```
<Case Name="SampleCase" Creator="…" DateCreated="…" DateModified="…" MD5="…" SHA1="…">
    <Notes>…</Notes>
    <Item Name="SampleItemA" DateCreated="…" DataType="…" StorageType="…" ItemType="…" MD5="…" SHA1="…">
        …
    </Item>
    <Investigation Name="SampleInvestigation1" Creator="SampleAuthor" DateCreated="…" MD5="…" SHA1="…">
        <Notes>…</Notes>
        <Item Name="SampleItem1" DateCreated="…" DataType="…" StorageType="…" ItemType="…" MD5="…" SHA1="…">
            …
        </Item>
        <Item Name="SampleItem2" DateCreated="…" DataType="…" StorageType="…" ItemType="…" MD5="…" SHA1="…">
            …
        </Item>
        …
    </Investigation>
    <Investigation Name="SampleInvestigation2" Creator="…" DateCreated="…" MD5="…" SHA1="…">
        …
    </Investigation>
    <Item Name="SampleItem3" DateCreated="…" DataType="…" StorageType="…" ItemType="…" MD5="…" SHA1="…">
        …
    </Item>
    …
</Case>
```

Figure 2: Tulp 2G Data Format

The problem with the L01 format is the lack of documentation around how it works. That makes it difficult for other tools to implement and verify the results produced by a tool. FTK's hash list is quite simple to verify in that it is a plaintext file which could be read and verified by a human. The first problem is that it only considers the file data. If the file's metadata, such as creation or modification date, is altered, that change will not be detected. The second problem is that because the hash of every single file has to be

$$Hash(Item) = Hash(Name + DateCreated + DataType + StorageType + ItemType + StringContent)$$

$$Hash(Investigation) = Hash\left(Name + Creator + DateCreated + Notes + \sum_{i=1}^{\#Items} Hash(Item[i])\right)$$

$$Hash(Case) = Hash(Name + Creator + DateCreated + DateModified + Notes + \sum_{i=1}^{\#ItemChildenOfCase} Hash(Item[i]) + \sum_{j=1}^{\#Investigations} Hash(Investigation[j]))$$

Figure 3: Calculation of Hash

specified, the hash list gets large very quickly, making it difficult for human verification. The Tulp2G XML format is an improvement on the above in that it is open and takes all attributes into account when calculating the hash. However, it is designed for a specific application and is not flexible enough to handle other applications. If other attributes were introduced, it would not be able to handle them without changing the algorithm used to calculate the hash. In addition, it does not address the ordering problem discussed above.

3 SORTED VECTOR HASHING

A logical image will consist of the following: Item, Attributes and Children. The item is a logical entry in the image. For example on a logical filesystem image, the item could be a file. The attribute will be the filename, date or owner. The children could be other files (if this file is a folder) or data streams in the file. Most file systems allow a single child in the latter vase (but note that NTFS can have multiple streams).

The way that a hash is calculated depends on what you are trying to verify. For example when trying to calculate the hash of a folder, the order of the children is not important. Actually, from a logical perspective, there is no universally defined way of sorting the contents of a folder. That is, the following should hold for a folder f, with files $f1$, $f2$ and $f3$: $h(f) = h(f1, f2, f3) = h(f2, f1, f3) = ... = h(f3, f2, f1)$. In this text, this is referred to as commutativity. However when calculating the hash of the attributes, the order is important. For example a file will have a creation, modification and access date. Now even though there also is no obvious order in which the hash must be applied to the attributes, it must be applied

uniformly to yield the same output. That is: if $a = \{a1, a2, a3\}$ is the set of attributes then $h(a) = h(a1, a2, a3)$, but

$$h(a1, a2, a3) \neq h(a2, a1, a3) \ldots \neq h(a3, a2, a1)$$

The process for performing the hash must be able to cater for the two scenarios above, one where the order is not important and the other where it is. To achieve this, the Sorted Vector Hashing (SVH) algorithm was conceived and its aim is to generate a single hash from a number of logical items. The algorithm assumes that the sort order is not important for children but is for attributes.

To compute the hash on the data represented in the tree on figure 1, one would start at the terminal nodes in the tree and work one's way up to the root. Each node in the tree has zero or more attributes and children. Stated formally:

$$I = (\langle a_1, a_2, \ldots, a_m \rangle \langle c_1, c_2, \ldots, c_n \rangle) \tag{1}$$

I represents an arbitrary node in the tree. a_i and c_i represent an arbitrary attribute and child of I respectively. Note that the inner text content of a node is treated as an attribute.

The ordering of the attributes must be preserved. For that reason computing the hash(*H(x)*) of the attributes is straightforward since it only involves concatenating the hashes of the respective attributes and then calculating the hash over the resultant output. This is referred to as Unsorted Vector(UV). Therefore the hash of the attributes is:

$$H(I_a) = UV(I_a) = H(\biguplus_{i=1}^{m} H(a_i)) \tag{2}$$

where $\biguplus$ is used to denote concatenation.

The ordering of the children however is not important and as a result the oder in which the happen to be must not affect the output hash. To achieve this, the child hashes are computed and then the hashes are sorted. It is only after that that the resultant output is concatenated and hashed. To state this formally, assume $H(c_i) = y_i$. Then, construct a sorted vector, $\langle z_1, \ldots, z_n \rangle$, such that

$$\forall y_i \exists z_j \text{ where } z_j = y_i \tag{3}$$

and

$$(\forall z_i, i < n) z_i \leq z_{i+1} \tag{4}$$

$$\text{Then } H(I_c) = SV(I_c) = H(\biguplus_{j=1}^{n} z_i) \tag{5}$$

To get the hash of I, the hash of its attributes is concatenated with that of the children and then rehashed. That is:

$$H(I) = UV(H(I_a) \biguplus H(I_c)) \tag{6}$$

Note that equation 6 is applied recursively to the nodes in the tree starting from the root. To compute the hash of the child, the equation is applied where the child would then be I.

4 EVALUATION AND FUTURE WORK

To evaluate the algorithm, it was tested against a number of sample XML documents where a node represents an item. The XML attributes represent the item's attributes and the child elements the item's children. If the algorithm works, it must fulfil the following requirements:

1. The names of the tags should not affect the output
2. The ordering of the attribute tags should affect the output
3. The ordering of the child elements should not affect the output
4. The values of the child or attribute nodes should affect the output

A number of these sample XML documents were used to test if the algorithm behaved as it should. These are given in figures 4, 5, 6 and 7. The hash values for the documents are summarised in the tables 1 and 2. In the following paragraphs, the results from applying the algorithm are discussed. It is also described how the requirements stated above are satisfied by the algorithm.

The algorithm described in the previous section does not take into account the names when computing the output. Therefore the first requirement is obviously satisfied.

```
<?xml version="1.0"?>
<image date="16/03/2008" examiner="Jack Sparrow">
    <files>
        <file path="C:/multimedia/file1.jpg" create_date="7 Jan 2006" modified_date="8 Jan 2007">
            abc...xyz
        </file>
        <file path="C:/multimedia/file2.jpg" create_date="10 Jan 2008" modified_date="11 Jan 2008">
            zyx...cba
        </file>
    </files>
    <phonebook>
        <contact name="John" lastname="Doe" email="" fax="011 111 1123" />
        <contact name="Mary" lastname="Doe" email="mary@doe.com" fax="011 112 1123" />
    </phonebook>
</image>
```

Figure 4: Sample 1

```
<?xml version="1.0"?>
<image date="16/03/2008" examiner="Jack Sparrow">
    <files>
        <file filepath="C:/multimedia/file2.jpg" create_date="10 Jan 2008" modified_date="11 Jan 2008">
            zyx...cba
        </file>
        <file filepath="C:/multimedia/file1.jpg" create_date="7 Jan 2006" modified_date="8 Jan 2007">
            abc...xyz
        </file>
    </files>
    <phonebook>
        <entry firstname="Mary" lastname="Doe" email="mary@doe.com" fax="011 112 1123" />
        <entry firstname="John" lastname="Doe" email="" fax="011 111 1123" />
    </phonebook>
</image>
```

Figure 5: Sample 2

In sample 4, the order of the attributes is changed. Since in this paper the names of the tags are not considered, the ordering of the attributes is important in order to preserve the semantic meaning of the attributes. Therefore, sample 4 would be considered different from 1. Since the hash is different, it satisfies the second requirement.

Sample 1 and sample 2 are supposed to represent the same logical data. The only differences are the ordering of the children of the contact and file node as well as the names of some of the tags. The reader's attention is drawn to the fact that their hashes are exactly the same. This is consistent with the third requirement that the ordering of the child elements should not affect the output.

In sample 3 the value of the contact node is changed. The name, lastname and email attribute values are changed to *"Jack"*, *"Johnson"* and *""* respectively. For that reason, the hash produced is different, thus satisfying requirement 4.

```
<?xml version="1.0"?>
<image date="16/03/2008" examiner="Jack Sparrow">
    <files>
        <file path="C:/multimedia/file1.jpg" create_date="7 Jan 2006" modified_date="8 Jan 2007">
            abc...xyz
        </file>
        <file path="C:/multimedia/file2.jpg" create_date="10 Jan 2008" modified_date="11 Jan 2008">
            zyx...cba
        </file>
    </files>
    <phonebook>
        <contact name="Jack" lastname="Johnson" email="" fax="011 111 1123" />
        <contact name="Mary" lastname="Doe" email="mary@doe.com" fax="011 112 1123" />
    </phonebook>
</image>
```

Figure 6: Sample 3

```
<?xml version="1.0"?>
<image date="16/03/2008" examiner="Jack Sparrow">
    <files>
        <file path="C:/multimedia/file1.jpg" create_date="7 Jan 2006" modified_date="8 Jan 2007">
            abc...xyz
        </file>
        <file path="C:/multimedia/file2.jpg" create_date="10 Jan 2008" modified_date="11 Jan 2008">
            zyx...cba
        </file>
    </files>
    <phonebook>
        <contact name="Jack" email="" fax="011 111 1123" lastname="Johnson" />
        <contact name="Mary" email="mary@doe.com" fax="011 112 1123" lastname="Doe" />
    </phonebook>
</image>
```

Figure 7: Sample 4

From the analysis of the hashes, one can conclude that the algorithm works as expected — two logical items must produce the same hash if and only if their content is the same.

5 CONCLUSION

In this paper, the challenges of calculating a hash for logical items are discussed. The current tools which handle logical data are analysed and their limitations discussed. The biggest contribution of this paper is the introduction of a method for verifying logical data that overcomes some of the limitations of the other tools. The method was tested on a small dataset and the results confirm that it works.

The work presented is an important aspect of the research into mobile forensics since it enables reliable verification and comparison of logical data from potentially different sources. It is hoped that this will contribute to a

Sample	*MD*5
1	85 19 86 F2 24 B2 70 EA F4 3B F8 93 F7 E6 79 71
2	85 19 86 F2 24 B2 70 EA F4 3B F8 93 F7 E6 79 71
3	78 77 A6 E5 FF C6 33 B4 A8 F7 82 AC 73 D9 9F EE
4	96 E1 6F FD 78 05 E8 5B F8 5A AD 05 90 63 81 BB

Table 1: MD5 Hash Summary

Sample	*SHA*1
1	17 04 13 47 FA 0E 1A A4 C0 CC A4 B6 0A 50 90 F9 4F 35 1C D3
2	17 04 13 47 FA 0E 1A A4 C0 CC A4 B6 0A 50 90 F9 4F 35 1C D3
3	7B 5F B0 8F 83 EC 07 61 68 6D DC 6E B9 BD 09 8A 05 11 E9 A9
4	0A 89 2D 95 0B D7 D5 51 5C 8F 2F B5 5B F9 89 50 06 24 7B FE

Table 2: SHA1 Hash Summary

standardisation of verification methods in computer forensics.

References

[Guidance Software, 2008a] Guidance Software (2008a). Encase. www.guidancesoftware.com.

[Guidance Software, 2008b] Guidance Software (2008b). Forensic toolkit. www.accessdata.com.

[Harrill and Mislan, 2007] Harrill, D. C. and Mislan, R. P. (2007). A small scale digital device forensics ontology. *Small Scale Digital Forensics Journal*, 1(1):1–7.

[Jansen and Ayers, 2006] Jansen, W. and Ayers, R. (2006). Guidelines on cell phone forensics. Technical report, National Institution of Standards and Technology.

[Mokhonoana and Olivier, 2007] Mokhonoana, P. and Olivier, M. S. (2007). Acquisition of a symbian smart phones content with an on-phone forensic tool. In *Proceedings of the Southern African Telecommunication Networks and Applications Conference 2007 (SATNAC 2007)*, Mauritius.

[van den Bos and van der Knijff, 2005] van den Bos, J. and van der Knijff, R. (2005). TULP2G – an open source forensic software framework for acquiring and decoding data stored in electronic devices. *International Journal of Digital Evidence*, 4(2).

P. M. Mokhonoana and M. S. Olivier, “Application of message digests for the verification of logical forensic data,” in *Proceedings of the ISSA 2008 Innovative Minds Conference*, H. S. Venter, M. M. Eloff, J. H. P. Eloff, and L. Labuschagne (eds.), Johannesburg, South Africa, July 2008. (Published electronically)..

Source: http://mo.co.za