Digital Investigation 20 (2017) S107–S115

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

DFRWS 2017 Europe — Proceedings of the Fourth Annual DFRWS Europe

Characterizing loss of digital evidence due to abstraction layers

Felix Freiling [a, *], Thomas Glanzmann [b, **], Hans P. Reiser [c, ***]

[a] Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Martensstr. 3, 91058 Erlangen, Germany
[b] Rathsberger Str. 28, 91054 Erlangen, Germany
[c] Faculty of Computer Science and Mathematics, University of Passau, Innstraße 43, 94032 Passau, Germany

ARTICLE INFO

Article history:
Received 26 January 2017
Accepted 26 January 2017

Keywords:
Abstraction layer
Virtualization
Forensic analysis
File system
Virtual memory

ABSTRACT

We study the problem of evidence collection in environments where abstraction layers are used to organize data storage. Based on a formal model, the problem of evidence collection is defined as the task to reconstruct high-level from low-level storage. We investigate the conditions under which different levels of evidence collection can be performed and show that abstraction layers, in general, make it harder to acquire evidence. We illustrate our findings by describing several practical scenarios from file systems, memory management, and disk volume management.

Introduction

Abstraction layers are a ubiquitous concept in modern computer systems. Like in computer networks (Tanenbaum, 2002) or file systems (Carrier, 2005), they usually come as a hierarchy where resources on a "higher" layer of abstraction are mapped to a set of resources on a "lower" layer of abstraction. Abstraction layers usually come with the benefit of isolation and more efficient resource usage stemming from separation of concerns.

When accessing upper layers, however, details of lower layers are "hidden". This observation is relevant to the process of evidence collection for forensic purposes: It is usually easier to establish the authenticity of digital evidence when it is accessed at lower layers of abstraction, but the items of interest are at higher levels of abstraction. Therefore, the items need to be reconstructed taking one abstraction layer at a time. But every abstraction layer introduces ambiguity and the possibility of interpretation errors (Casey, 2011). Even worse, sometimes it is impossible to access the lower layer because it is physically out of reach, such as in many cloud computing environments (Roussev et al., 2016).

Loss of digital evidence due to abstraction layers

Given modern systems with complex hierarchies of abstraction layers, we ask the following question: Under which conditions is it possible to reconstruct higher levels of abstraction from data given at lower layers of abstraction? This question may appear trivial at first sight since all evidence that exists at the higher abstraction layer *must* also exist on the lower layer. However, from a digital forensics point of view we not only target actively used data but also deleted or otherwise unused data. This includes fragments of deleted files or the remains of process data structures in unused parts of physical memory. Not being able to interpret the data at the lower layer means that—at least from the viewpoint of the forensic expert—such data is "lost" through abstraction. Furthermore, there are many examples where abstraction layers combined with concurrency (as in the case of virtualization) may change data at the lower layer in ways that do not exist without abstraction. One example is resource pooling in cloud data centers: disk space that is not used by one virtual machine may be re-allocated to another virtual machine and overwritten, which may destroy evidence at higher layers. Our aim, therefore, is to better understand the different ways in which digital evidence is lost due to abstraction layers.

Related work

The concept of abstraction underlies most techniques to build computer systems. Many early software and hardware engineering techniques are based on functional (or code) abstraction, as for example in structured programming with stepwise refinement. Later, data abstraction became a major paradigm for the construction of software systems, for example in object-oriented programming or other types of current formal modeling languages.

* Corresponding author.
** Corresponding author.
*** Corresponding author.
E-mail addresses: felix.freiling@cs.fau.de (F. Freiling), thomas@glanzmann.de (T. Glanzmann), hans.reiser@uni-passau.de (H.P. Reiser).

http://dx.doi.org/10.1016/j.diin.2017.01.012

Abstraction layers are not only a basic principle of constructing computer systems but also to reason about them. This is apparent in the area of (formal) verification where abstraction layers have been investigated in great depth. Formally, an abstraction layer relates to two system descriptions, one at a higher layer and one at a lower layer. Proving that the lower layer "implements" the higher layer, requires to map the state space (and therefore also the executions) of the lower layer to the higher layer. This can be done by constructing a *refinement mapping* (Abadi and Lamport, 1991) between the two layers.

The relations between "primitive" and "complex" event sequences were also influential to early attempts to formalize (digital) investigative techniques. For example, Carrier introduces the idea of constructing analysis tools along the abstraction levels of computer systems (Carrier, 2003). Later, Carrier and Spafford introduce *abstraction transformation functions* (Carrier and Spafford, 2006, p. S126) that allow the formulation high-level hypotheses and mapping them to low-level system operations. This is done to structure the area of investigative techniques and not to study the eventual loss of evidence through abstraction.

As early as 2010 it was observed that abstraction in form of virtualization poses problems to incident handling and forensics (Grobauer and Schreck, 2010). Similarly, Martini and Choo (2012) embed these issues into a process model framework for the forensic analysis of cloud computing platforms.

Birk and Wegener (2011) and later Kolb (2012) raised many detailed technical issues in the context of virtual machine analysis, especially the problem of correct resource attribution under resource pooling. Dykstra and Sherman (2012) investigated the problems of recovering deleted files or processes from memory in cloud computing environments. Zawoad and Hasan (2016a, 2016b) take a different approach and design an environment to aid in the forensic analysis of cloud systems.

Roussev and McCulley (2016) recently characterized the reason for problems in cloud forensic analysis, namely *cloud-native artifacts*. These artifacts encompass evidence that can only be acquired by physical access to the server, thereby pointing to a fundamental limitation of evidence collection in virtualized environments. While it is well-known that evidence collection on higher layers for multiple reasons is usually inferior to collection at lower layers, little research has been done to characterize the information loss that results from having abstraction layers at all.

Contributions

We approach the research question using a formal model and derive conditions describing certain types of evidence collection for different conditions. The model covers systems that use abstraction layers to organize block-based storage. Based on our model, we derive conditions enabling the reconstruction of active and inactive data at lower layers. We can show, that these conditions are only partly satisfied by real data structures and that therefore data is "lost" through abstraction.

Our model formalizes page-based or block-based storage abstractions that may be fragmented or not. API-based storage abstractions like network file systems are excluded, because they hide state information. A further issue of the paper concerns reconstructing *state* instead of computation. The term *storage* is used to describe content of volatile physical memory as well as persistent physical disks alike, i.e., main memory and disks do not have to be distinguished. Our formalization covers a wide range of relevant data organization techniques like file systems, virtual memory, guest physical memory, logical volumes, and RAID systems.

Paper outline

We present our model of abstraction layers in Section "Model", definitions of different evidence collection and reconstruction tasks in Section "Forensic reconstruction problems", the illustration of these classes using practical examples in Section "Abstraction layers in practice", and, finally, our conclusions in Section "Conclusions".

Model

Layers

We consider a single level of abstraction at a time, and for each level we distinguish two layers, the *upper* layer and the *lower* layer. Each layer provides computation and storage resources. However, the upper layer is implemented using resources from the lower layer.

Storage

We consider a block-based storage abstraction (storage can be either memory or disk). Both layers consist of a finite ordered sequence of storage blocks, but blocks can be of different sizes and sequences can have different lengths in both layers. More precisely, on the lower layer, we model storage as a finite ordered sequence of storage blocks $l[0],l[1],\ldots,l[n]$, and on the upper layer we model storage similarly as a finite ordered sequence of storage blocks $u[0],u[1],\ldots,u[m]$. Each storage block is identified by a unique index, where L indicates the range of indexes into l, i.e., the set $\{0,\ldots,n\}$, and U the range of indexes into u, i.e., the set $\{0,\ldots,m\}$. Note that m must not necessarily be equal to n, and that our model can be used to "simulate" multiple block sequences on both upper and lower layers.

Mappings

Between upper and lower layers exists an entity that manages the lower storage so that upper storage is available at the upper layer. This entity uses a management data structure to map the set of upper storage blocks to lower storage blocks over time. In our model, this structure can map any subset of blocks in u to any subset of blocks in l. Since the mapping is time dependent, we need to model its evolution over time. For simplicity, we use the set of natural numbers as time domain, i.e., $T = \{0,1,2,\ldots\}$. We then formally model the mapping at a certain time $t \in T$ as a relation $\varphi_t \subseteq U \times L$ that associates elements of u to elements of l, i.e., $(x,y) \in \varphi_t$ iff $u[x]$ is mapped to $l[y]$ at a certain time t.

The definitions are illustrated in Fig. 1 where the assignment of blocks from u to blocks from l are depicted at a certain point in time t. Accordingly, at time t, upper layer block 1 ($u[1]$) is mapped to lower layer block 1 ($l[1]$). Furthermore, upper layer blocks $u[2]$ and $u[5]$ are both mapped to lower layer block $l[4]$, and $u[7]$ is mapped to $l[8]$ and $l[11]$.

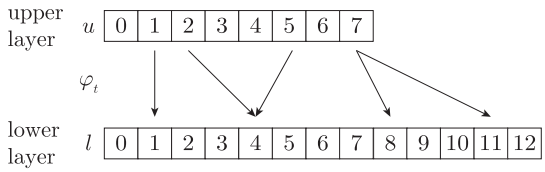

Fig. 1. Model of storage on different abstraction layers and mappings between layers at time t.

For terminological simplicity, we say that an element of u is "mapped to" one or more elements of l, or that the element is a "mapping" to elements of l, i.e., x is a mapping to y at time t iff $(x,y) \in \varphi_t$. Although the element is called a mapping, note that the *value* or the *content* of an element is always contained in the lower blocks. For example, in Fig. 1, the content of $u[7]$ can be found in the lower blocks $l[8]$ and $l[11]$.

Mappings over time

As mentioned above, mappings are dynamic and change over time. For example, the static state depicted in Fig. 1 at time t may change by new mappings being added or existing mappings removed at time $t+1$. Because such changes are of forensic interest, we introduce special terminology to describe them.

If an upper block leaves the mapping at time t, then we say that the upper storage block has been *unmapped* from the lower storage block at time t. More formally, *upper block x has been unmapped to lower block y at time t* iff $(x,y) \in \varphi_{t-1}$ and $(x,y) \notin \varphi_t$. If a block has been unmapped, we also say that the mapping from the upper storage block to the lower storage block has been removed.

If an upper block enters the mapping at time t, then we say that the upper block has been *mapped* to the lower block at time t. More formally, *upper block x has been mapped to lower block y at time t* iff $(x,y) \notin \varphi_{t-1}$ and $(x,y) \in \varphi_t$. If a block is mapped, we also say that a mapping from the upper block to the lower block has been assigned at time t.

To illustrate the terms, consider Fig. 2 in which a sequence of changes of φ_t is shown for $t = 0,\ldots,4$. At $t = 1$, upper blocks 0 ($u[0]$) and 1 ($u[1]$) are mapped to lower block 0 ($l[0]$). At $t = 2$, $u[1]$ is unmapped from $l[0]$ and mapped to $l[1]$. At $t = 3$, $u[1]$ is unmapped from $l[1]$ and $u[0]$ is mapped to $l[1]$. At $t = 4$, $u[0]$ is unmapped from both $l[0]$ and $l[1]$.

Unmapped blocks

In digital forensics, it is of interest to find out what mappings existed in the past. In this context, unmapped blocks are of special interest.

Intuitively, an unmapped block exists if in the past an upper block has been unmapped from a lower block. Therefore, depending on the sequence of mappings and unmappings, for any lower level block there exists a history of deleted mappings. Informally, a deleted mapping from x to y is a previous mapping from x to y that has been unmapped in the meantime. More formally, we say that *x is a deleted mapping to y at time t* iff there exists a point in time $t' \leq t$ such that x has been unmapped from y at time t'.

As an example, consider Fig. 3 where deleted mappings are shown as dashed arrows. At time $t = 2$, we have one deleted mapping from $u[0]$ to $l[1]$ since $u[0]$ was unmapped from $l[1]$ at that time. Note that our definition of deleted mappings takes the *entire history* of φ into account. This leads to situations where mappings may have been multiply assigned and deleted, as in $t = 5$ where $u[0]$ previously assigned to $l[1]$ has been mapped again to $l[0]$. This

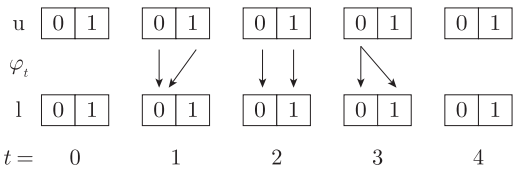

Fig. 2. Sequence of changes of φ_t over time.

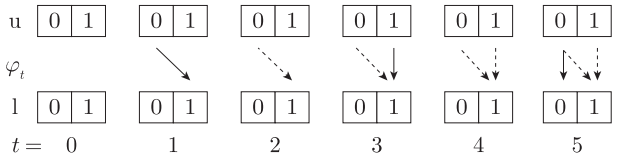

Fig. 3. Sequence of changes of φ_t over time. Deleted mappings are shown as dashed arrows.

leads to a monotonic increase in the set of deleted mappings over time. In the end, i.e., at $t = 5$, the set of deleted mappings consists of the mappings (0,1) and (1,1).

Most recently unmapped blocks

The set of deleted mappings is "flat" in the sense that the elements are not qualified regarding any time they were deleted. So while mappings can be deleted several times, for every deleted mapping a unique *most recent deletion time* can be defined, i.e., the most recent deletion time of the mapping from x to y is the largest t' such that x was unmapped to y at time t'. This time (and therefore the most recently deleted mapping) is of forensic interest since the content of storage blocks resides in the lower layer and deleted mappings may still exist in the upper layer. Using the most recently deleted mapping, it might still be possible to interpret (and recover) the content of the lower layer in the context of the upper layer.

We define the set of *most recently deleted mappings for lower layer block y at time t* the set of all upper layer blocks x that were most recently unmapped from y.

Fig. 3 shows a sequence of mappings and unmappings for times $t = 0,\ldots,5$. In the end, i.e., at time $t = 5$, both $u[0]$ and $u[1]$ are deleted mappings to $l[1]$ but $u[1]$ was most recently deleted.

Mapping aware lower layers

The content of unused blocks at the lower layer is not interesting from the functionality viewpoint of the upper layer: The fact that an upper layer block is unmapped from a lower layer block essentially means that the content of the lower layer block is no longer relevant for the functioning of the upper layer.

Still, such an unmapped block may contain evidence, as long as it has not been mapped again to another upper layer block. But as long as it is unmapped, the lower layer can in principle use the storage in other ways, e.g., to store φ. Thus, the storage on the lower layer can be utilized more efficiently. In practice, SSDs can use the IDE command TRIM (ATA/ATAPI Command Set - 2 (ACS-2), 2011) or the SCSI command UNMAP (SCSI Block Commands - 3 (SBC-3), 2014) to optimize their internal data organization. This spares the lower layer from having to "understand" the mapping of file system blocks to physical hard disk blocks. If there is an explicit way to communicate the active/inactive status of storage blocks to a lower layer we call this layer *mapping aware*.

If the lower layer does not reuse unmapped blocks for other purposes than storage, there is no need for the lower layer to "understand" the semantics of φ in order to work correctly. For example, the standard HDD physical block device is often unaware of whether one of its blocks is in use by the file system or not. So if there is no mechanism to communicate the active/inactive status to the lower layer, then we call such a lower layer *mapping agnostic*.

Evidence collection

Depending on the level of access, evidence collection can be performed at the upper or the lower layer. It corresponds to the usual habits of forensic evidence collectors to strive for evidence collection at the lowest layer necessary to capture all relevant information. Evidence collection at the lowest layer implies making a bit-by-bit copy of the sequence of lower storage blocks l to a target storage area outside of the model.

Evidence collection can also be performed at the upper layer. In this case, it is performed by making a bit-by-bit copy of the sequence of upper storage blocks u to a target storage area.

Forensic reconstruction problems

Let us consider a case where an investigator has performed evidence collection in the sense described above, i.e., the investigator has acquired a bit-by-bit copy of all lower layer storage blocks. The task is now to reconstruct the upper layer storage and to extract as much information about previous storage actions as possible. Ideally, the investigator can associate any bit of lower layer content with some upper layer block. This obviously includes active mappings as well as unmapped blocks. In our model, the investigator has to reconstruct φ at the time of acquisition, but he or she also needs to look at and interpret unmapped blocks. We now discuss under which conditions this can be done. To approach this question, we define three different increasingly powerful notions of reconstruction that characterize three different practical scenarios that are illustrated in Section "Abstraction layers in practice". We begin with some general observations.

Finding and decoding φ

The mapping φ is a management data structure that must be stored somewhere where the lower layer can access it. The first challenge faced by the investigator is therefore to *find* φ, i.e., to make sure that φ has been acquired. This may be difficult in practice if, for example, φ is never stored in l but is rather kept on different storage or in a cache in memory only. Investigators must therefore take care that φ is part of the evidence collected.

Even if φ is contained in the collected evidence, still another issue is that it must be fully understood to be able to perform evidence reconstruction. This is problematic for closed source systems in which the mapping is part of the intellectual property of the vendor. From the perspective of digital forensics, the functionality of φ must be known. If vendors are uncooperative, this knowledge must be reverse engineered, e.g., from the meaning of the code that handles the data. There are many examples from the literature that perform such analyses (Gruhn, 2015; Willems et al., 2012; Kurtz, 2012; Russinovich et al., 2012a, 2012b). Next to acquiring φ, investigators must be able to decode φ and convert it into a representation according to our model.

Given availability of φ, we now define three increasingly difficult reconstruction tasks for the investigator.

Active mappings reconstruction

Consider the example shown in Fig. 3 and assume the investigator has acquired evidence at time $t = 5$. What could be a desirable outcome of a forensic analysis, given the evidence? Clearly, the reconstruction should cover at least all active mappings from u to l, i.e., the outcome of the analysis is the relation φ_t.

Definition 1. (active mappings reconstruction). *Given a copy e of l at time t, solving* active mappings reconstruction *means to*

- *find and decode φ_t from e and*
- *enumerate all elements of φ_t.*

Using the information from φ_t it is possible to reconstruct all mapped blocks in u from l. For example, in Fig. 3 for $t = 5$ this will result in identifying that $u[0]$ is mapped to $l[0]$ at acquisition time and that no other block is actively mapped.

In a sense, active mappings reconstruction is the "easiest" reconstruction task. If all information about active mappings is accessible within l, solving basic reconstruction is only restricted by the difficulty to find and "understand" the encoding of φ. The non-existence of φ in e (i.e., not being able to *find* φ) is probably the only reason for Def. 1 not being solvable in practice. If it can be found, *understanding* φ might also be hard but not fully impossible. In this sense, active mapping reconstruction can be understood as a *lower bound benchmark* problem: If a reconstruction method cannot solve basic reconstruction, it is probably only partially useful.

Deleted mappings reconstruction

From a digital forensics perspective, it is usually desirable to solve more difficult problems than merely active mapping reconstruction. As a demonstration, consider Fig. 3 at $t = 2$ where $u[0]$ was unmapped from $l[1]$. In many practical cases data residing in $l[1]$ remains present and is part of evidence collected. Depending on the respective situation, the content of $l[1]$ may be vital evidence, i.e., the data in $l[1]$ may be the content of a deleted file. In case the deleted mapping from $u[0]$ to $l[1]$ can be reconstructed, it is possible to attribute $l[1]$ to $u[0]$, i.e., to reconstruct the mapping that existed before the mapping was deleted. As we will show later, this situation is not uncommon in practice.

Definition 2. (deleted mappings reconstruction) *Given a copy e of l at time t, solving* deleted mappings reconstruction *means to*

- *find and decode φ_t from e,*
- *enumerate all elements of φ_t, and*
- *enumerate a non-trivial subset Δ of all deleted mappings of φ_t, i.e., the empty set is only allowed as a response if deleted mappings do not exist in φ_t.*

Note that Def. 2 is strictly stronger than Def. 1, i.e., it includes computation of all active mappings but requires solving an additional task, namely the computation of deleted mappings. As discussed in Section "Model", the set of deleted mappings of φ_t for $t = 5$ in Fig. 3 is $\{(0,1),(1,1)\}$. Our definition only requires computing a *subset* of deleted mappings, i.e., both $\{(0,1)\}$ and $\{(1,1)\}$ would be solutions of deleted mapping reconstruction, as well as $\{\}$.

In theory, Def. 2 also allows to return mappings that have been re-assigned (as is the case for (0,1) since an active mapping (0,0) exists at $t = 5$). However, since the notion of deleted mappings includes the entire history of φ, computation of a subset of these mappings appears the best feasible solution. Since any method that solves Def. 1 trivially solves Def. 2 by simply returning the empty set as a third response, we require a *non-trivial* subset to be returned. This means that the empty set is only allowed in certain trivial scenarios. Intuitively, the solution should "make an effort" to find deleted pointers. The possibility of finding them should not depend on the concrete scenario but rather on the technique used to manage the abstraction layer.

Last deleted mappings reconstruction

Solving deleted mappings reconstruction is only somewhat satisfactory, because any one of equally treated deleted mappings may be responsible for the data in the lower layer. While it may appear infeasible to decide which block was most recently unmapped, it is sometimes possible to distinguish the "age" of a mapping in practice and therefore to tell which one is more likely to have been responsible for the last changes in a data block at the lower layer.

Definition 3. (last deleted mappings reconstruction) *Given a copy e of l at time t, solving* last deleted mappings reconstruction *means to*

- find and decode φ_t from e,
- enumerate all elements of φ_t,
- *enumerate a non-trivial subset of all deleted mappings of* φ_t, *and*
- *to enumerate the most recently deleted mappings from that subset*

In a sense, Def. 3 refines Def. 2, since it additionally "qualifies" the deleted mappings in a certain way. In Fig. 3 at $t = 5$, a possible solution would be to identify (1,1) as the most recent active link from u to l. Since the evidence is stored in l, the most recent active link is most likely the best for reconstruction: previous mappings are interesting but not really useful since the content of l will probably be changed during the assignment. In this sense, Def. 3 is the strongest useful reconstruction problem we can define.

Def. 3 intentionally includes Def. 2 (and therefore also Def. 1). Similar to Def. 2, Def. 3 only requires computing a subset of deleted mappings, and from this set those elements that were last deleted. A method that solves Def. 2 could "intentionally" only return deleted pointers that were not deleted last, thus avoiding to do extra work. It could then return an empty set as fourth element and therefore trivially also solve Def. 3. Therefore currently, Def. 3 is *not* strictly stronger than Def. 2. At present, it is not clear to us how to formulate an appropriate non-triviality condition to make it strictly stronger.

Identifying the model in practice

After giving the definitions, the task is to investigate the options to solve these problems in practice. For this, we need to focus on a particular abstraction layer and a concrete implementation. To begin, we need to identify the upper and lower layer storage abstractions u and l in the implementation and name the evidence e that is the basis for solving the problem. Describing e precisely is important since it might exceed a "plain" copy of l which might enable to solve stronger reconstruction problems. For example, in many practical scenarios last deleted mapping reconstruction can be performed using backups, data from snapshots, or deleted snapshots (including memory snapshots). If backups or memory snapshots belong to the evidence set e, it is obviously easier to solve more difficult reconstruction problems. Basically, we wish to focus on the analysis of the "pure" management mechanisms and not on any backup mechanism, but while in many practical settings the difference between these mechanisms is obvious, there are certainly systems in which this difference is blurred. Therefore, the solvability of the reconstruction problems must depend on the available evidence e.

Abstraction layers in practice

This section provides block storage and memory mapping examples for each of the three definitions above. We begin by presenting a real example that shows the complexity of abstraction layers in practice.

A real example

To illustrate the complex interactions of mappings between virtualization layers, consider Fig. 4 which shows a typical virtualized installation found in many computing centers today: On the top are amd64 based host servers running VMware ESX with multiple VMs. On the bottom is a 3PAR block storage array from Hewlett Packard Enterprise (HPE).[1] The hosts are connected with iSCSI or Fibre Channel (FC) to the block storage array.

Starting at the bottom of Fig. 4, we can see four physical disks which are divided into regions called chunklets.[2] Chunklets from different disks are organized as a RAID using RAID level 1 in our example (mapping (i) in Fig. 4): two chunklets from different disks are assembled into a RAID Set. This guarantees that at least one chunklet out of any RAID set is responsive if one disk fails.

3PAR, like every other block storage system, has two or more processing elements (called *nodes*). In case of 3PAR, a node is a Linux amd64 PC within the block storage array that connects the hosts to the physical disks. Each physical disk is connected to two nodes. In normal operation each node accesses half of the disks connected to it. If one node fails, the other node takes over the remaining disks. A logical disk is a stripe set which spawns over RAID sets in order to use all available input/output operations of the underlying physical disks to provide the best possible performance (mapping (h)). Logical disks are local to each node.

The volume that is presented to the hosts is striped across all logical disks of all nodes, to provide the best possible performance by spawning a single volume over all physical disks (mapping (g)).

Fig. 4 depicts three volumes: The left two volumes are used as block devices for the file systems VMFS 1 and VMFS 2 (mapping (f)). The right volume is used as a raw device mapping (RDM) providing a block device for VM2 (mapping (l)). Hosts are connected using a SAN (storage area network) to the 3PAR system allowing reading and writing blocks to the storage.

VMFS 1 and VMFS 2 use the cluster file system VMFS which allows to be mounted on up to 64 hosts read/write at the same time. The VMFS stores the persistent or static state of a guest VM consisting of a VMDK (Virtual Machine Hard Disk) file, the virtual machine configuration, and the BIOS settings in form of the NVRAM (mapping (e)).

VMware ESX simulates widely used physical hardware in software as an IDE or SCSI block device (mappings (d) and (k)) so that inside the guest we have access to the virtual machine hard disk. The guest operating system usually creates a file system on top of the block device (mappings (c) and (j)).

Changing our focus from disk to memory, in virtual environments we have three different memory address space abstractions and mappings in-between: The virtual address space is the space that any kind of user or kernel space software uses. It is mapped by the virtual MMU to the guest physical address space (mapping (a)). The guest physical space contains part of the host physical RAM and simulated PCI devices such as the IDE or SCSI controllers we described above. The guest physical address space is mapped by the physical MMU to the host physical address space containing physical memory modules and physical PCI devices (mapping (b)).

In the following sections we discuss selected abstraction layers from Fig. 4 and relate these to the definitions above.

[1] While we could have chosen a different hypervisor and block storage vendor, ESX and 3PAR are often used in practice and allow for a large variety of examples.

[2] While in our example we have only 4 physical disks, block storage systems can grow up to several hundred hard disks.

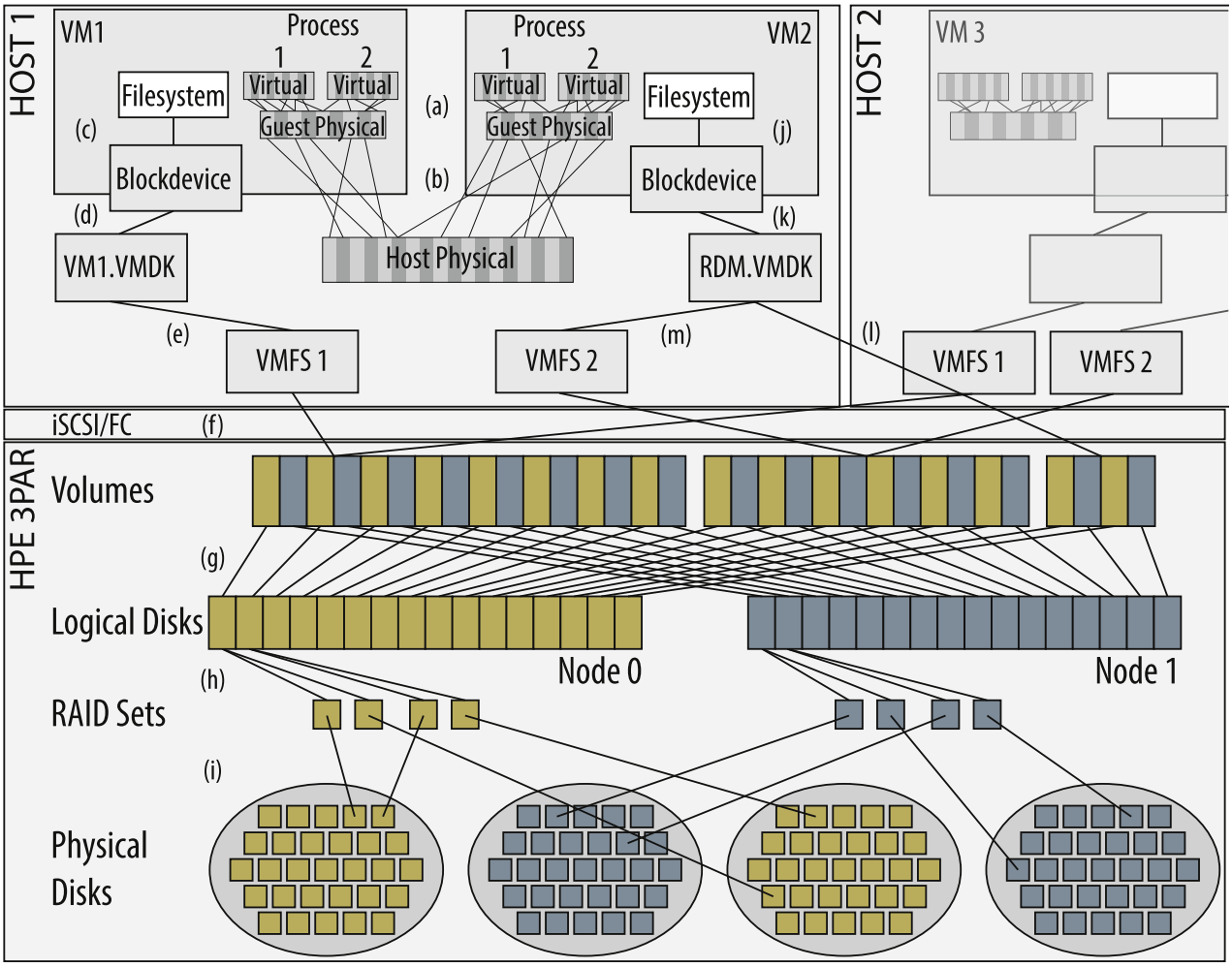

Fig. 4. Example of abstraction layers for block storage and memory: Multiple VMs access central block storage through a SAN. The same VMs utilize physical memory of the host.

Virtual memory management in Linux

The first example we consider is the virtual memory abstraction that is present in modern operating systems. More precisely, we refer to mapping (a) in Fig. 4, i.e., the mapping of guest virtual to guest physical memory for Intel x86 architecture. Each process (including the kernel) has its own 32 bit virtual address space which consists of a sequence of pages. These pages are mapped by additional hardware (the MMU) to the physical memory of the guest using a *page table* data structure.

Fig. 5 shows how a virtual address is mapped to a physical address. Each process has its own mapping. When the process is

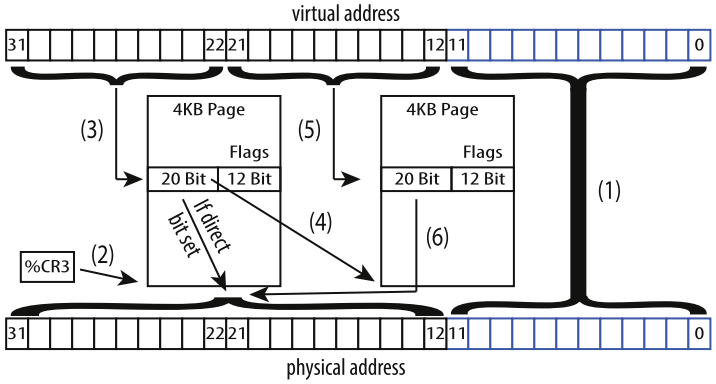

Fig. 5. Management structure to map a virtual address to a physical address on the Intel X86 platform.

active, the upper 20 bit of the CPU control register 3 (`%CR3`) point to the root of a tree data structure called the page directory (item (2) in Fig. 5). Pages are always aligned to 4 KB meaning that the lower 12 bit of their addresses are always zero. Therefore, the lower 12 bit can be mapped one-to-one from the virtual address to the physical address (item (1)). The upper 10 bit are the index of the page directory (item (3)). The page directory contains 1024 entries, 4 bytes each. The upper 20 bit are used for the upper 20 bit if the direct bit is set within the flags, otherwise it references the page table (item (4)). Bits 12–22 of the virtual address are used as indices for the page table (item (5)). Entries in the page tables are again 4 bytes each. The upper 20 bit point to the upper 20 bit of the physical address (item (6)) in case the present bit in the flags is set. If the present bit is not set, the page table entry is undefined and a page fault is evoked when the page is accessed.

We assume that the evidence e consists of a copy of the guest physical memory. In the terms of our model, the upper layer is the sequence of virtual memory pages and the lower layer is the sequence of guest physical memory pages. The mapping is given by the page tables that are also stored in guest physical memory. Given a memory snapshot of the guest physical memory, it is possible to find the page directory for the virtual address space of the process in question. Using the page directory enables us to follow the individual page table entries and their *present* bit to see which virtual page is mapped to which physical page. So the data structure enables us to solve at least active mappings reconstruction (Def. 1).

Sometimes virtual pages are unmapped from physical pages. One example is a physical page being swapped out to disk. Handling the page table entry is dependent on the operating system. In the Linux kernel (since version 2.0.40), a page table entry is cleared by setting it to 0 (function `native_pte_clear` in `pgtable-2level.h`). Therefore, it is impossible to extract the prior value of the physical page that pointed to the entry as well as solving deleted mapping reconstruction (Def. 2) for Linux.

The guest physical memory is managed as a sequence of *page frames* to which pages of virtual address spaces are mapped. Linux uses a data structure `mem_map` to keep track of the status of physical pages. This structure is also used by page management to swap out "old" pages to make space for "new" ones. In this process, pages from one virtual address space can be unmapped without request by the respective process. In this sense, the guest physical layer is mapping-aware.

DOS/MBR partition management

The next example refers to the management of partitions on block devices along the popular DOS/MBR partition scheme (Carrier, 2005). In this scheme, partitions are used to structure the space of LBA-addressable blocks of the block device into logically separated areas. In Fig. 4 this is part of mapping (c) consisting of multiple abstraction layers (Carrier, 2003). In analogy to the logical block address (LBA) of a block device, blocks within a partition can be referenced using a *logical partition volume address* numbering the sequence of blocks within the partition from 0 to the end.

In the DOS/MBR scheme, a *partition table* is used to specify and delineate partitions. Roughly speaking, the partition table can be found in the first sector of the block device and contains starting block and length of up to four partitions. Using this information, it is possible to map the logical partition volume address of a block in the partition to the LBA of the block on the block device (see Fig. 6).

In our model, the upper layer u consists of the sequence of logical blocks of the partition. The lower layer l is the sequence of

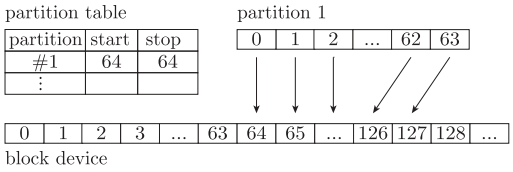

Fig. 6. Partition management in DOS/MBR.

LBA addressable blocks on the block device. The evidence e usually consists of a bitwise copy of the entire block device that includes the first sector containing the partition table. The mapping from u to l can be performed by adding the logical partition volume address to the starting LBA of the partition.

Partitions are usually rather stable objects; they do not change very often. The deletion of partitions, however, is usually done with command line tools like `fdisk` (MS-DOS/Windows 98/Linux), `diskpart` (from Windows NT), or with GUI-based tools like `gparted` (Linux). If an existing partition is deleted, the usual behavior is to overwrite the partition table entry with zeros. Furthermore, the partition entry n + 1 moves down if partition entry n is deleted. This happens if the disk manager in the Windows control panel (Freiling and Gruhn, 2015) is being used. In effect, the deletion of a partition table entry results in a deletion of the mappings from u to l. In general, the DOS/MBR scheme does not maintain a redundant copy of the boot sector (Carrier, 2005). Therefore, it is generally impossible to reconstruct mappings of deleted partitions. Thus, for this abstraction layer we cannot solve the last deleted mapping reconstruction problem.

Logical volume management

Linux LVM2 is a volume manager that can assemble one or more logical volumes (LV) out of one or more physical volumes (PV). An LV is a block device that can be used raw, to create a file system, or as a basis for another layer, e.g. disk encryption. LVs can be added, resized, and removed. PVs can be hard disks, partitions, or loopback files; PVs can also be added and removed. A volume group (VG) contains one or more PVs. Extents are used to map the LVs to PVs.

LVs contain one or more virtual extents (VE) which are mapped to one or more physical extents (PE). VEs and PEs are the same size. The mapping is stored during runtime in the kernel as part of the generic device mapper framework. Additionally, the mapping is kept in the metadata area of each PV and in the directory `/etc/lvm/backup` as ASCII file. The complete history is backed up to the `/etc/lvm/archive` directory. During creation of an LV, the first 64 KB are initialized with zeroes. Otherwise the physical blocks are left intact.

Fig. 7 shows vg0 at $t = 0$ containing one active LV lv0 with three VEs mapped to three PEs of pv0. At $t = 1$ LV lv0 is resized using only one VE. One additional LV lv1 has been created which utilizes PE1 and PE2 of pv0, both previously allocated to lv0.

In order to reconstruct the mapping of the recently deleted VE1 and VE2 of lv0, metadata need to be recovered. By default the complete history of all previous mappings can be found in the directory `/etc/lvm/archive`. If the directory is unavailable the metadata can be recovered from the metadata area of each PV.

The layout of a PV is as follows: The first 512 bytes of the PV contain a label with a unique identifier, followed by a metadata area which is by default 1 MB − 512 Bytes, followed by the physical extents, sized 4 MB by default, starting at 1 MB. If the device is not a multiple of 4 MB plus 1 MB, the remaining space is unused.

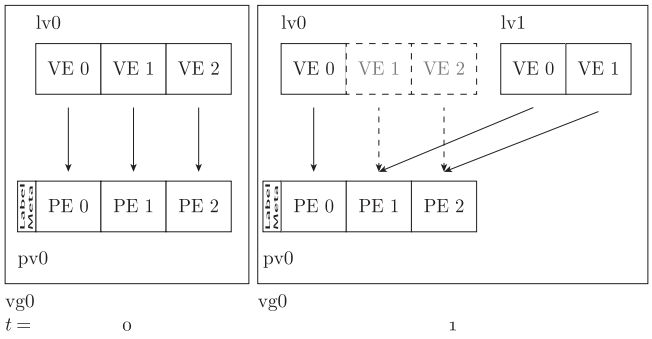

Fig. 7. LVM2 physical volume layout.

In order to recover previous mappings from the metadata of the PV, the first 1 MB of a PV can be extracted using dd. The metadata is a ringbuffer containing the metadata as a continuous ASCII string. A pointer at the beginning of the metadata points to the current active mapping. Once extracted, the metadata area can be viewed using a text editor since it contains by design the active and at least the most previous mapping in ASCII representation. It is likely to recover the complete history of all previous mappings, because the ASCII representation of a mapping is usually less than 4 KB. The PV metadata area is almost 1 MB, and LVM metadata updates are infrequent.

Listing 1 contains a shortened version of the ASCII representation of the metadata extracted from the metadata region previous to resizing lv0. The creation_time contains a unix time stamp in seconds from Jan 1st, 1970, vg0 is the name of the volume group. The id within the volume group is a unique identifier for the VG, seqno is the sequence number which will be incremented for every modification, and extent_size contains the size of the extent in sectors. A sector is 512 Bytes.

The section physical_volumes contains one entry for every PV: pv0 is the first physical device. The id is a unique identifier, which can also be found in the label of the PV, device is a hint for the physical device, dev_size is the size of the device in sectors, pe_start is the location of the first PE on the device in sectors from start of the device, and pe_count is the number of PEs on the device.

The section logical_volumes contains all LVs for the VG. Similar to the VG it contains an id and creation_time, segment_count describes how many segments are used to back the LV. A segment is a continuous allocation of PEs, segment1 describes the first segment, start_extent is the VE number starting with 0, extent_count is the number of consecutive allocated extents, type can either be linear or striped. The stripe_count is 1 which results in linear (continuous) allocation of PEs. The section stripes references the PVs and the start extent.

In the above example the upper layer u is the set of VEs and the lower layer l is the set of PEs. We assume that the evidence e consists of a copy of all PVs that belong to the LVM. This evidence in LVM2 allows for solving last deleted mappings reconstruction (Def. 3) as LVM2 keeps the current mapping and the most recent mapping in the metadata area of every device as well as in the directory `/etc/lvm`.

LVM2 issues UNMAP commands to the physical devices if (a) they support it, (b) the filesystem on top of the LVM2 logical volume supports discard, and (c) issue_discards is set to 1 in `/etc/lvm/lvm.conf`. LVM2 is therefore mapping aware in these cases, i.e., the lower layer always "knows" if a certain PE in use or not.

Listing 1. LVM2 Text format volume group metadata

```
...
creation_time = 1483598692

vg0 {
 id = "Ubd6dH-...-LmhjbO"
 seqno = 2
...
 extent_size = 8192 # 4 Megabytes
...
 physical_volumes {
  pv0 {
   id = "kEeQij-...-BanVIt"
   device = "/dev/loop0" # Hint only
...

   dev_size = 26624 # 13 Megabytes
   pe_start = 2048
   pe_count = 3     # 12 Megabytes
  }
 }

 logical_volumes {
  lv0 {
   id = "bihcBw-...-z51XDO"
...
   creation_time = 1483598689
   segment_count = 1
   segment1 {
    start_extent = 0
    extent_count = 3 # 12 Megabytes
    type = "striped"
    stripe_count = 1 # linear




    stripes = [
     "pv0", 0
    ]
   }
  }
 }
}
```

Conclusions

In this paper we investigated the effect of abstraction layers on forensic evidence collection. It is well-known that evidence collection on higher layers (such as cloud environments (Roussev

and McCulley, 2016)) for multiple reasons is usually inferior to collection at lower layers. But little research has been done to formally characterize the loss of digital evidence that arises from having abstraction layers at all. Understanding the corresponding issues is important since the growing complexity of modern computer systems will lead to an increasing complexity in hierarchies of abstraction layers.

The differences in the semantics of different abstraction layers have been described as *semantic gap* (Birk and Wegener, 2011; Kolb, 2012), a term borrowed by computer science from linguistics. The semantic gap has been identified to exist in virtualized environments, especially in the context of VM introspection (Dolan-Gavitt et al., 2011). The reason for this is that many management structures for the resources of a virtual machine (VM) usually run within the VM itself; without detailed knowledge of these structures it is impossible to analyse and interpret the "semantics" of upper layers from information stored at lower layers. In the context of forensic analysis, the semantic gap widens even further because last deleted mapping reconstruction (as defined above) tries to interpret the "deletion semantics" of the upper layer: Usually, mappings cater only for the blocks that are currently in use. However, former mappings, i.e., assignments of blocks that are no longer "actively" mapped, are also of interest in forensic investigations and qualify as evidence.

One major insight in the present paper is that logical access restricts evidence for mapping-aware systems. This is becoming increasingly evident for systems using SSDs with IDE TRIM enabled (Nisbet et al., 2013; King and Vidas, 2011), but with increasing use of logical volumes and storage virtualization it appears that evidence collection at lower layers in its result will become increasingly similar to collection at higher layers, even using APIs (Roussev et al., 2016). For mapping agnostic systems, it appears that deleted mappings can often be recovered, overwritten mappings usually not.

Interestingly, while individual file storage can be represented within our model, our model does not generally fit well onto the abstractions provided by *file systems*. File systems provide a hierarchical structure of files and directories which our "flat" model of storage blocks cannot represent. In future work we aim to generalize our model to also be able to describe file system abstractions more easily. Additionally, since we focussed on the analysis of one particular abstraction layer, it makes sense to investigate questions of *compositionality* in the future. The question to be answered is: Can solutions to reconstruction problems at one layer be composed to solutions that cover multiple layers?

Acknowledgments

The authors wish to thank the anonymous reviewers for their helpful feedback. This work was partly supported by the "Bavarian State Ministry of Education, Science and the Arts" as part of the FORSEC research association.

References

Abadi, M., Lamport, L., 1991. The existence of refinement mappings. Theor. Comput. Sci. 82 (2), 253–284. http://dx.doi.org/10.1016/0304-3975(91)90224-P.

ATA/ATAPI Command Set - 2 (ACS-2), 2011. INCITS 482-2012, INCITS.

Birk, D., Wegener, C., 2011. Technical issues of forensic investigations in cloud computing environments. In: Proceedings of the 2011 Sixth IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering, SADFE '11. IEEE Computer Society, Washington, DC, USA, pp. 1–10. http://dx.doi.org/10.1109/SADFE.2011.17.

Carrier, B.D., 2003. Defining digital forensic examination and analysis tool using abstraction layers. IJDE 1 (4). URL. http://www.utica.edu/academic/institutes/ecii/publications/articles/A04C3F91-AFBB-FC13-4A2E0F13203BA980.pdf.

Carrier, B., 2005. File System Forensic Analysis. Addison-Wesley Pub. Co. Inc., Boston, MA, USA.

Carrier, B.D., Spafford, E.H., 2006. Categories of digital investigation analysis techniques based on the computer history model. Digit. Investig. 3 (Suppl. 1), 121–130. http://dx.doi.org/10.1016/j.diin.2006.06.011.

Casey, E., 2011. Digital Evidence and Computer Crime – Forensic Science, Computers and the Internet, third ed. Academic Press.

Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J.T., Lee, W., 2011. Virtuoso: narrowing the semantic gap in virtual machine introspection. In: IEEE Symposium on Security and Privacy. IEEE Computer Society, pp. 297–312.

Dykstra, J., Sherman, A.T., 2012. Acquiring forensic evidence from infrastructure-as-a-service cloud computing: exploring and evaluating tools, trust, and techniques. Digit. Investig. 9 (Supplement), S90–S98. http://dx.doi.org/10.1016/j.diin.2012.05.001 the Proceedings of the Twelfth Annual DFRWS Conference.

Freiling, F.C., Gruhn, M., 2015. What is essential data in digital forensic analysis?. In: Ninth International Conference on IT Security Incident Management & IT Forensics, IMF 2015, Magdeburg, Germany, May 18–20, 2015, pp. 40–48. http://dx.doi.org/10.1109/IMF.2015.20.

Grobauer, B., Schreck, T., 2010. Towards incident handling in the cloud: challenges and approaches. In: Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop, CCSW '10. ACM, New York, NY, USA, pp. 77–86.

Gruhn, M., 2015. Windows NT pagefile.sys virtual memory analysis. In: Ninth International Conference on IT Security Incident Management & IT Forensics, IMF 2015, Magdeburg, Germany, May 18–20, 2015, pp. 3–18. http://dx.doi.org/10.1109/IMF.2015.10.

King, C., Vidas, T., 2011. Empirical analysis of solid state disk data retention when used with contemporary operating systems. Digit. Investig. 8, S111–S117.

Kolb, A., 2012. Ansätze für forensische Untersuchungen in IaaS Cloud-Umgebungen. Diplomarbeit, Ruhr-Universität Bochum.

Kurtz, A., 2012. Pentesting iOS Apps — Runtime Analysis and Manipulation. In-depth Security Conference (DeepSec 2012), Presentation.

Martini, B., Choo, K.-K.R., 2012. An integrated conceptual digital forensic framework for cloud computing. Digit. Investig. 9 (2), 71–80.

Nisbet, A., Lawrence, S., Ruff, M., 2013. A forensic analysis and comparison of solid state drive data retention with trim enabled file systems. In: 11th Australian Digital Forensics Conference, pp. 103–111.

Roussev, V., McCulley, S., 2016. Forensic analysis of cloud-native artifacts. Digit. Investig. 16 (S), S104–S113. http://dx.doi.org/10.1016/j.diin.2016.01.013.

Roussev, V., Barreto, A., Ahmed, I., 2016. Forensic Acquisition of Cloud Drives, CoRR abs/1603.06542.

Russinovich, M., Solomon, D., Ionescu, A., 2012. Microsoft Windows Internals, Part 1, sixth ed. Microsoft Press.

Russinovich, M., Solomon, D., Ionescu, A., 2012. Microsoft Windows Internals, Part 2, sixth ed. Microsoft Press.

SCSI Block Commands - 3 (SBC-3), 2014. INCITS 514-2014, INCITS.

Tanenbaum, A.S., 2002. Computer Networks, 4. ed. Prentice Hall.

Willems, C., Freiling, F.C., Holz, T., 2012. Using memory management to detect and extract illegitimate code for malware analysis. In: 28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3–7 December 2012, pp. 179–188. http://dx.doi.org/10.1145/2420950.2420979.

Zawoad, S., Hasan, R., 2016. Chronos: towards securing system time in the cloud for reliable forensics investigation. In: COMPSAC. IEEE, pp. 423–432.

Zawoad, S., Hasan, R., 2016. Trustworthy digital forensics in the cloud. IEEE Comput. 49 (3), 78–81.