Remote data acquisition on block devices in large environments

A study into copy-on-read and copy-on-write methods

Eric van den Haak

July 11, 2014

UNIVERSITY OF AMSTERDAM

Version
Final

Supervisor
Ruud Schramp

Abstract

This research is performed to aid in remote forensics on large environments. It is becoming more difficult each day to store copies of all available data locally to perform acquisition onto. Therefore, a method is desired that enables to store already (remote) read data into separate sparse-files locally. Another desire is the possibility to perform live forensics on remote data, for example booting an operating system from an external block device, without interfering with original data. Research has been done into existing methods that can do both copy-on-read and copy-on-write simultaneously. Since no existing method was found that implements this concept, a concept is introduced called CoRaW (copy-on-read-and-write) that implements both copy-on-read and copy-on-write simultaneously. Existing methods that have implemented either copy-on-read or copy-on-write are code-reviewed and have been tested upon their functionality. Finally, two different copy-on-read-and-write proof-of-concepts have been implemented based upon two of the found methods[1]. Experiments were performed on live forensics as well as on data-integrity of locally stored data. Concluded is that both implemented proof-of-concepts can be used in a satisfactory way. However, Xmount is preferred because it has already been accepted within the forensics world and has a larger community.

[1] Proof-of-concepts can be found on github[8].

Contents

1. Introduction

In modern days the amount of available data in data centers is enormous. In a forensics aspect, it is a daily growing challenge to get all of the data out of a data center in order to perform forensic research on it. The Netherlands Forensic Institute(NFI) has requested to perform research in order to find a solution to this problem by developing an “easy” way to remotely connect directly to the required hard drives of a certain system and be able to store only the required content necessary for forensics locally. This research has been divided into three sub-researches; a client, which should be a very small operating system bootable by CD or PXE that automatically connects the systems block devices to a server over a secure channel; a server, which offers to read this block devices in a smart way; and finally the acquisition part which can perform analyses on the acquired data.

1.1. Research

This research will focus upon the server part of the concept and mainly upon the block device level. As it is not always possible to copy all data, only necessary data should be transferred to the server’s storage. In order to achieve this, a copy-on-read(CoR) system combined with a copy-on-write system is desired. While copy-on-write(CoW)[2] (file) systems already exist at large scale, for example fusecow[6], it is hard to find copy-on-read (file) systems. A copy-on-read file system would allow only remotely read data to be stored locally, resulting in always having access to already read data. Ideally, a solution is found that can mount an existing block device that performs both CoW and CoR simultaneously.

This leads to the following research questions:

What is a good way to mount block devices read only and store read and changed data into separate sparse files?

- *What methods exist that allow copy-on-write and copy-on-read on block device level?*
- *Can these methods be effectively used to perform remote data acquisition while storing read and changed data locally?*
- *If necessary, how can an existing method be modified in order to meet the requirements of this research?*

[2] `https://en.wikipedia.org/wiki/Copy-on-write`

1.2. Related research

Last month, researchers published an article about Liquid[9], a proposed file system designed for large virtual machine clusters. The interesting part of this proposed file system is that it supports both copy-on-read as well as copy-on-write, which makes it very interesting to look into for this research.

An existing tool that is used in forensics is xmount[2]. Xmount allows to mount raw block device images but also supports EWF (Expert Witness Compression Format) and AFF (Advanced Forensic Format) files. Also, it allows for copy-on-write, so the images can be mounted and then booted via a virtual machine manager without interfering with the original image itself.

Recent studies of NIST[3] indicate that proper methods that can aid in large environment forensics are demanded. Among found problems are the sheer volume of the media, the lack of ability to respond from more than one physical location in a relative short time and validation of the forensics image. The aim of this research is to provide an effective, yet proper method to aid in forensics on a large scale.

1.3. Approach

First, a literature study will be performed regarding existing copy-on-read and copy-on-write systems. The next step is to research if they are sufficient (or can be combined in an efficient way) to aid in remote data acquisition. Finally, if no sufficient method is found, research will be performed upon finding a new method. The result will be implemented in a proof of concept.

2. Research

This section describes the research that has been performed.

2.1. Existing methods

In this section, existing methods of mounting block devices are analyzed. Next section will go further into each found method. Methods described in this section are those found that might aid in the problem this research tries to solve. Subsections describe how and why. Each of the methods found operates on block device level. This is required as it does not interfere with on-lying file system.

FUSE

FUSE[7] is a framework that allows user space file systems to be build upon. Because FUSE allows to implement a file system relatively "easy" and it does not require a user to be root to operate in user space, FUSE is very popular[3]. FUSE itself is not an immediate solution, but a possible method to build a solution upon. Figure 1 gives an impression of how FUSE works. In this example, a "hello world" file system is used.

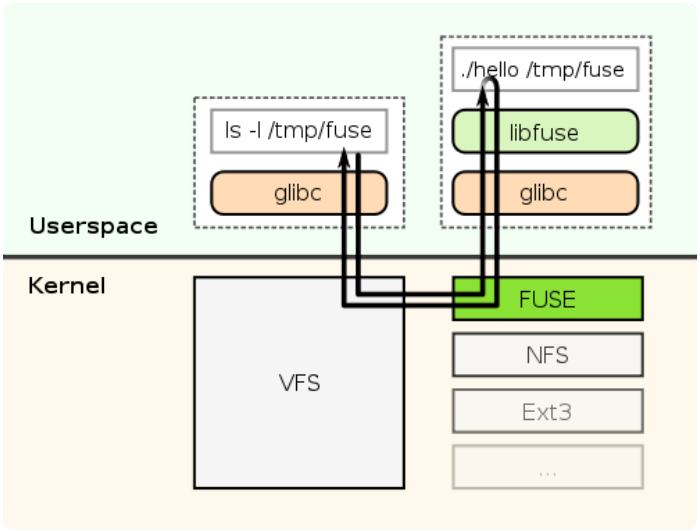

Figure 1: FUSE working. (Source: `https://en.wikipedia.org/wiki/File:FUSE_structure.svg`)

[3] A list of FUSE based file systems can be found at `http://sourceforge.net/apps/mediawiki/fuse/?title=FileSystems`

Xmount

Xmount[2] is an open source tool used in forensics that allows to convert different hard disk images on the fly. It supports raw images (for example those made with the dd tool[4]) but also forensic specific disk images (see section 1.2). It also supports copy-on-write. This means that the tool is out-of-the-box suitable to mount disks and boot from them to inspect the operating system that the image contains while in a running state without interfering with the original data. Xmount is build upon FUSE and is open source. As Xmount in its current state does not allow copy-on-read, it might be possible that this functionality can be added by modifying the source code.

Fusecow

Another open source FUSE based tool is fusecow[6]. Fusecow does nothing more than providing a copy-on-write functionality for a block device or an image. Fusecow itself is also an open source tool and can thus be modified. Authors indicate that it is currently slow, unstable and has a limitation that it cannot grow files yet. As it supports copy-on-write and it also seems to be a small program (only having 468 lines of code), it seems useful to look into.

Bcache

Bcache[5] is a block layer cache operating within the Linux kernel and is also open source. It is designed to use a solid state drive (SSD) combined with an ordinary hard disk drive (HDD) to gain benefits from both the SSD's speed as the HDD's capacity. It performs copy-on-read from one block device (usally an HDD) towards another (usually an SSD) and uses the copied data as cache. Since Linux kernel version 3.10, Bcache is included into the mainline[1]. While the functionality is originally designed for caching purposes, it might be suitable for this research when it is combined with copy-on-write.

Liquid

A proposed file system is Liquid[9]. As of writing this report, no existing implementation could be found. As writing an entire implementation is beyond the scope of this research, further research into liquid has been renounced.

[4] https://en.wikipedia.org/wiki/Dd_(Unix)

2.2. Method effectiveness

Currently, no existing method is found that allows to do both copy-on-read and copy-on-write simultaneously. As of now, each method can theoretically act as either the copy-on-write(CoW) or copy-on-read(CoR) layer (figure 2). As it is very desirable to have one method that allows both techniques, existing methods will be analyzed upon their designed functionality and will be code reviewed. The findings from the analysis will be used to develop a copy-on-read-and-write[5] method (figure 3).

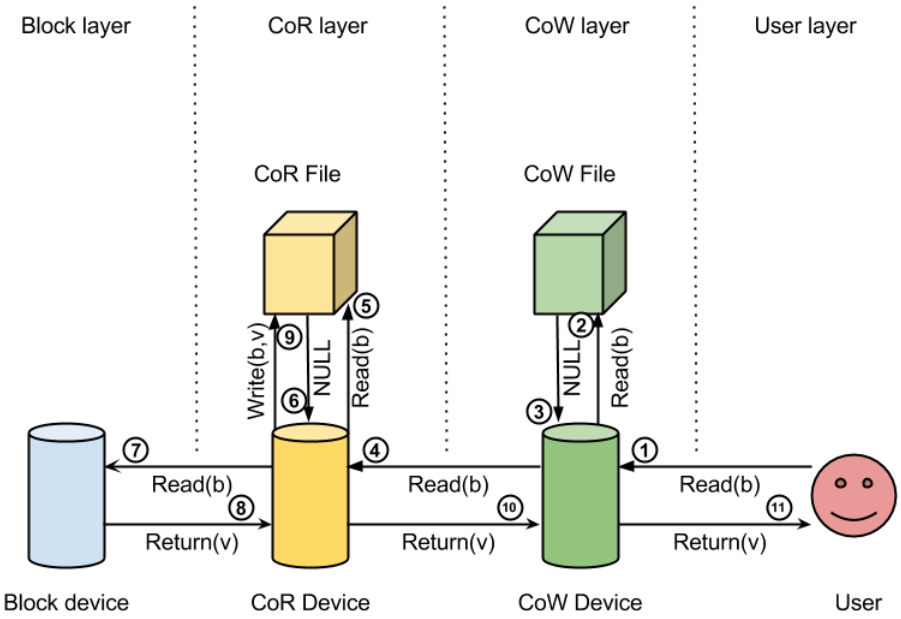

Figure 2: Copy-on-read combined with copy-on-write, initial read.

[5]Hereby introduced as CoRaW.

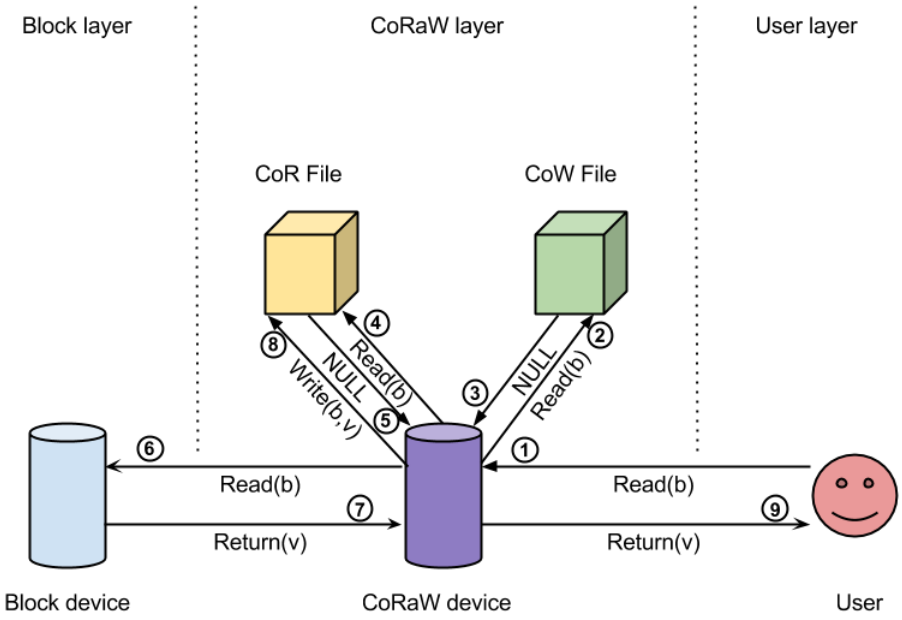

Figure 3: Copy-on-read-and-write, initial read.

2.2.1. Bcache

Bcache is a kernel level operating method that allows to cache reads from one block device towards another (copy-on-read), designed to combine the storage capacity of a hard disk drive along with the speed of a solid state drive. While this method looks very promising for this research, an issue is found regarding forensics. In order to make Bcache work on a not already configured Bcache device, some changes have to be written to the block device. A possibility would be to add a copy-on-write layer in between, but that does not fit into the goal of this research. This is due to the fact that the copy-on-read sparse file should only contain data that is identical to that on the source block device. Therefore, further research into Bcache is renounced.

2.2.2. Testing

As further research into Bcache is renounced, methods that are left to research are copy-on-write techniques only. Therefore, experiments are done regarding copy-on-write. In order to test these methods, a small logical volume[6] was created (appendix A.1) as well as a large logical volume(appendix A.2) that is used for an Ubuntu[4] server installation. The small image serves a purpose for testing the working of the method itself as the large image serves a purpose for a “real” scenario test. The used virtualisation technique is KVM(Kernel-based Virtual Machine)[7]. The following tests will be performed to test the methods.

File system test

The small block device will be mounted via the researching method and will be modified. The original file system should not be affected and changes should be persistent while continuing using the same copy-on-write files.

Block level test

The large block device will also be mounted via the visiting method and will be used to boot the Ubuntu server installation. As in the former test, booting and using the operating system should not interfere with the original block device and changes should be persistent. To test this, some random writes (see appendix A.5) will be performed within the running system. Reading is already tested because the operating system has to boot from the block device.

[6] https://en.wikipedia.org/wiki/Logical_volume_management
[7] http://www.linux-kvm.org/page/Main_Page

2.2.3. Xmount

Xmount allows, as already described in section 2.1, copy-on-write via a cache file. Figures 4 and 5 show the global working of Xmount.

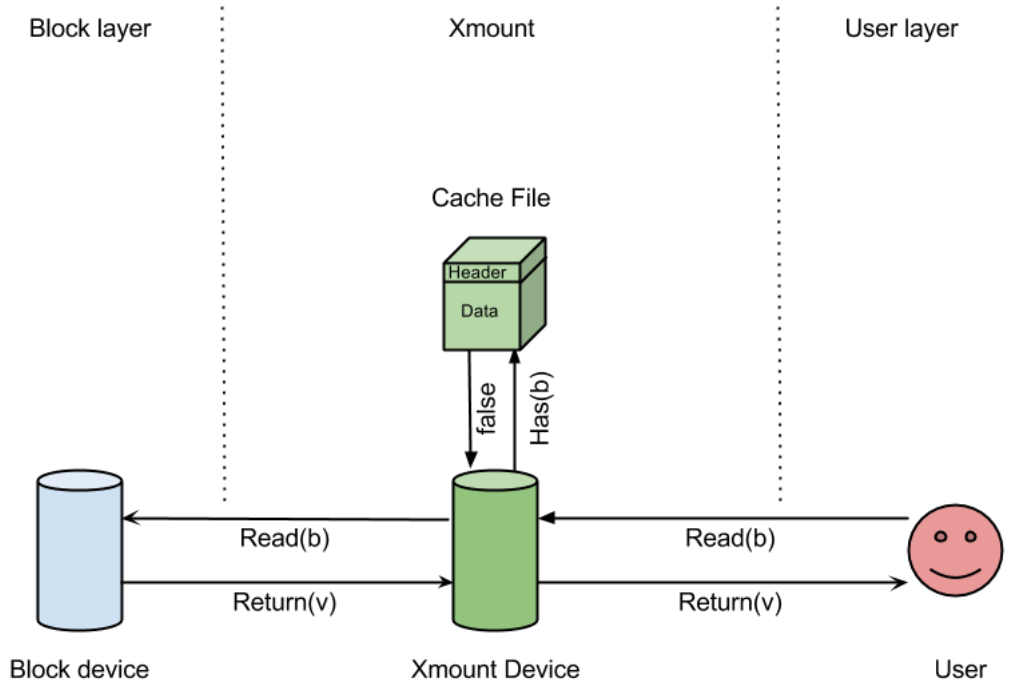

Figure 4: Xmount with cache file - initial read.

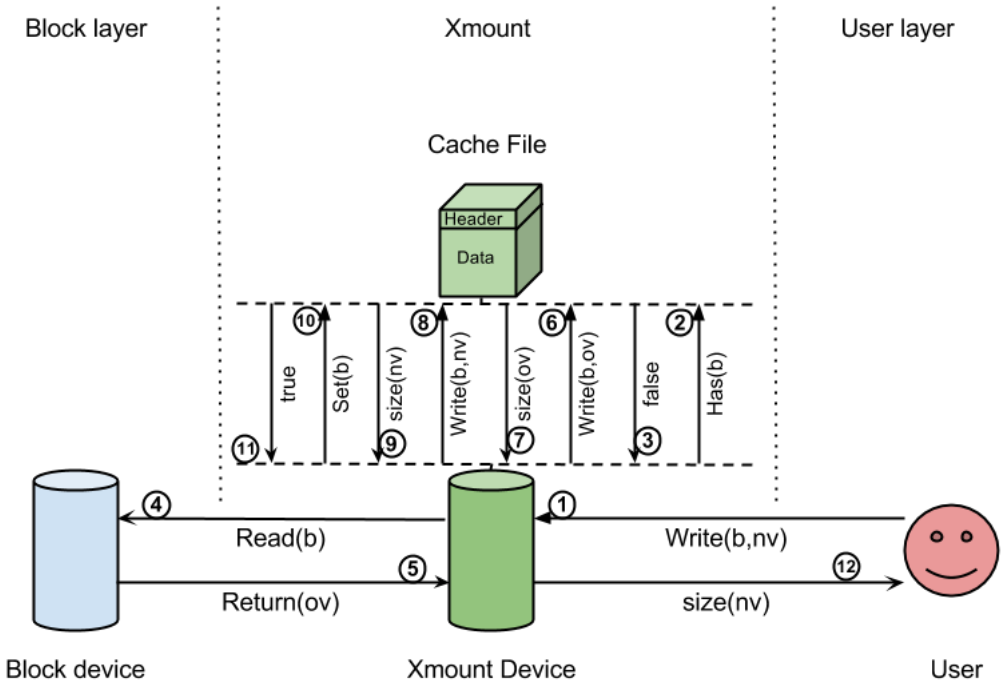

Figure 5: Xmount with cache file - initial write.

Code

Sourcecode of Xmount is well organized and commented. The code can be learned relatively quick and thus be modified for copy-on-read.

```
/*
 * GetVirtImageData:
 *   Read data from virtual image
 *
 * Params:
 *   buf: Pointer to buffer to write read data to (Must be preallocated!)
 *   offset: Offset at which data should be read
 *   size: Size of data which should be read (Size of buffer)
 *
 * Returns:
 *   Number of read bytes on success or "-1" on error
 */
static int GetVirtImageData(char *buf, off_t offset, size_t size) {

...

if(XMountConfData.Writable==TRUE &&
       pCacheFileBlockIndex[CurBlock].Assigned==TRUE)
    {
      // Write support enabled and need to read altered data from cachefile
      if(fseeko(hCacheFile,
                pCacheFileBlockIndex[CurBlock].off_data+BlockOff,
                SEEK_SET)!=0)
      {
        LOG_ERROR("Couldn't seek to offset %" PRIu64
                  " in cache file\n")
        return -1;
      }
      if(fread(buf,CurToRead,1,hCacheFile)!=1) {
        LOG_ERROR("Couldn't read data from cache file!\n")
        return -1;
      }
      LOG_DEBUG("Read %zd bytes at offset %" PRIu64
                " from cache file\n",CurToRead,FileOff)
    } else {
      // No write support or data not cached
      if(GetOrigImageData(buf,
                          FileOff,
                          CurToRead)!=CurToRead)
      {
        LOG_ERROR("Couldn't read data from input image!\n")
        return -1;
      }
      LOG_DEBUG("Read %zd bytes at offset %" PRIu64
                " from original image file\n",CurToRead,
                FileOff)
    }
    ...
```

Listing 1: Part of *xmount.c*[2] that is responsible for block reading.

When reading data, Xmount first checks whether the requested block is available in the cache file (listing 1, line 18), before reading it from the block device. Xmount stores a block index in the header of the cache file that keeps track of already written blocks. Xmount then reads the cached block from the cache file if available, or from the raw block device if not.

If a write action occurs, Xmount checks within the block header if the block is already present. If present, the block will be updated. If not, the original block will be copied from the raw block device to the cache file and will then be overwritten with the data of the write action.

By adding an additional check in the source code on line 35 (listing 1) for a copy-on-read file, desired behavior can be added. The working of the copy-on-read functionality differs from the copy-on-write method as only original data will be written to the copy-on-read file. When reading a block from the original block device, the block should be immediately stored within the copy-on-read file.

Findings

Xmount works as expected (see appendix A.3 for test and results). When used as a mapping device for the Ubuntu VM, the cache file grows along with the written data and is persistent. Xmount's file format for copy-on-write can be reused for copy-on-read functionality.

2.2.4. Fusecow

Fusecow is a very simple copy-on-write system. It works by providing a block device, mount point and copy-on-write file. As mentioned in section 2.1, authors state a few “problems”; fusecow is slow, unstable and has a limitation that it cannot grow files yet. The purpose of this research is acquisition on data in a forensics aspect, meaning that performance is not as important as getting the right data. However, having some performance is desirable.

Code

Fusecow’s code is easy to read and understand. Fusecow operates by using a mapping file that keeps track of modified blocks combined with a separate copy-on-write file that contains the actual changed data (see images 6 and 7 for a global overview). As fusecow does not allow growing files, one can not change more blocks in the file system than originally exist. This does not have to be a problem as in data acquisition not many changes have to occur. For example, booting an operating system from an image should not fail because the image lacks the possibility of growing.

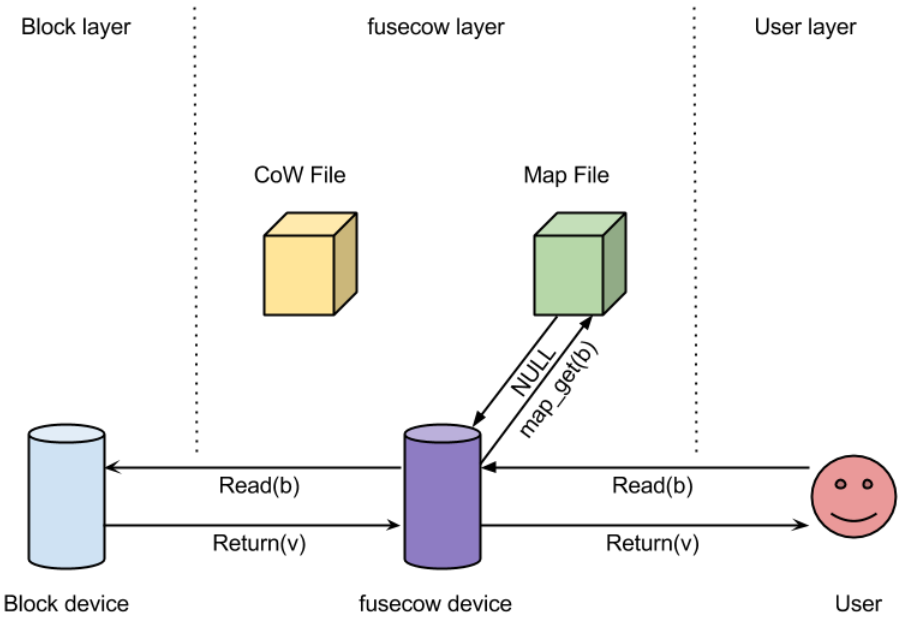

Figure 6: Fusecow - initial read.

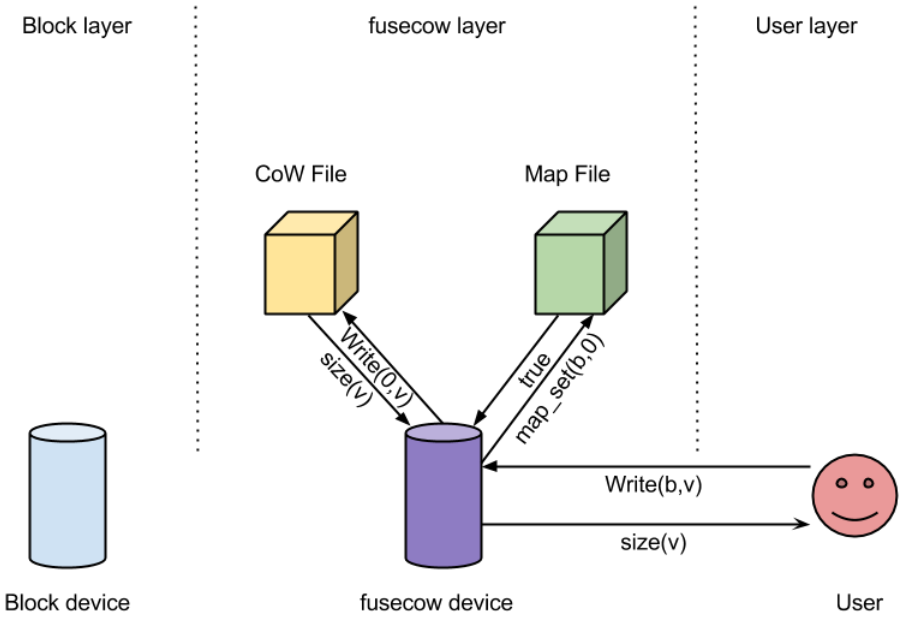

Figure 7: Fusecow - initial write.

As with Xmount, implementing a copy-on-read functionality within fusecow is easily feasible. An additional mapping file and copy-on-read file, which would store already read block information and already read block data respectively, would be necessary.

Findings

On the block device level, fusecow does a proper job. Changes are sufficiently stored within the copy-on-write file and are persistent (see appendix A.4 for tests and results). However, it is found that fusecow is not suited to be used as raw block device copy-on-write as non-root user. Default mount file will have 0600 as permissions and has not implemented the ability to changing these settings; "*chmod: changing permissions of '/tmp/rp2/fsc.ubuntu.mnt': Function not implemented*". When running KVM as root, the virtual machine can be started and copy-on-read works as expected.

2.3. Implementing a proof-of-concept

This section describes the research performed towards proof-of-concept implementations for a CoRaW (see section 2.2) method.

2.3.1. Decisions

As found in former research to existing methods, the candidates left to develop a good method to perform copy-on-read-and-write are Xmount and Fusecow. As research into Bcache is renounced, more time is available to research implementations. Therefore, it has been decided that for both methods a proof-of-concept will be developed, that will be analyzed.

In order to determine a scope, it has been decided that each software will be able to store already read blocks into a separate file and will use that file to read from when a desired block is available within. Another demand is to be able to use an existing copy-on-read file read-only, so stored data can be retrieved, but nothing can be added.

2.3.2. Original concept

As explained in the introduction (section 1), this research focuses on a specific part of a larger concept. The larger concept consists of a client, server and acquisition part. The acquisition part actually reads and analyses blocks from the block device. This research is performed to find a good method that stores already read data from a (remote) block device into a sparse (copy-on-read) file. As the findings of the acquisition software are used to build a legal case, the copy-on-read file becomes the evidence as a result. Figure 8 provides an overview.

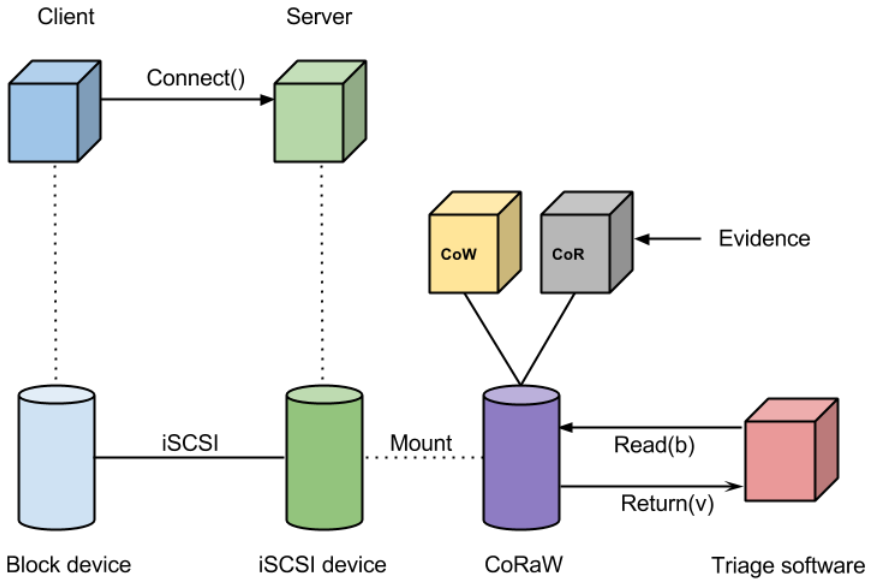

Figure 8: Greater picture.

One or more clients connect to the server via a secure channel and enable the server to directly connect to their block devices via iSCSI. CoRaW software mounts the iSCSI device and Triage software performs acquisition. All analyzed (i.e. read), original data will be stored into a copy-on-read (CoR) file, which is the evidence file. This file will be stored in a secure environment for as long the forensic investigation is going on.

2.3.3. Requirements

Based on the greater picture, the following requirements should be met;

- ***Integrity***
Locally stored blocks have to be exactly the same as the corresponding blocks on the original device.

- ***Storable format for copy-on-read file***
All evidence has to be stored. Having a sparse file that contains empty blocks can result in a problem regarding storing and/or copying. The use of existing methods of both Xmount and fusecow should prevent these problems as new added blocks are appended to the sparse file when they are written.

- ***Re-mountable copy-on-read file***
As this is the evidence file, a method has to be provided that enables reading it. Stored blocks should be present and unstored blocks should return 0x00.

2.3.4. Testing

In order to test the copy-on-read functionality, the Ubuntu installation is used. The idea is to boot the operating system and to login. After it has been running for a while, it has to be shut down and boot again without having the original image present. When no different actions are performed upon this next boot, everything should work as it worked in the former boot sequence. As the developed method will be a proof of concept, some tricks might be needed to remount the block device.

Since perhaps the most important requirement is that stored data is persistent, already backed-up data should be exactly the same as the original. Therefore the second test is to read the same randomly chosen blocks in multiple scenarios:

- Raw block device
- Mount point after mounting the raw block device for the fist time
- Mount point after re-mounting without having the original block device present

If everything works as designed, read blocks should be identical in all cases.

2.3.5. Xmount

As found in section 2.1, Xmount can be altered to enable copy-on-read. Xmount is an open source C program and is published under the GNU license. The approach is to reuse its caching method for a copy-on-read functionality. This method uses a single file containing a header and block data. The header contains a bitmap that maps already read blocks to specific locations within the copy-on-read file. New read blocks will be appended to the file and a new bitmap entry will be added. When initializing a copy-on-write file, Xmount calculates the amount of blocks within the original block device (or image) and makes sure the bitmap in the header is large enough to be able to map each block.

Enabling copy-on-read introduces a few changes into the source code. Firstly, the possibility of specifying an optional copy-on-read file has to be added. This file has to be opened/created upon running Xmount. Secondly, an additional check has to be done before reading data from the original source if the block is not already stored in the copy-on-read file (listing 2). Finally, read blocks have to be stored to the file upon reading from the original source (listing 3).

Implementation

A patch is developed for Xmount that adds copy-on-read functionality. Implementation was more difficult than expected because the original version has a lot of compatible image types and there are a lot of dependencies within Xmount.

```
static int GetRealVirtImageData(char *buf, off_t offset, size_t size, int *readRawData) {
  ...
  if(XMountConfData.Writable==TRUE &&
        pCacheFileBlockIndex[CurBlock].Assigned==TRUE)
  {
    // Block is changed and will be read from cache/copy-on-write file.
  } else if(XMountConfData.CopyOnRead==TRUE &&
               pCoRFileBlockIndex[CurBlock].Assigned==TRUE)
  {
    // Block is already read and will be read from copy-on-read file.
  } else {
      // Block not read before read from original file and set readFromSource when
    XMountConfData.CopyOnRead==TRUE.
      *readFromSource = true;
  }
  ...
  return read;
}
```

Listing 2: Parts of modified Xmount that show global working of the copy-on-read system.

```
static int GetVirtImageData(char *buf, off_t offset, size_t size) {

    // Define read from source integer.
    int readFromSource=0;

    int read = GetRealVirtImageData(buf, offset, size, &readFromSource);

    // No copy-on-read
    if(! XMountConfData.CopyOnRead)
        return read;

    // If data is read from original source, write to copy-on-read file

    if(readFromSource == 1){
        // Write data
        int write = SetCoRImageData(buf,offset,size);

        // Compare read and write size.
        if(read != write){
            LOG_ERROR("Write to CoR not successfull.");
            return -1;
        }
    }

    return read;
}
```

Listing 3: Part of modified Xmount that copies on read.

Testing OS

This section describes the test of the patched version of Xmount. For a full overview on performed steps, see appendix B.

The implemented patched version of Xmount allows to provide new copy-on-read parameters;

```
xmount --cache copy-on-write.file --copy-on-read copy-on-read.file device mountdir
xmount --cache copy-on-write.file --copy-on-read-ro copy-on-read.file device mountdir
```

Listing 4: Using both CoW and CoR

The first step is to mount the Ubuntu block device and boot it. The next step is to re-mount the image without actually using the original image itself. As Xmount performs a few additional checks by default, the header of the image has to be preserved as well as the length. Xmount compares the image header via an MD5sum with the headers in the cache and copy-on-read files. Xmount also seeks towards the known size of the image to see if the provided image is at least the same size. The following listings show how this can be achieved.

```
dd if=image of=headerbackup bs=512 count=20
```

Listing 5: Making a backup of the header.

```
cp headerbackup fakeimage
dd if=/dev/null of=fakeimage bs=512 count=20 seek=$((originalsize/512))
```

Listing 6: Making a working mountfile of the header

Second approach

Some problems occurred when trying to remount an image with only the copy-on-read file combined with a “fresh” cache file. Hence, another approach was implemented in order to find out if this works better. The second approach writes read blocks from the original source to copy-on-read file directly when read. However, the same problems occur when testing it in the same way. For now, the best method to remount is to use a copy of the existing copy-on-read file as either cache file or as copy-on-read file to be able to boot a writable or read-only environment, respectively.

```
static int GetVirtImageData(char *buf, off_t offset, size_t size) {

  ...

  if(XMountConfData.Writable==TRUE &&
        pCacheFileBlockIndex[CurBlock].Assigned==TRUE)
  {
    // Block is changed and will be read from cache/copy-on-write file.
  } else if(XMountConfData.CopyOnRead==TRUE &&
                pCoRFileBlockIndex[CurBlock].Assigned==TRUE)
  {
    // Block is already read and will be read from copy-on-read file.
  } else {

      // No write support or data not cached
        if(GetOrigImageData(buf,
                            FileOff,
                            CurToRead)!=CurToRead)
        {
          LOG_ERROR("Couldn't read data from input image!\n")
          return -1;
        }

        if(XMountConfData.CopyOnReadWriteable)
          if(SetCoRImageData(buf,FileOff,CurToRead) !=CurToRead){
              LOG_ERROR("Couldn't write data to copy-on-read file.");
              return -1;
          }
        LOG_DEBUG("Read %zd bytes at offset %" PRIu64
                " from original image file\n",CurToRead,
  }

  ...

  return read;
}
```

Listing 7: Part of the second approach.

Random block read test

The random read test performed satisfactorily (see Appendix B.4 for test and results). All of the read blocks were exactly the same at all times, even when the original image was not present.

2.3.6. Fusecow

Fusecow is also open source and published under the GNU license. The approach is to modify the source so that it allows for copy-on-read-and-write. This is similar to the Xmount approach; use its existing method to store read blocks in a separate file. This requires an additional parameter for the copy-on-read file and an optional parameter for its map file. If the latter is not provided, the same file name will be used as the copy-on-read file appended with *.map*. Other requirements are write calls to the copy-on-read and map file when a "fresh" block is read from the original source, and a "hook" so that the map file is checked for block existence before reading from the original source.

Code

Compared to Xmount, fusecow was very easy to modify. In order to keep the code readable, all the copy-on-write features where renamed so that "write" occurred in the functions and variables and additional copy-on-read features where added. The most important code changes are listed below.

```
static int fusecow_read(const char *path, char *buf, size_t size,
                    off64_t offset, struct fuse_file_info *fi)
{
    int res;

    long long int block_number = offset / block_size;
    if(offset + size > (block_number+1)*block_size) {
    size = (block_number+1)*block_size - offset; // read only one block
    }

    if(write_map_get(block_number)) {
    res=pread64(fd_write, buf, size, offset);
    } else if(read_map_get(block_number)) {
    res=pread64(fd_read, buf, size, offset);
    } else {
    res=pread64(fd, buf, size, offset);
    // Data was never read yet, write to copy-on-read file.
    if (res != write_read(buf, size, offset) && read_only)
        res = -errno;
    }

    if (res == -1)
        res = -errno;

    return res;

}
```

Testing OS

During testing, it is found to be really easy to mount and remount a block device, even when providing a new copy-on-write file to store changes. The only problem is that by default, fusecow reads the created map file into memory and overwrites the existing one. An additional read only option (-RO) disables writing to the copy-on-read file and related map file itself.

Random block read test

The random read test performed satisfactorily here too (see appendix C.4 for test and results). All of the read blocks were exactly the same at all times, even when the original image was not present.

2.3.7. Comparison

For both methods, a proof of concept is implemented. Table 1 gives a brief comparison.

Description	Xmount	Fusecow
Difficulty of the implementation process of the p.o.c.[8]	Hard	Easy
Is read data persistent.	Excellent	Excellent
Types of block devices this method can handle	Many[9]	Raw only
Remounting an image without having the original image present	Difficult	Easy

Table 1: Comparing proof-of-concepts.

[8] Proof-of-concept

[9] Does support a lot of different virtual disk types and has the ability to remap these on the fly.

3. Conclusion

This section describes the conclusions per research question.

What methods exist that allow copy-on-write and copy-on-read on block device level?

No method exists that allows copy-on-read and copy-on-write simultaneously on block device level. Therefore, a new method is required that allows for both. This concept is introduced as CoRaW (copy-on-read-and-write). In order to implement a new method, research has been done regarding existing methods that do either copy-on-read or copy-on-write. Modifying an existing method will save a significant amount of time into researching a complete new method. The possible methods that have been found are Xmount, fusecow and Bcache.

Can these methods be effectively used to perform remote data acquisition while storing read and changed data locally?

The possible methods have been code-reviewed and have been tested upon their functionality. It has been found that Bcache does not meet the requirements for this research as it requires a non-Bcache enabled block device to be altered. Both Xmount and fusecow are open source and seem understandable enough to alter in order to enable a copy-on-read functionality. It has been decided that for both Xmount as fusecow a proof-of-concept patch will be implemented in order to find out a possible best method.

If necessary, how can an existing method be modified in order to meet the requirements of this research?

While developing an implementation upon Xmount, it is found that existing code is more difficult than expected. Developing a copy-on-read method itself is successfully achieved, but remounting the created sparse file in a way that it can be combined with a fresh copy-on-write file is still a challenge. Therefore, more research is required regarding this implementation. However, random block reads show that the storage of modified Xmount is persistent. Read blocks are retrieved successfully and match the original mount point and raw block device after remounting the copy-on-read file without the presence of the original image.

Compared to Xmount, implementing copy-on-read in fusecow was less difficult. A great benefit from this implementation is that it does handle remounting of an existing copy-on-read file along with a “fresh” copy-on-write file without any problems. As with Xmount, data stored by the fusecow implementation is also persistent. It is found that fusecow is a good base to program upon as it is simple and does its job.

What is a good way to mount block devices read only and store read and changed data into separate sparse files?

Both altered Xmount as well as fusecoraw[10] perform satisfactory regarding data integrity and live forensics, and can therefore be used to mount a block device read only and store read and changed data into separate sparse files in a good way. For now, the best working method is fusecoraw as it is the only solution that allows for remounting a copy-on-read file writable (and using a new empty copy-on-write file). With some future work, Xmount should be capable of doing this as well. As Xmount has the benefit of a larger community and has already proven itself within the forensic world, Xmount is preferred.

[10] Modified version of fusecow that implements the CoRaW concept.

4. Future work

This section describes possible future work.

4.1. Xmount

The proof of concept developed for Xmount works as intended. It stores read blocks in a persistent way. However, remounting it with a new cache file does not work very well. It also requires some “tricks” so that Xmount will actually use an existing copy-on-read file along with the passed “fake” block device. A nice feature for future research would be an additional option that allows to choose an existing copy-on-read file as source block device.

4.2. Fusecoraw

Fusecoraw works as designed. Compared to Xmount, remounting an existing copy-on-read file is a lot easier. If /dev/zero is provided as block device along with an existing copy-on-read file, desired behavior is achieved. As with Xmount, a feature that enables to mount an existing copy-on-read file as source would be nice.

4.3. Concept implementation

As this research is preformed as a sub research of a larger concept, integration within this concept is not performed yet. Extended research can be done in integrating the whole into an acquisition server. For example; it would be useful to allow a police officer to boot up a device from a USB stick, that automatically connects to the server and provides it with raw disk access via iSCSI. The server then uses a CoRaW technique to mount the original device and uses QEMU to boot it up. Finally, the server provides a VNC[11] server and allows the police officer to connect. Live acquisition can now be performed without interfering with the original device and found evidence is stored immediately.

[11] https://en.wikipedia.org/wiki/Virtual_Network_Computing

References

[1] Diego Calleja. Linux 3.10, 2013. http://kernelnewbies.org/Linux_3.10.

[2] Gillen Daniel. xmount, 2008. https://www.pinguin.lu/index.php.

[3] NIST Cloud Computing Forensic Science Working Group. Nist cloud computing forensic science challenge (draft), 2014. http://csrc.nist.gov/publications/drafts/nistir-8006/draft_nistir_8006.pdf.

[4] Cannonical Ltd. Ubuntu server 14.04, 2014. http://www.ubuntu.com/server.

[5] Kent Overstreet. Bcache, 2014. http://bcache.evilpiepirate.org/.

[6] Vitaly Shukela. fusecow, 2011. https://github.com/vi/fusecow.

[7] Miklos Szeredi. Fuse v2, 2004. http://fuse.sourceforge.net/.

[8] Eric van den Haak. Evdh's git repository, 2014. https://github.com/evdh-nl.

[9] Yongwei Wu Kang Chen Jinlei Jiang Member IEEE Xun Zhao, Yang Zhang and IEEE Keqin Li, Senior Member. Liquid: A scalable deduplication file system for virtual machine images. 2014.

Appendices

A. Test scripts

A.1. Preparing small volume

```
#!/bin/sh

# Create 100M logical volume
lvcreate --size 100M --name small ehvg

# Add file system
mkfs.ext4 /dev/ehvg/small

# Mount file system
[[ -d /mnt/test ]] || mkdir /mnt/test
mount /dev/ehvg/small /mnt/test

# Add some stuff
mkdir /mnt/test/testdir
echo -e "Hi,\n\nI am a testfile.\n\nGr,\n\nE." > /mnt/test/testfile

# Unmount it
umount /mnt/test
```

Listing 8: Setup LVM test

A.2. Preparing ubuntu volume

```
#!/bin/sh

# Create 4G logical volume
lvcreate --size 4G --name ubuntu ehvg
```

Listing 9: Another logical volume is setup to use as VM storage for an Ubuntu virtual machine.

A.3. Testing Xmount

```
#!/bin/sh

# Create mountdir and copy-on-write file
touch /tmp/rp2/xmnt.small.cache
mkdir /tmp/rp2/xmnt.small.mnt

# Mount logical volume via Xmount
xmount --cache /tmp/rp2/xmnt.small.cache /dev/ehvg/small /tmp/rp2/xmnt.small.mnt
mount /tmp/rp2/xmnt.small.mnt/small.dd /mnt/test

# Show output
```

```
cat /mnt/test/testfile
echo -e "\n---\n"

# Change file
echo -e "Hi,\n\nI am a testfile.\n\nGr,\n\nEvdH" > /mnt/test/testfile

# Unmount
umount /mnt/test
umount /tmp/rp2/xmnt.small.mnt

# Mount and check original volume
mount /dev/ehvg/small /mnt/test -o ro
cat /mnt/test/testfile
echo -e "\n---\n"

# Unmount
umount /mnt/test

# Remount via fusecow
xmount --cache /tmp/rp2/xmnt.small.cache /dev/ehvg/small /tmp/rp2/xmnt.small.mnt
mount /tmp/rp2/xmnt.small.mnt/small.dd /mnt/test

# Show output
cat /mnt/test/testfile
echo -e "\n---\n"

# Show file status
du -sh /tmp/rp2/xmnt.small.cache
du -sh /tmp/rp2/xmnt.small.mnt/small.dd
```

Listing 10: Testing Xmount

Output;

```
Hi,

I am a testfile.

Gr,

E.

---

Hi,

I am a testfile.

Gr,

E.

---

Hi,

I am a testfile.
```

```
Gr,

EvdH

---

3.1M    /tmp/rp2/xmnt.small.cache
0       /tmp/rp2/xmnt.small.mnt/small.dd
```

A.4. Testing fusecow

```
#!/bin/sh

# Create mountpoint and copy-on-write file
touch /tmp/rp2/fsc.small.{cow,mnt}

# Mount logical volume via fusecow
fusecow /dev/ehvg/small /tmp/rp2/fsc.small.mnt /tmp/rp2
mount /tmp/rp2/fsc.small.mnt /mnt/test

# Show output
cat /mnt/test/testfile
echo -e "\n---\n"

# Change file
echo -e "Hi,\n\nI am a testfile.\n\nGr,\n\nEvdH" > /mnt/test/testfile

# Unmount
umount /mnt/test
umount tmp/rp2/fsc.small.mnt

# Mount and check original volume
mount /dev/ehvg/small /mnt/test -o ro
cat /mnt/test/testfile
echo -e "\n---\n"

# Unmount
umount /mnt/test

# Remount via fusecow
fusecow /dev/ehvg/small /tmp/rp2/fsc.small.mnt /tmp/rp2
mount /tmp/rp2/fsc.small.mnt /mnt/test

# Show output
cat /mnt/test/testfile
echo -e "\n---\n"

# Show file status
du -sh /tmp/rp2/fsc.small.cow
du -sh /tmp/rp2/fsc.small.cow.map
```

Listing 11: Testing fusecow

Output;

```
Hi,

I am a testfile.

Gr,

E.

---

Hi,

I am a testfile.

Gr,

E.

---

Hi,

I am a testfile.

Gr,

EvdH

---

72K     /tmp/rp2/fsc.small.cow
4.0K    /tmp/rp2/fsc.small.cow.map
```

A.5. Linux random write test

```
#!/bin/bash

# Do some writing to tmp;
mkdir /tmp/testfiles
for i in {0..99}; do dd if=/dev/urandom of=/tmp/testfiles/${i} bs=512 count=20000; done
```

Listing 12: Write test

B. Patched Xmount test

This appendix shows the performed steps in order to test the patched version of Xmount.

B.1. Setup

```
mkdir /mnt/rp/xmnt.ubuntu.mnt
xmount --cache /mnt/rp/xmnt.ubuntu.cow --copy-on-read /mnt/rp/xmnt.ubuntu.cor /dev/ehvg/ubuntu /
    mnt/rp/xmnt.ubuntu.mnt
startvm
```

Listing 13: Setting up test environment

Output

```
DEBUG: ParseCmdLine.0.4.7@380 : Enabling virtual write support using cache file "/mnt/rp/xmnt.ubuntu.cow"
DEBUG: ParseCmdLine.0.4.7@394 : Enabling copy-on-read support using CoR file "/mnt/rp/xmnt.ubuntu.cor"
DEBUG: main.0.4.7@3518 : Options passed to FUSE: xmount /mnt/rp/xmnt.ubuntu.mnt -o subtype=xmount,fsname=/dev/ehvg/ub
DEBUG: main.0.4.7@3547 : Loading image file "/dev/ehvg/ubuntu"...
DEBUG: main.0.4.7@3622 : Input image file opened successfully
DEBUG: GetOrigImageData.0.4.7@800 : Read 10485760 bytes at offset 0 from DD file
DEBUG: main.0.4.7@3633 : Partial MD5 hash of input image file: 4d31d9471bcb1c38a82c1ddd65899467
DEBUG: ExtractVirtFileNames.0.4.7@625 : Set virtual image name to "/ubuntu.dd"
DEBUG: ExtractVirtFileNames.0.4.7@627 : Set virtual image info name to "/ubuntu.info"
DEBUG: main.0.4.7@3645 : Virtual file names extracted successfully
DEBUG: main.0.4.7@3652 : Virtual image info file build successfully
DEBUG: InitCacheFile.0.4.7@3183 : Cache blocks: 4096 (1000) entries, 49152 (0000C000) bytes
DEBUG: InitCacheFile.0.4.7@3193 : Cache file has 0 bytes
DEBUG: InitCacheFile.0.4.7@3251 : Cache file is empty. Generating new block header
DEBUG: main.0.4.7@3691 : Cache file initialized successfully
DEBUG: InitCoRFile.0.4.7@3342 : Copy-on-Read blocks: 4096 (1000) entries, 49152 (0000C000) bytes
DEBUG: InitCoRFile.0.4.7@3352 : copy-on-read file has 0 bytes
DEBUG: InitCoRFile.0.4.7@3410 : copy-on-read file is empty. Generating new block header
DEBUG: main.0.4.7@3700 : Copy-on-read file initialized successfully
```

B.2. After run

```
stopvm
du -sh /mnt/rp/xmnt.ubuntu.*
umount /mnt/rp/xmnt.ubuntu.mnt
```

Listing 14: Setting up test environment

Output

```
319M    xmnt.ubuntu.cor
24M     xmnt.ubuntu.cow
4.0K    xmnt.ubuntu.mnt
```

B.3. Remount

Mounting a “fake” image and run VM.

```
umount /mnt/rp/xmnt.ubuntu.mnt
dd if=/dev/ehvg/ubuntu of=/mnt/rp/xmnt.ubuntu.hdr bs=512 count=20
cp /mnt/rp/xmnt.ubuntu.hdr /mnt/rp/xmnt.ubuntu.fdd
dd if=/dev/null bs=512 of=/mnt/rp/xmnt.ubuntu.fdd seek=$((4*1024*1024*1024/512))
ls -lh /mnt/rp
du -sh /mnt/rp/*
cp /mnt/rp/xmnt.ubuntu.cor /mnt/rp/xmnt.ubuntu.cow.rerun
# use copy of copy-on-read file as new cache file.
xmount --cache /mnt/rp/xmnt.ubuntu.cow.rerun --copy-on-read-ro /mnt/rp/xmnt.ubuntu.cor /mnt/rp/
    xmnt.ubuntu.fdd /mnt/rp/xmnt.ubuntu.mnt
```

Listing 15: Re-mount with fake block device

Output

```
root@earch /mnt/rp # dd if=/dev/ehvg/ubuntu of=/mnt/rp/xmnt.ubuntu.hdr bs=512 count=20
20+0 records in
20+0 records out
10240 bytes (10 kB) copied, 0.32054 s, 31.9 kB/s
root@earch /mnt/rp # cp /mnt/rp/xmnt.ubuntu.hdr /mnt/rp/xmnt.ubuntu.fdd
root@earch /mnt/rp # dd if=/dev/null bs=512 of=/mnt/rp/xmnt.ubuntu.fdd seek=$((4*1024*1024*1024/512))
0+0 records in
0+0 records out
0 bytes (0 B) copied, 0.000112207 s, 0.0 kB/s
root@earch /mnt/rp # ls -lh /mnt/rp
total 674M
-rw-r--r-- 1 root root 324M Jun 23 13:56 xmnt.ubuntu.cor
-rw-r--r-- 1 root root  26M Jun 23 13:55 xmnt.ubuntu.cow
-rw-r--r-- 1 root root  49K Jun 23 13:52 xmnt.ubuntu.cow.orrig
-rw-r--r-- 1 root root 326M Jun 23 14:05 xmnt.ubuntu.cow.rerun
-rw-r--r-- 1 root root 4.0G Jun 23 14:11 xmnt.ubuntu.fdd
-rw-r--r-- 1 root root  10K Jun 23 14:11 xmnt.ubuntu.hdr
drwxrwxrwx 2 root root    0 Jan  1  1970 xmnt.ubuntu.mnt
root@earch /mnt/rp # du -sh /mnt/rp/*
324M    /mnt/rp/xmnt.ubuntu.cor
26M     /mnt/rp/xmnt.ubuntu.cow
52K     /mnt/rp/xmnt.ubuntu.cow.orrig
326M    /mnt/rp/xmnt.ubuntu.cow.rerun
12K     /mnt/rp/xmnt.ubuntu.fdd
12K     /mnt/rp/xmnt.ubuntu.hdr
0       /mnt/rp/xmnt.ubuntu.mnt
root@earch /mnt/rp # xmount --cache /mnt/rp/xmnt.ubuntu.cow.rerun /mnt/rp/xmnt.ubuntu.fdd /mnt/rp/xmnt.ubuntu.mnt
DEBUG: ParseCmdLine.0.4.7@380 : Enabling virtual write support using cache file "/mnt/rp/xmnt.ubuntu.cow.rerun"
DEBUG: main.0.4.7@3518 : Options passed to FUSE: xmount /mnt/rp/xmnt.ubuntu.mnt -o subtype=xmount,fsname=/mnt/rp/xmnt
DEBUG: main.0.4.7@3547 : Loading image file "/mnt/rp/xmnt.ubuntu.fdd"...
DEBUG: main.0.4.7@3622 : Input image file opened successfully
DEBUG: GetOrigImageData.0.4.7@800 : Read 10485760 bytes at offset 0 from DD file
DEBUG: main.0.4.7@3633 : Partial MD5 hash of input image file: 96dcb0744eb2fb0bfda1d7604f1f3282
DEBUG: ExtractVirtFileNames.0.4.7@625 : Set virtual image name to "/xmnt.ubuntu.dd"
DEBUG: ExtractVirtFileNames.0.4.7@627 : Set virtual image info name to "/xmnt.ubuntu.info"
DEBUG: main.0.4.7@3645 : Virtual file names extracted successfully
DEBUG: main.0.4.7@3652 : Virtual image info file build successfully
DEBUG: InitCacheFile.0.4.7@3183 : Cache blocks: 4096 (1000) entries, 49152 (0000C000) bytes
```

```
DEBUG: InitCacheFile.0.4.7@3193 : Cache file has 338739712 bytes
DEBUG: InitCacheFile.0.4.7@3197 : Cache file not empty. Parsing block header
DEBUG: main.0.4.7@3691 : Cache file initialized successfully
```

B.4. Read random blocks

```
#!/bin/bash

echo Randomblockstest.

# Clear md5 files
echo -n > /tmp/MD5SUMS_raw
echo -n > /tmp/MD5SUMS_mount

# Determine random blocks.
export random='42 512 1337 7777 12345 20802 67890 88888 500000 777777'

# Read random blocks from raw device.
for i in $random; do dd if=/dev/ehvg/ubuntu bs=512 seek=$i count=100 of=/tmp/dmp_${i} 2>/dev/
    null;  md5sum --tag /tmp/dmp_${i} >> /tmp/MD5SUMS_raw; done

# Mount device via xmount and read the same random blocks
mkdir /mnt/rp/xmnt.random.mnt
xmount --cache xmnt.random.cow --copy-on-read xmnt.random.cor /dev/ehvg/ubuntu /mnt/rp/xmnt.
    random.mnt
for i in $random; do dd if=/mnt/rp/xmnt.random.mnt/ubuntu.dd bs=512 seek=$i count=100 of=/tmp/
    dmp_${i} 2>/dev/null;  md5sum --tag /tmp/dmp_${i} >> /tmp/MD5SUMS_mount; done
umount /mnt/rp/xmnt.random.mnt

# Remount /dev/null as block device and read same random blocks

dd if=/dev/ehvg/ubuntu of=/mnt/rp/xmnt.ubuntu.hdr bs=512 count=20
cp /mnt/rp/xmnt.ubuntu.hdr /mnt/rp/xmnt.ubuntu.fdd
dd if=/dev/null bs=512 of=/mnt/rp/xmnt.ubuntu.fdd seek=$((4*1024*1024*1024/512))


xmount --copy-on-read-ro /mnt/rp/xmnt.ubuntu.cor /mnt/rp/xmnt.ubuntu.fdd /mnt/rp/xmnt.random.mnt
for i in $random; do dd if=/mnt/rp/xmnt.random.mnt/xmnt.ubuntu.dd bs=512 seek=$i count=100 of=/
    tmp/dmp_${i} 2>/dev/null ; done


# Compare latest reads with raw and original mounted block device.
md5sum -c /tmp/MD5SUMS_raw
echo
md5sum -c /tmp/MD5SUMS_mount

# Unmount and delete used files.
umount /mnt/rp/xmnt.random.mnt
rm -rf /mnt/rp/xmnt.random.*
rm -rf /tmp/dmp_*
rm /tmp/MD5SUMS_raw
rm /tmp/MD5SUMS_mount
```

Listing 16: Read random blocks from original device and xmount

Output

```
Randomblockstest.
20+0 records in
20+0 records out
10240 bytes (10 kB) copied, 0.0189198 s, 541 kB/s
0+0 records in
0+0 records out
0 bytes (0 B) copied, 6.1893e-05 s, 0.0 kB/s
/tmp/dmp_42: OK
/tmp/dmp_512: OK
/tmp/dmp_1337: OK
/tmp/dmp_7777: OK
/tmp/dmp_12345: OK
/tmp/dmp_20802: OK
/tmp/dmp_67890: OK
/tmp/dmp_88888: OK
/tmp/dmp_500000: OK
/tmp/dmp_777777: OK

/tmp/dmp_42: OK
/tmp/dmp_512: OK
/tmp/dmp_1337: OK
/tmp/dmp_7777: OK
/tmp/dmp_12345: OK
/tmp/dmp_20802: OK
/tmp/dmp_67890: OK
/tmp/dmp_88888: OK
/tmp/dmp_500000: OK
/tmp/dmp_777777: OK
```

C. Patched fusecow test

This appendix shows the performed steps in order to test the patched version of fusecow.

C.1. Setup

```
touch /mnt/rp/fsc.ubuntu.{cor,cow,mnt}
fusecoraw /dev/ehvg/ubuntu /mnt/rp/fsc.ubuntu.mnt /mnt/rp/fsc.ubuntu.cow /mnt/rp/fsc.ubuntu.cor
runvm
```

Listing 17: Setting up test environment

C.2. After run

```
stopvm
umount /mnt/rp/fsc.ubuntu.mnt
ls -alh /mnt/rp/
du -sh /mnt/rp/*
```

Listing 18: Setting up test environment

Output

```
root@earch /mnt/rp # umount /mnt/rp/fsc.ubuntu.mnt
root@earch /mnt/rp # md5sum /mnt/rp/fsc.ubuntu.cor
root@earch /mnt/rp # ls -alh /mnt/rp/
total 92M
drwxr-xr-x 3 root root 4.0K Jun 23 15:22 .
drwxr-xr-x 7 root root 4.0K Jun 20 16:59 ..
-rw-r--r-- 1 root root 4.0G Jun 23 15:23 fsc.ubuntu.cor
-rwxr-xr-x 1 root root  68K Jun 23 15:23 fsc.ubuntu.cor.map
-rw-r--r-- 1 root root 2.3G Jun 23 15:23 fsc.ubuntu.cow
-rwxr-xr-x 1 root root  68K Jun 23 15:23 fsc.ubuntu.cow.map
-rw-r--r-- 1 root root    0 Jun 23 15:22 fsc.ubuntu.mnt
drwxrwxrwx 2 root root    0 Jan  1  1970 xmnt.ubuntu.mnt
root@earch /mnt/rp # du -sh /mnt/rp/*
91M     /mnt/rp/fsc.ubuntu.cor
44K     /mnt/rp/fsc.ubuntu.cor.map
988K    /mnt/rp/fsc.ubuntu.cow
24K     /mnt/rp/fsc.ubuntu.cow.map
0       /mnt/rp/fsc.ubuntu.mnt
0       /mnt/rp/xmnt.ubuntu.mnt
```

C.3. Remount

```
touch /mnt/rp/fsc.ubuntu.cow.remount
fusecoraw /dev/null /mnt/rp/fsc.ubuntu.mnt /mnt/rp/fsc.ubuntu.cow.remount /mnt/rp/fsc.ubuntu.cor
      -RO
startvm
```

Listing 19: Remount and rerun vm

C.4. Read random blocks

```
#!/bin/bash

echo Randomblockstest.

# Clear md5 files
echo -n > /tmp/MD5SUMS_raw
echo -n > /tmp/MD5SUMS_mount

# Determine random blocks.
export random='42 512 1337 7777 12345 20802 67890 88888 500000 777777'

# Read random blocks from raw device.
for i in $random; do dd if=/dev/ehvg/ubuntu bs=512 seek=$i count=100 2>/dev/null > /tmp/dmp_${i
    };  md5sum --tag /tmp/dmp_${i} >> /tmp/MD5SUMS_raw; done

# Mount device via fusecoraw and read the same random blocks
touch /mnt/rp/fsc.random.{cor,cow,mnt}
fusecoraw /dev/ehvg/ubuntu /mnt/rp/fsc.random.mnt /mnt/rp/fsc.random.cow /mnt/rp/fsc.random.cor
for i in $random; do dd if=/mnt/rp/fsc.random.mnt bs=512 seek=$i count=100 2>/dev/null > /tmp/
    dmp_${i};  md5sum --tag /tmp/dmp_${i} >> /tmp/MD5SUMS_mount; done
umount /mnt/rp/fsc.ubuntu.mnt

# Remount /dev/null as block device and read same random blocks
touch /mnt/rp/fsc.ubuntu.cow.remount
fusecoraw /dev/null /mnt/rp/fsc.ubuntu.mnt /mnt/rp/fsc.ubuntu.cow.remount /mnt/rp/fsc.ubuntu.cor
      -RO
for i in $random; do dd if=/mnt/rp/fsc.random.mnt bs=512 seek=$i count=100 2>/dev/null > /tmp/
    dmp_${i}; done

# Compare latest reads with raw and original mounted block device
md5sum -c /tmp/MD5SUMS_raw
echo
md5sum -c /tmp/MD5SUMS_mount

# Unmount and delete used files.
umount /mnt/rp/fsc.random.mnt
rm -rf /mnt/rp/fsc.random.*
rm -rf /tmp/dmp_*
rm /tmp/MD5SUMS_raw
rm /tmp/MD5SUMS_mount
```

Listing 20: Read random blocks from original device and fusecoraw

Output

```
Randomblockstest.
/tmp/dmp_42: OK
```

```
/tmp/dmp_512: OK
/tmp/dmp_1337: OK
/tmp/dmp_7777: OK
/tmp/dmp_12345: OK
/tmp/dmp_20802: OK
/tmp/dmp_67890: OK
/tmp/dmp_88888: OK
/tmp/dmp_500000: OK
/tmp/dmp_777777: OK

/tmp/dmp_42: OK
/tmp/dmp_512: OK
/tmp/dmp_1337: OK
/tmp/dmp_7777: OK
/tmp/dmp_12345: OK
/tmp/dmp_20802: OK
/tmp/dmp_67890: OK
/tmp/dmp_88888: OK
/tmp/dmp_500000: OK
/tmp/dmp_777777: OK
```