UNIVERSITY OF AMSTERDAM

Master in System and Network Engineering

Final Research Project

Pre-boot RAM acquisition and compression

Student: Martijn Bogaard
martijn.bogaard@os3.nl

Supervisor: Ruud Schramp
schramp@holmes.nl

June 2015

Abstract

This paper presents a novel memory acquisition algorithm for cold boot attacks to compress the RAM of a system in the boot loader, before the operating system is loaded. This makes efficient memory acquisition possible using minimal, but fully featured Linux based operating systems. The benefit of this approach is that the acquisition of the memory during a forensic investigation can be combined with the collection of other evidence from for example connected storage devices and other media. This project was initiated as part of a larger project of the Netherlands Forensics Institute that is currently developing a custom solution for automated evidence collection and network based transmission.

As part of this project an extensive data compression algorithm comparison has been conducted to select the optimal algorithm for this project. The compression algorithms have been tested with RAM dumps generated from scenarios that have been created to have samples of the wide spectrum of possibilities regarding the content of the RAM and found entropy. Based on the challenges identified in this project a new acquisition algorithm has been developed that should recover the majority of the RAM even under the most difficult conditions.

The memory acquisition algorithm has been realized in a proof of concept and the results are compared against the current state-of-the-art solutions. The amount of data that is overwritten before it can be secured is (slightly) increased compared to the existing solutions. However, a significant improvement has been realised compared with booting an unmodified operating system.

Contents

1 Introduction

During forensic investigations it is common practice to retrieve the content of the memory of a running system. This is the place where advanced malware hides, suspects leave digital traces regarding their recent computer usage and also contains the crypto keys for the increasingly common full disk encryption. While several techniques (like DMA and (cold) boot attacks) are available for this, none of them work in all possible scenarios and in most cases only a single attempt to retrieve the data will be possible.

On some systems the memory is not cleared when the system is booted. While in most cases the memory will be empty because the content of the RAM is not refreshed when the system is turned off, there is a short time window (possibly extended by cooling the RAM) where the system can be rebooted and a memory acquisition tool can be used to retrieve its content.

The tools to acquire the memory can be divided in two groups. The first group consists of tools that are part of an (forensic) operating system that is booted. The downside of this approach is that this will destroy a part of the memory content to load the operating system. This will mostly be in the lower memory regions where (for forensic purposes) interesting data can be found such as memory page tables. An alternative approach is the usage of a bootable, self containing tool that has a very small memory footprint and dumps the memory.

Several of such tools have already existed for some years, such as msramdmp[1] and {bios, efi}_memimage[2]. Their downside is that because they try to keep their memory footprint as low as possible the functionality and abilities to store the retrieved data is pretty limited. For example, storing the memory dump on the USB drive that was used to boot the tool or over the network using a very limited driver and network stack. This results in a limited portability and time-consuming effort if the memory of many machines has to be retrieved.

The Netherlands Forensic Institute proposed the idea to research the feasibility of adding a pre-boot compression stage before the forensic operating system to compress the content of the memory and move the content of the lower memory regions to higher memory regions and protect them against the booting operating system.

1.1 Research question

The main research question to answer during this project is: *Is pre-boot compression a useful technique to reduce the destruction of data when an operating system is loaded*?

To answer the research question, the following sub-questions have been formulated:

- How much memory is used by the firmware (e.g. BIOS or UEFI) before the operating system is started?
- Does this amount depend on the firmware type of the system?
- How compressible is the data that is typically found in RAM?
- What is the optimal algorithm to compress data considering maximal compression using a minimal amount of memory during execution?

[1] http://mcgrewsecurity.com/oldsite/projects/msramdmp.1.html
[2] http://citp.princeton.edu/research/memory/code

- In which way can a memory region be hidden from or protected against a booting operating system?
- How can the compressed data be retrieved after the operating system is started?
- How can the validity of the decompressed data be proven?

1.2 Related work

As part of his master thesis regarding the forensic analysis of memory dumps Kollr [25] has created an overview of different methods to acquire such dumps. It lists several methods, including DMA attacks using FireWire and special insert cards and cold and hot boot attacks.

While the theory for a cold boot attack was not new, the attack was accomplished for the first time by Halderman *et al.* [19] in 2008. Because their tools were initially not released, McGrew decided [32] to create an independent implementation of their attack and released the tool *msramdmp*.

The usefulness of cold boot attacks for forensic purposes was studied during a research project by Hannay and Woodward [21] and a project by Carbone *et al.* [8]. While in some cases it is the only option a forensic investigator has, the method has serious limitations and drawbacks, and mixed success was achieved.

Besides the previously mentioned specialized tools to dump the memory of a system is also looked into the usage of forensic operating systems. Such systems exist and are commonly Linux based systems that boot from CD, DVD or USB drive into a ramdisk. A longer, on-going project of The Netherlands Forensic Institute is to acquire the evidence of a computer system over the network. Last year, Cortjens [12] and van den Haak [48] looked into the feasibility of such system to do automated forensic acquisition of the local storage devices. The results of this project will be used to integrate support for memory acquisitions.

The compression of the RAM content of a system is hardly a new concept. While in the past it was mainly used because RAM was expensive and swapping to disk is slow and power consuming, even modern operating systems [36] use it to make more efficient use of the available resources by compressing in-active applications. This resulted in the development [37, 45] of special compression algorithms by the academic community.

2 Background Information

As introduction to the concepts used later on in this research project, several of them are explained in this chapter. First, a general overview of memory forensics and data compression will be given. Then the memory layout of common modern operating systems will be discussed.

2.1 Memory Forensics

Getting a copy of the hard drive of the involved machine(s) has been a long-standing procedure [35, p. 28] in digital forensics. However, the increasing popularity of full disk encryption and advancing malware makes this no longer sufficient [10]. Without the crypto keys which are lost when a machine is powered off, it will be an (almost) worthless effort to break or bypass the encryption and gain access. Because the crypto keys and other valuable data reside in the memory of the system, methods to extract the content of the system memory have seen an increased interest in the last decade.

2.1.1 Typical Content

For a forensic analyst the memory is a goldmine with information. Based on the information that is commonly [6] found, it is possible to reconstruct the last actions of the user, the files that are used and sometimes even previous versions.

The memory of a system contains all the live data of that system. It is the location of the operating system, the running applications, opened files, crypto keys and all other objects needed for a correct functioning system. When certain data is needed during the usage of the system, it is loaded into the memory where it remains until it is no longer needed and gets overwritten with other data. How long it takes before the old data is overwritten depends [6, p. 9] on how actively the system is used.

Some modern operating systems even try to predict [22] the user's next action. By loading files from the disk that the user is likely to use based on earlier behaviour, the responsiveness and experienced performance of the system is increased. At the same time, this reveals information about the usage patterns of the user as certain metadata such as usage counters are stored to improve this mechanism.

It is also the location where malware hides. Advanced malware will not leave traces on the disk [29], but successful hiding in the memory of a system is almost impossible. If the malware wants to be executed, it must be somewhere in the memory of the system leaving traces of modified OS components or inconsistent data structures. Even when it cannot easily be found, because the malware consists only of malicious data instead of executable code [51].

2.1.2 Acquisition Methods

There are multiple ways to get a copy of the memory of a system. In 2013, Osborn gave [33] an overview of the known techniques and their characteristics.

When the system is unlocked and trusted, an application whether or not combined with a kernel driver can be used to create a dump of the memory. However, this will not be the case in most cases during a forensic investigation. Either the system is locked and the user is not willing to give his password or the system cannot be trusted because installing and running an application could trigger a booby trap that shuts down the system and/or destroys all data.

Multiple hardware-based methods have been developed of which the DMA attack using FireWire [31] is best known. This is not the only way as several of the busses in a system allow peripherals to read and/or write to the memory of a system without intervention of the CPU. This enabled the possibility to develop dedicated insert cards for the PCI [9] and PCIe [3] busses to copy the memory.

Hardware based methods have the risk that they can crash [20] the system during the acquisition. Modern systems have the possibility to block the attack by preventing unauthorized access to the RAM using the IOMMU [53]. As there is in general only a single attempt possible, software based approaches could have the preference.

Some systems do not overwrite the memory during the initialization of the system. This opens the door for a (cold) boot attack [19] where the system is rebooted or reset and another (micro) operating system is started.

When this is not possible, for example because the system firmware (i.e. the BIOS) reinitializes the memory, the same research team proposed the idea to cool down the memory module, remove it from the system and use another system to extract the content. Almost two years ago, a different approach was demonstrated [42] where a custom version of the system firmware was developed and used to acquire a memory dump.

2.1.3 Data Integrity

One of the challenges [23] of digital forensics is to prove later on the evidence has not been changed or manipulated. Cryptographic hashes are calculated during or soon after the acquisition of the data. Such hashes, like MD5 and SHA-1, can later be used to compare the data against and confirm it has not been changed. The change of only a single bit results in a totally different hash value.

Cryptanalysis in the last two decades revealed weaknesses [47, 15] in both algorithms. Stronger alternatives like SHA-2 and SHA-3 are available, but MD5 and SHA1 remain widely in use for forensic purposes as intentional and meaningful collisions are hard to introduce [26].

2.2 Data Compression

Data compression is used to store data in less space than needed before the compression. This is possible by encoding the same information in fewer bits or removing information that is not entirely necessary as is commonly done with, for example, audio, video or image compression. However, this requires insight in the data that is compressed and what must be preserved or is acceptable to remove. This project focuses on data compression where the original data can be recovered by decompressing it, so called lossless compression.

The ideas still used today in compression algorithms go back many years, even from before the invention of the computer. One of these ideas is the usage of short codes for commonly used words and expressions. Braille and Morse code can be seen [13, p. 17] as one of the first methods to compress textual information.

Two important groups of algorithms for general-purpose data compression are the statistical based and dictionary based algorithms. The first group [13, p. 47] uses variable length codes assigned to the symbols they represent, based on how frequently they occur in the data. By assigning shorter codes to (a group of) symbols that occur more frequently, less data is needed to represent the same information. The second group uses fixed size dictionaries [13, p. 171], which could be static or dynamic depending on the algorithm. If an entry of the dictionary is found, it is replaced with the corresponding index. Otherwise the original data are used together with a flag that the decoder shouldn't interpret the following symbols as an index.

2.2.1 Compression Algorithms

Only a few algorithms form the building blocks and inspiration for most currently widely used algorithms like those that are used in the gzip, zip or 7z formats. One of those building blocks is RLE, Run Length Encoding. It replaces sequences of characters with their occurrence. A very simple example would be to encode `aaaaa` as `5a`.

In 1948 Claude Shannon published the very famous paper *A Mathematical Theory of Communication* [44]. Not only was it the start of a whole new field of applied mathematics, called information theory, it also introduced an algorithm developed by Robert Fano. It creates a binary tree sorted by the occurrence of the symbols. Based on the position in the tree, each symbol is assigned a code. A variant of this algorithm that gives slightly better results by building the tree in a different way was proposed [24] by David Huffman in 1952. Even today, his method, called Huffman coding, is still widely used in DEFLATE [14], JPEG [52], MP3 [7] and many others.

LZ77 is another "ancestor" (and in fact, DEFLATE is a combination of LZ77 and Huffman coding) of many newer algorithms. It was designed [56] in 1977 by Jacob Ziv and Abraham Lemper and together with LZ78 [57] it was the start of dictionary-based compression. It works by a fixed-size *sliding window* which forms the current dictionary. The encoding works by outputting relative offsets to corresponding symbols into the dictionary or the original data if no substitute can be found.

Many, many more algorithms have followed resulting in hundreds of algorithms in thousands of applications and file formats. While some newer, better methods are developed such as arithmetic coding, patents and the complexity of some algorithms prevented [40, p. 5] wide adoption.

2.2.2 Shannon Entropy

In the same paper where Shannon introduced information theory, he proposed [44, p. 14] a new measuring unit to express the (un)predictability of information. The idea [34] is that when the probability of a symbol is relatively high, the information a message can carry is low. The Shannon entropy limits this way the (theoretical) best possible lossless compression of the data. This makes it to some level an interesting metric to have a (rough) estimate about the compressibility of certain data.

2.3 Memory Layout

It is inevitable that when the content of the RAM of a system is acquired using a method that requires an application or operating system running from that same RAM, a part of it will be overwritten. For this reason it is interesting to know how the physical layout of common operating systems such as Linux and Windows look like and what the impact of the system firmware on this is.

The study of how the RAM of a system should be managed is a field on its own because of the performance implications it has on the system. Several [30] books [18] have been published that give an insight in how an operating system manages it, but most of this information is (to some degree) outdated.

Most information is known about the virtual memory layout of operating systems. Detailed information about how this maps to the physical memory is much rarer. An attempt is made to create a high-level overview based on this information and empirical research on several machines using the RamMap tool[3] for Windows and the information that can be found in `/proc` on Linux based systems.

2.3.1 System Firmware

Before the loader of the operating system starts, the firmware of the system (e.g. BIOS, UEFI, Coreboot) is executed first. It is located on a flash chip that is mapped in the address space of the system. Therefore, it has no impact on the content of the RAM. Only the data that are generated and used by the firmware is written to the RAM.

However, the space on the flash chip is limited. This resulted in the compression [41, p. 107] of parts [4, p. 949] of the firmware. This has to be decompressed into the memory first before it can be executed.

The size of the data and decompressed parts remain largely unknown. Intel published [54] a whitepaper with some numbers as part of their effort to optimize the reference UEFI implementation[4] for their Quark platform[5] that is intended to be used in wearables and other low resource devices. This whitepaper states that a stripped down version (that fits compressed in 64 kilobytes) of the UEFI firmware requires 624 kilobytes for the data and 136 kilobytes for the decompressed firmware components.

The firmware found in the average computer system is much larger. Therefore, it can be expected that the memory usage is also much higher. Browsing through a site[6] that offers modified versions of the firmware of many systems, reveals sizes up to eight megabytes. Another whitepaper [53] from Intel where the memory map of "a typical" UEFI system during several phases of the initialization is discussed shows a situation where 232 megabytes of RAM is reserved by the UEFI firmware. However, reserved memory is not the same as used or overwritten memory and more information about this is very rare. Listing 1 shows two examples of typical memory maps, one for a BIOS based system and one for a UEFI based system.

2.3.2 Linux

The Linux kernel divides the physical memory in several zones [28, p. 233] depending on the architecture and memory configuration of the system. The reason for this is that some devices have limitations on which addresses they can use for DMA transfers and that especially on 32-bit systems the virtual address space is limited. While for a major part of the memory it is not a problem to be paged out, many kernel-space structures

[3] https://technet.microsoft.com/en-us/library/ff700229.aspx
[4] http://www.tianocore.org
[5] http://www.intel.com/content/www/us/en/embedded/products/quark/overview.html
[6] https://www.bios-mods.com/BIOS

```
e820:  BIOS-provided physical RAM map: Type       Start            End
[mem 0x00000000-0x0008efff] usable     BS Code    0000000000000000-0000000000000FFF
[mem 0x0008f000-0x0008ffff] reserved   Available  0000000000001000-000000000003CFFF
[mem 0x00090000-0x0009e7ff] usable     BS Code    000000000003D000-0000000000057FFF
[mem 0x0009e800-0x0009ffff] reserved   Reserved   0000000000058000-0000000000058FFF
[mem 0x000e0000-0x000fffff] reserved   Available  0000000000059000-000000000005FFFF
[mem 0x00100000-0x7ee94fff] usable     BS Code    0000000000060000-0000000000087FFF
[mem 0x7ee95000-0x7eebefff] reserved   BS Data    0000000000088000-0000000000088FFF
[mem 0x7eebf000-0x7eee3fff] usable     BS Code    0000000000089000-000000000009EFFF
[mem 0x7eee4000-0x7efbefff] ACPI NVS   Reserved   000000000009F000-000000000009FFFF
[mem 0x7efbf000-0x7efeefff] usable     Available  0000000000100000-000000000FFFFFFF
[mem 0x7efef000-0x7effefff] ACPI data  BS Code    0000000010000000-000000001000AFFF
[mem 0x7efff000-0x7effffff] usable     Available  000000001000B000-000000006AAFCFFF
[mem 0x7f000000-0x7fffffff] reserved   BS Data    000000006AAFD000-000000006AFAAFFF
[mem 0xe0000000-0xe3ffffff] reserved   Available  000000006AFAB000-000000006AFDBFFF
[mem 0xffe00000-0xffffffff] reserved   BS Data    000000006AFDC000-000000006B1BDFFF
                                       ACPI Recl  000000006B1BE000-000000006B1DCFFF
                                       BS Data    000000006B1DD000-000000006B26DFFF
                                       Available  000000006B26E000-000000006B2ABFFF
                                       BS Data    000000006B2AC000-000000006B2BCFFF
                                       Available  000000006B2BD000-000000006C27BFFF
                                       BS Data    000000006C27C000-000000006C2CEFFF
                                       Available  000000006C2CF000-000000006C2D2FFF
                                       BS Data    000000006C2D3000-000000006C431FFF
                                       LoaderCode 000000006C432000-000000006C50DFFF
                                       (Skip lots of BS Data/BS Code entries)
                                       RT Code    000000007A160000-000000007A22FFFF
                                       RT Data    000000007A230000-000000007A24FFFF
                                       Reserved   000000007A250000-000000007A74FFFF
                                       ACPI NVS   000000007A750000-000000007A767FFF
                                       Reserved   000000007A768000-000000007A768FFF
                                       ACPI NVS   000000007A769000-000000007A7B5FFF
                                       ACPI Recl  000000007A7B6000-000000007A7E2FFF
                                       BS Data    000000007A7E3000-000000007A7FEFFF
                                       Available  0000000100000000-00000001007FFFFF
```

Listing 1: On the left the memory map of a BIOS based system as seen by the Linux kernel. On the right the memory map of a UEFI system [53, p. 17].

are critical for a correct functioning kernel or for performance reasons have to remain in the address space at all times. When memory is allocated, the requester can indicate in which zone the memory must reside and if this is a hard requirement or that when there is not enough memory available in that zone, a different zone can be used.

Assuming an Intel x86 based system, three zones (see Figure 1) are used. Both 32-bit and 64-bit systems assign the first sixteen megabyte to `ZONE_DMA` for DMA transfers with devices that are not able to use the full 32-bit address space. On 32-bit systems the remaining memory is divided over `ZONE_NORMAL` and `ZONE_HIGHMEM`. Up to 896 megabytes is assigned to `ZONE_NORMAL`, when more is available this will be assigned to `ZONE_HIGHMEM`. The reason [55] for this limit is that normally the virtual address space for applications is split in one gigabyte for the kernel and three gigabytes for the application. However, PAE allows a 32-bit system to use more than four gigabytes of memory by temporary mapping those parts into the virtual address space. The ZONE_NORMAL zone is always mapped in the (virtual) kernel address-space (by adding 0xC0000000 [11] to the physical address), leaving 128 megabytes for mapping parts of ZONE_HIGHMEM.

On 64-bit systems the (virtual) address space is big enough to be able to map the whole memory without the need of a separation between normal and high memory. Therefore, ZONE_HIGHMEM is no longer used and everything above sixteen megabytes would be assigned to ZONE_NORMAL. However, a new problem appeared. Many devices that weren't limited to the first sixteen megabyte for their DMA transfers are limited to the memory in the first 32-bit of the address space. This resulted in the creation of the ZONE_DMA32 zone that consists of all the memory above the first sixteen megabyte in the lower 32-bit of the address space.

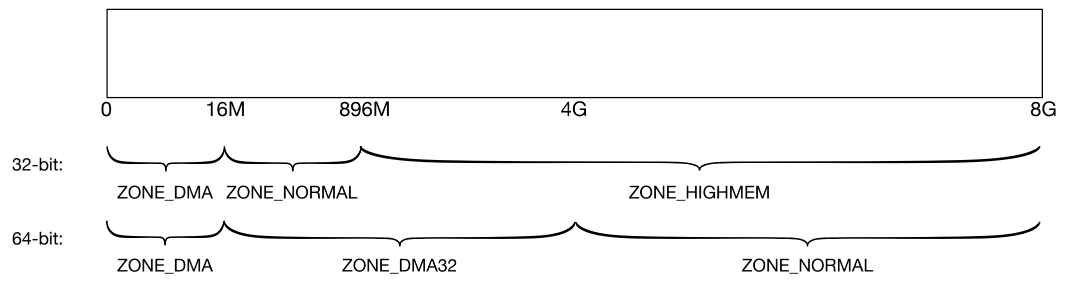


Figure 1: An impression of which parts of the RAM are assigned to which zone on 32-bit and 64-bit Intel systems.

Where specific data is allocated is hard to predict. Most allocations in the kernel are dynamic and position independent. In the default configuration, the kernel itself is loaded directly at the beginning of ZONE_NORMAL followed by most of the critical data structures. Userspace allocations and the page cache are evenly spread [49] over the zones, proportionally to their size.

2.3.3 Windows

The situation on Windows based systems is quite different. While the layout of the virtual address space is well understood [50] and documented [1, 2] and not that different from Linux, the way the allocation of pages is managed is barely documented.

The Windows kernel manages its memory by assigning memory pages to pools with a certain purpose. Two of the main pools [38], the paged and nonpaged pool, are used for memory that can be paged out and not paged out respectively. A typical example of data stored in the nonpaged pool are the data structures of the kernel and drivers. The registry is an example what is stored in the paged pool.

When RamMap is used to explore where the data resides that is used for certain purposes, no pattern could be identified. The pages belonging to a pool are spread all over the physical address space. The only exception appears to be the first megabyte of memory. This region is mostly used by drivers, but even there a few exceptions could be found.

The Page Frame Number database [39, p. 297] is used to keep track of the status of the physical pages. Each page is assigned a state like Active, Free or Zeroed and put in a corresponding list. When the system needs a free or zeroed page, depending on the security requirements of the allocation, the corresponding list is consulted. Because pages are tracked by their state instead of their location like on Linux based systems, the location of an object can be anywhere in the memory of the system.

3 Approach and Methods

The next step is to determine the approach of this research project. While some answers have already been found during the literature study, of which the results have been described in the previous section, most of them will require several experiments to gather the necessary data. The information learned will be used during the design of an algorithm to compress the content of the RAM of a computer system. The last step is to realise the algorithm in a proof of concept to validate it and compare it against the existing solutions.

3.1 RAM Profiles

As preparation for the selection of the compression algorithm to use and to get insight in the behaviour of operating systems and the way the RAM is managed, several profiles of potential target systems have been created. Each profile describes a different usage pattern of the system using an operating system that could be found during a forensic investigation. While many configurations are possible, a selection of twelve different has been made. Table 1 shows an overview of the selected scenarios.

The profiles can be divided in three groups. The first one is the group with office systems. Both Windows and Linux based systems are tested using different configurations such as versions, the amount of RAM and the usage of full disk encryption. The second group consists of two server scenarios. A fileserver and a webserver, both Linux based, are added. The fileserver will be filled half with text-based content using public domain e-books and half with pictures and other media. The webserver will use full disk encryption and is filled with pictures.

The last group consists of three special scenarios. Two of them consist of the usage of the Tails live operating system, which can be booted from a DVD or USB stick.

Operating System	Version	RAM	FDE[7]	Simulated usage
Windows XP	SP2 32-bit	512 MiB	✗	Desktop[8]
Windows 7	SP1 32-bit	2048 MiB	✗	Desktop
Windows 7	SP1 64-bit	2048 MiB	✗	Desktop
Windows 8.1	RTM 64-bit	4096 MiB	✓	Desktop
Windows 8.1	RTM 64-bit	8192 MiB	✗	Desktop[9]
Ubuntu	14.04 32-bit	2048 MiB	✗	Desktop
Ubuntu	14.04 64-bit	4096 MiB	✓	Desktop
Tails	1.4 64-bit	2048 MiB	✗	Browsing with Tor
Tails	1.4 64-bit	2048 MiB	✓	Encrypted volume (10 GiB) filled with pictures
Debian	7.8 32-bit	2048 MiB	✗	File server filled with text files and pictures
Debian	8.1 64-bit	4096 MiB	✓	Web server with pictures
OpenWRT	15.05-rc1 64-bit	256 MiB	✗	Embedded device e.g. NAS or STB

Table 1: The scenarios that are used to create a baseline for the content of RAM dumps.

[7] Full Disk Encryption

[8] Normal desktop usage e.g. browsing the web, playing music and video, etc.

[9] This is a dump of a virtual machine that was used during this project as normal workstation for a couple of hours.

It can be used to access the web anonymously using TOR, but also to encrypt the hard drive of the system. Tails promises to leave no persistent traces on a system during its usage and as such is also interesting to make the forensic analysis of a seized system harder. The last special scenario is the usage of an embedded operating system such as OpenWRT. Such systems can be found as NAS or setop box and have usually limited resources. While such systems are not the primary target for this project, their existence could have implications for the applicability of the project.

All the profiles are realised as virtual machine. This makes it easy to extract the whole content of the RAM and differences between the systems can in no way influence the content or its spreading. The desktop systems are used for web browsing, listening to music and viewing photos. To make sure that each system represents the usage for a longer period of time, for Windows a tool called RAMMap[10], part of the Sysinternals Suite, is used to determine if the RAM has been completely filled at least once. For Linux based systems the `free` command is used.

The dumps of the RAM content are then analysed. The most interesting property to measure is the Shannon entropy. This gives an indication of the information density (in bits of information per byte of data) and how well the data are compressible. This is used to determine if there are certain locations with an exceptionally low or high entropy and could be used to determine the starting point of the compression because in the worst-case scenario the compression of already compressed data could result in a bigger output than input. Since the only place to store the compressed output is on the location of the already processed input, this would result in overwriting the input data or having to drop some of the input until there is enough space again to store the compressed output.

A special case that is added to the desktop systems is a dump of the RAM content of a system that was used for a couple of hours during this project. This is to add the content of a real system to compare with the results from the simulated profiles. The reason for this is to have a sufficient level of certainty that the acquired information is valid for real-world scenarios.

3.2 Compression Algorithms

The next step is to select the compression algorithm to use. While several[11] benchmarks[12] exist[13] that compare compression algorithms, most of them focus on speed or compression ratio. While these are important factors to determine which algorithm to use, at least as important for this project is the information about the memory usage during the compression, the size of the output if the input is not compressible in the worst case scenario, and if the compression can be split in smaller blocks that are chained. However, when this is not possible, a solution would be to wrap the compressed blocks in a custom container format. The decompression characteristics of the algorithms are not tested.

A total of thirteen different general-purpose data compression algorithms (see Table 2) have been tested. Five of them are compression algorithms that are used as building block or inspiration for most widely used compression algorithms. A special case is the chaining of three (RLE, LZ77 and Huffman) of them as this would combine the strengths of each of them [16]. The other algorithms are widely used and selected based on

[10] https://technet.microsoft.com/en-us/library/ff700229.aspx
[11] http://compressionratings.com
[12] http://www.squeezechart.com
[13] http://mattmahoney.net/dc/text.html

Algorithm	Library	Used settings	Max. output (in byte)[14]	Window[15]	Part. comp.?[16]	Remarks
RLE	BCL 1.2.0	n/a	$\frac{257}{256} \cdot \texttt{originalSize + 1}$	n/a	After mod.[17]	The full input is read to create the histogram, which is used during the compression. Then the input is processed byte for byte.
Huffman	”	n/a	`originalSize + 320`	n/a	”	The full input is read to create the histogram and Huffman tree, which are used during the compression. Then the input is processed byte for byte.
Rice	”	Unsigned 8-bit words	`originalSize + 1`	n/a	”	When the compression fails, the output buffer is cleared and a null byte is written to the begin of it. Then the input buffer is copied to the output buffer.
Rice	”	Unsigned 16-bit words	`originalSize + 1`	n/a	”	”
Rice	”	Unsigned 32-bit words	`originalSize + 1`	n/a	”	”
LZ77	”	n/a	$\frac{257}{256} \cdot \texttt{originalSize + 1}$	10000 b	”	The full input is read to determine the least commonly used byte to use as marker during the compression.
Shannon-Fano	”	n/a	$\frac{257}{256} \cdot \frac{257}{256} \cdot \texttt{originalSize + 386}$	n/a	”	The full input is read to create the histogram, which is used during the compression. Then the input is processed byte for byte.
RLE - LZ77 - Huffman	”	n/a	`originalSize + 320`	10000 b	”	
BWT	libbzip2 1.0.6	Block size 100 KiB	$\texttt{originalSize} \cdot \texttt{1.05 + 50}$ [43]	100 KiB	✓	
DEFLATE	zlib 1.2.8	Default	$\texttt{originalSize + 5} \cdot (\frac{\texttt{originalSize}}{2^{15}} \texttt{ + 1)}$ [14]	32 KiB	✓	
LZ4	lz4 r130	Default	$\texttt{originalSize + } \frac{\texttt{originalSize}}{\texttt{255}} \texttt{ + 16}$	64 KiB	✗	Maximum output length based on the `LZ4_COMPRESSBOUND` macro. The maximum input size is two gigabytes.
LZF	liblzf 3.6	Default	Up to 104% of original size	4 KiB	✓	
LZMA	xz 5.2.1	Default	$\texttt{1.001} \cdot \texttt{originalSize + 1024}$ [17]	2 MiB	✓	
LZO	lzo 2.0.9	LZO1 preset	Up to 106% of original size	8 KiB	✗	
LZO	”	LZO1X-1 preset	Up to 106% of original size	2 KiB	✗	
LZO (mini)	minilzo 2.0.9	Default	Up to 106% of original size	16 KiB	✗	
LZW	Unknown[18]	9-bit table	Unknown	n/a	✗	The input is processed byte for byte using a dynamic dictionary table.
LZW	”	12-bit table	Unknown	n/a	✗	”
LZW	”	15-bit table	Unknown	n/a	✗	”

Table 2: An overview of the tested algorithms and their theoretical characteristics.

[14] Based on the documention or code. Other sources are cited.
[15] The amount (input) data that is considered during the compression, for example, as a sliding window based dictionary.
[16] Partial Compression: Is it possible to compress the data in (small) blocks?
[17] After modification: Not by default in the used implementation, but it is easy to implement.
[18] http://rosettacode.org/wiki/LZW_compression#C

their inclusion in operating systems or usage in file formats. Several algorithms have been tested with different settings that should influence the compression ratio or the memory usage during the compression. Of the tested algorithms, three of them are used in multiple configurations. This results in a total of nineteen different tests.

The data to conduct the tests with are the RAM dumps gathered during the previous step. A script is written to automate the testing and to exclude the human factor. Each algorithm is tested using a minimal application written in C to set up the environment and run the algorithm. The applications are compiled in release mode (stripped binaries, maximal optimisation and no debug symbols). The `gettimeofday` API is used to determine how long it takes to compress the data. This information is recorded with millisecond precision. The amount of memory required for the compression is determined using Valgrind with the Massif profiler. Because the usage of Valgrind imposes a significant overhead, this is done as a separate test instead of combined with measuring the compression duration. For the code size the `size` utility is used. The theoretical maximum output length for the worst case scenario and the ability to split the compression up in smaller blocks is based on the documentation of the used library, a (manual) review of the library source code and the corresponding theoretical analysis of the compression algorithm.

The tests are conducted on a 64-bit system with a Linux based operating system as a Linux based system is required by the Valgrind tools and there is no easy substitute available to determine the amount of used heap and stack memory. The proof of concept will be executed before an operating system is loaded and only a very limited runtime will be available. This will influence (to some level) the measured compression duration and memory usage. However, for this project the relative differences are used which should not be influenced.

Another difference from the proof of concept is that the RAM dumps have to be loaded from disk. Compared to the execution of the compression algorithm the overhead should be limited, because in the proof of concept the data to compress is the content of the RAM instead of a file on disk. For this reason, the time measurement is implemented in the application itself and started after the full image is loaded in memory. The input and output buffers are allocated on the heap and subtracted from the reported usage as in the proof of concept these buffers are not used.

3.3 Acquiring Algorithm

While the compression of the RAM content of a system appears quite easy at first sight, doing it with code stored in and running from the same RAM is quite complicated. This results in the need for a more advanced acquiring algorithm than just starting at address zero, that takes care of the practical implications of such situation.

During the design phase of the algorithm, several design goals have been used as guideline. Those goals are that as much as possible of the RAM should be acquired, in a reasonable time frame and independent from the memory layout of the system. As long there is enough space to boot the forensic operating system to extract the compressed data, achieving the highest possible compression is not the goal.

While the goal to acquire the data in a reasonable time frame is very vague, the time taken by the acquiring algorithm is (almost) fully dependent on the chosen data compression algorithm. Therefore, the designed algorithm should be as generic as possible and fully independent from the used algorithm to compress the data. When selecting the data compression algorithm for the proof of concept, only the algorithms that are able to compress four gigabytes of data in less than an hour will be considered, even when a slower algorithm is able to compress better or overwrites less data.

For legacy reasons, the memory layout of a system is not continuous. The available RAM is split in multiple regions. Between some of those regions address ranges are reserved for devices that have to be mapped in the address space. Also the firmware of the system will reserve space for its own use, for the system configuration data or for other reasons (such as earlier detected memory errors) to mark a part of the memory as not usable.

Several memory locations are also safe to read, and the possibility exists they contain interesting information, but it is not possible to overwrite them. For example, because it is assigned to the system firmware or contains the code or data of the application that executes the acquiring algorithm. Such locations should be stored by the algorithm on a different location, where space is created by compressing that data first.

An inevitable consequence of compressing already compressed or encrypted data is that with most compression algorithms the resulting output is bigger than the input. As the output of the compression has to be written on the same location as the input originates from, this results in a conflict. If the situation is not handled, the still-to-be-processed input is overwritten resulting in the never ending situation of recompressing the last output from before the situation started. This would destroy all data after the moment this occurred and is only stopped by reaching the end of the memory region.

If this situation occurs, only two solutions are possible to prevent damaging the not processed data further. The first solution would be to stop compressing this region and finish the other regions. The operating system where the application runs that extracts the compressed data should be informed about where the compressed data ends and what part should be acquired by reading the raw data. However, if this occurs early on in the compression stage, it is very likely there will not be enough space to boot the operating system and an unsolvable situation is created.

A better solution is to drop a small amount of input data and replace it with null bytes, just enough so that there is space again to continue with the compression. While this results in a loss of data, compared to the situation when all data is lost because booting the operating system fails, this has to be accepted.

To reduce the chance such situation occurs, the algorithm should determine its starting point by measuring the Shannon entropy. By searching for a location with a low entropy measured over a couple of hundreds of kilobytes, in most situations the loss of data will be prevented as later on there will be enough space created by compressing low entropy data to handle the non-compressible regions.

A serious risk is the corruption of the compressed data while it is in RAM. This could happen because a (small) part is overwritten by the booting operating system or firmware itself or (in the very unlikely event of) faults in one of the RAM modules that result in flipped bits. While overwriting the data should be prevented by marking the locations where the compressed data blocks reside, it is better to take it in account. An easy solution is to split the compressed data in smaller blocks. Then is in the worst-case only a small amount of data lost.

Another issue is that many operating systems, such as the Linux kernel in the default configuration, assume they are loaded and started from a predetermined, fixed address. The usage of those locations to store the compressed data should be prevented where possible, for example by moving the compressed parts to a different location. By spreading the data over multiple blocks, it is easier to move them around.

3.4 Proof of Concept

To realise and test the effectiveness of the algorithm, a proof of concept is developed. This proof of concept is a minimal application that is executed before the real operating system. This application implements the previously described algorithm and compresses the RAM content. Then it loads the kernel of the operating system that runs an application to extract the compressed data. The operating system that is booted after the compression stage is a modified Linux distribution.

The goal of the algorithm is to acquire as much as possible of the RAM content while the data should not be overwritten before it is compressed. However, it is inevitable that a (small) part will be overwritten during the loading of the application that implements the algorithm. For this reason, the size of the application should be as small as possible and be executed as early as possible in the boot procedure.

Because this project fits in the bigger picture of projects to make remote acquisition possible, it is desirable the proof of concept works not only when it is started from CD or UBS stick, but also when it is booted over the network using PXE. And as more and more systems use UEFI nowadays, both BIOS and UEFI should be supported.

The development for each of the before mentioned environments is different. For this reason a bootloader is used to provide a generic runtime to implement the application on top of. The bootloader used for this project is Syslinux[19]. Syslinux provides specialised builds for booting from hard drives, CDs and DVDs, and over the network using PXE for both BIOS and UEFI based systems.

Applications for Syslinux are developed as loadable modules. Several modules have been developed for this project and one of them, the `memcompress` module is used as proof of concept of the algorithm. After executing the algorithm, it loads a 64-bit OpenWRT[20] based operating system. This system runs several scripts to mount a NFS volume on a remote server and executes a Python script that extracts the RAM content and saves it to the NFS volume.

OpenWRT is chosen because it is a Linux distribution designed for usage on devices with limited resources. It is easy to customize and has extensive support for many platforms. In the upcoming release support for 64-bit Intel systems is added. As this is required to have access to the full memory when a system has more than three gigabytes of RAM, the first release candidate that was released just before the start of this project is used.

To protect the compressed data from the booting operating system, the memory map as is provided by the firmware of the system is manipulated in such a way that to the operating system those parts appear to be not usable. By using a modified kernel command line, the extraction script is instructed which parts of the memory have to be stored.

The script that extracts the compressed blocks, accesses the RAM using the special `/dev/mem` device node that allows a userland application with sufficient access rights to read and write directly to the RAM using the physical addresses. As last step, the compressed data is decompressed and an overview is created with which data is recovered, what is missing and the integrity of the recovered data.

Normally, the Linux kernel prevents access to memory addresses that (according to the kernel) do not exist. Besides this, as part of an effort to improve the security of the

[19] Syslinux 6.03
[20] OpenWRT version 15.05-rc1

system, by default the access is restricted [5, 27] to a small subset of the memory such as the first megabyte of RAM for legacy reasons and the PCI mmio resources. Because the extraction scripts must have access to the full memory, including the regions with the compressed data that are shielded from the kernel, both protections are removed from the kernel.

3.4.1 Data Format

While not strictly necessary, the proof of concept stores a header before every block of compressed data. This block contains a marker, the compressed and uncompressed lengths, the originating address range, how much data is lost and a checksum generated from the original data. Listing 2 shows the struct definition of the header that is prepended.

The marker is added so that by simply carving the full RAM, the compressed blocks can be found back. While this is normally not needed as the `memcompress` module reports to the extraction script the locations using the kernel command line, the situation could occur that the extraction fails and another method is used to dump the RAM. As the RAM now contains parts of the loaded operating system together with the compressed blocks on unknown locations, this 8-byte marker can then be used to still recover the parts that are still intact. When a data block is moved, the original block is not removed but the marker is modified by overwriting the second byte and change “`E`” to “`3`”. This way if the moved block is lost but the original block is preserved a part of the data can still be recovered without the risk that both blocks are extracted. For 010editor[21], a binary editor of SweetScape, a carving template is developed to extract the headers from a raw memory dump.

Because it is not unlikely that compressed blocks are stored in a different region then the data originates from, the start and end addresses, and the compressed and uncompressed lengths are stored. This way it is always possible to know the exact location of the original data and create a map of which regions are extracted and which parts are missing.

Before it is compressed, the input data is hashed using the SHA-256 algorithm, part of the SHA-2 family, that acts as checksum during the decompression. This way the forensic integrity can be maintained and be assured that the decompressed data is equal to the original input data.

```
struct {
    unsigned char marker[8] = "MEM\xf1\x88\x15\x08\x5c";
    uint64_t start_address;
    uint64_t end_address;
    uint64_t compressed_length;
    uint64_t uncompressed_length;
    uint64_t skipped_length;
    unsigned char checksum[32];
};
```

Listing 2: The definition of the header that is prepended before every compressed block of data as struct definition.

[21] http://www.sweetscape.com/010editor

3.4.2 Effectiveness

The last step of this project is to determine the effectiveness of the proof of concept. To test this, two criteria will be assessed. The first thing to check is how much of the content of the RAM of a system can be recovered. This will be compared with the existing methods. To do this, the memory of a virtual machine will be (manually) filled with a known pattern. After extraction a script is used to determine how much of it can be recovered. A virtual machine is used as it is very hard to fill the whole RAM of a system and also to know for sure it is actually completely filled.

To test the second criterion, the proof of concept is tested on how much RAM is required to boot the operating system to extract the compressed data. With this information, an attempt is made to give guidelines for which systems this is a viable method while for others it remains better to use the traditional methods.

While no conclusive data are found regarding the impact of the system firmware on the amount of data that cannot be recovered from the RAM, further research regarding this is left outside the scope of this project.

4 Results

Here the results will be presented from the experiments described in the previous chapter. The same order will be used, first the results of the memory scenarios and data compression algorithm selection, then the results of the developed algorithm and proof of concept.

Appendix C has an overview of the developed software components and where they can be found.

4.1 RAM profiles

A total of twelve memory dumps have been examined as part of this project, with the focus on the information density of the content. This is measured for the whole dump and in blocks of four and sixteen kilobytes as this is relevant information regarding compression algorithms that use block based compression. Table 3 shows the results for each dump.

While this gives interesting information for the dump as a whole, a more detailed analysis is possible by visualising the data. Appendix A contains two different methods to visualize the information density. The first method uses a graph where the entropy is calculated by dividing the dump in 512 equal sized blocks which each representing a data point. The second group of graphs show the entropy with much more detail. Colours are used to indicate the information density and every pixel corresponds to a small block (between 2048 and 65536 bytes per block, depending on the size of the dump) in the image.

4.2 Compression Algorithms

The same memory dumps are used to gather the information regarding the data compression algorithms. Table 4 shows a summary consisting of the means for every algorithm of the time it took to compress one gigabyte of data, the percentage of the gained space by the compression and the memory usage during the compression. The results of the memory usage are split out in the size of the application code, the usage of the stack and the allocations on the heap. Appendix B contains the information for each individual dump that was used for this summary.

Based on this information RLE, LZF and Rice are the most promising candidates to use in the proof of concept. LZF is selected for the proof of concept because it is the fastest of the three algorithms and because it has a much smaller compression size than RLE. Rice is not selected because the algorithm has a major drawback. While it looks great on first sight, it only works under certain conditions and otherwise it crashes or the output size explodes to many times the size of the original input. If this happens, the algorithm aborts, clears the output buffers and copies the input buffer to the output buffer with a null byte prepended to indicate it is not compressed. However, in the context of this project this is not possible as the original input is no longer there and there would be no space to boot an operating system. In theory it is possible to revert the compression so a different algorithm can be used, but this is not implemented and has not been tested.

Huffman, Shannon-Fano, LZ4 and LZO still provide decent results, but do not perform as well as LZF and have a higher memory usage and a comparable or significantly

	Entropy	Entropy (4k blocks)						Entropy (16k blocks)					
Scenario	(full dump)	Min	Max	Mean	σ	<1.0	>7.0	Min	Max	Mean	σ	<1.0	>7.0
Windows XP 512 MiB (Desktop)	4.83	0.00	8.00	3.37	2.05	37.1%	5.1%	0.00	7.99	4.16	1.26	5.9%	0.3%
Windows 7 x86 2 GiB (Desktop)	5.03	0.00	8.00	3.63	2.65	208.5%	74.6%	0.00	7.99	4.39	1.59	24.3%	18.2%
Windows 7 x64 2 GiB (Desktop)	3.32	0.00	8.00	2.63	1.68	213.6%	0.7%	0.00	7.99	2.98	0.93	14.4%	0.2%
Windows 8.1 x64 4 GiB (Desktop w. FDE)	4.95	0.00	8.00	3.65	2.12	247.5%	93.9%	0.00	7.99	4.21	1.53	52.4%	58.8%
Windows 8.1 x64 8 GiB (Desktop, after couple hours of normal usage)	5.82	0.00	8.00	4.48	2.54	478.2%	666.3%	0.00	7.99	5.17	1.64	52.6%	403.9%
Ubuntu 14.04 x86 2 GiB	5.20	0.00	8.00	3.68	2.42	172.9%	71.3%	0.00	7.99	4.10	2.23	121.6%	56.2%
Ubuntu 14.04 x64 4 GiB (Desktop w. FDE)	4.83	0.00	8.00	3.37	2.17	279.9%	90.2%	0.00	8.00	3.65	1.99	190.0%	68.2%
Tails 1.4 2 GiB (Browsing w. Tor)	6.24	0.00	8.00	4.88	2.37	59.6%	212.2%	0.00	8.00	5.18	2.11	37.2%	195.4%
Tails 1.4 2 GiB (Encrypted folder w. images)	7.39	0.00	8.00	6.59	2.17	25.0%	542.7%	0.00	8.00	6.74	1.94	12.5%	536.5%
Debian x86 7.8 2 GiB (File server w. mixed content)	7.51	0.00	8.00	6.56	2.27	47.5%	543.6%	0.00	8.00	6.69	2.12	37.3%	542.7%
Debian 8.1 x64 4 GiB (Web server w. images)	7.86	0.00	7.97	7.51	1.42	22.0%	1470.2%	0.00	7.99	7.57	1.32	15.6%	1466.4%
OpenWRT 256 MiB	3.34	0.00	7.98	1.71	2.77	68.1%	10.0%	0.00	7.99	1.86	2.75	62.0%	10.1%

Table 3: The information density (Shannon entropy) of the memory dumps measured in bits per byte of data. The entropy is measured over the whole image and in blocks of four and sixteen kilobytes.

								Memory usage										
	Failures[22]	Duration (1G)			Gain			Code	Stack			Heap			Total			
Algorithm		Min	Mean	Max	Min	Mean	Max		Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
RLE	-	2.4s	6.3s	23.3s	3.1%	28.4%	73.3%	4.3 K	3.2 K	3.2 K	3.2 K	0	0	0	7.5 K	7.5 K	7.5 K
Huffman	1 / 12	11.6s	17.9s	25.8s	5.6%	39.5%	70.9%	6.4 K	16.8 K	16.8 K	16.9 K	0	0	0	23.2 K	23.2 K	23.3 K
Rice8	10 / 12	52.9s	57.0s	61.1s	16.0%	25.1%	34.3%	6.6 K	424	424	424	0	0	0	7.0 K	7.0 K	7.0 K
Rice16	10 / 12	32.8s	36.3s	39.8s	19.9%	29.2%	38.5%	6.6 K	424	424	424	0	0	0	7.0 K	7.0 K	7.0 K
Rice32	11 / 12	59.6s	59.6s	59.6s	-0.0%	-0.0%	-0.0%	6.6 K	424	424	424	0	0	0	7.0 K	7.0 K	7.0 K
LZ77	12 / 12	n/a	n/a	n/a	n/a	n/a	n/a	6.2 K	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Shannon-Fano	-	13.2s	20.6s	31.6s	1.1%	28.6%	57.5%	5.8 K	4.7 K	4.9 K	4.9 K	0	0	0	10.6 K	10.7 K	10.8 K
RLE-LZ77-Huffman	12 / 12	n/a	n/a	n/a	n/a	n/a	n/a	10.6 K	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
BWT	-	43.5s	131.0s	228.6s	6.8%	57.7%	92.1%	76.3 K	5.3 K	5.3 K	5.4 K	1.1 M	1.1 M	1.1 M	1.1 M	1.1 M	1.1 M
DEFLATE	-	16.7s	29.3s	36.8s	6.9%	58.3%	91.5%	49.1 K	672	672	672	261.8 K	261.8 K	261.8 K	311.6 K	311.6 K	311.6 K
LZ4	-	0.6s	1.4s	2.0s	4.6%	51.9%	87.9%	52.6 K	16.4 K	16.6 K	17.1 K	0	0	0	69.0 K	69.2 K	69.6 K
LZF	-	2.0s	4.1s	6.5s	3.2%	50.8%	84.7%	4.5 K	3.2 K	3.2 K	3.2 K	0	0	0	7.7 K	7.7 K	7.7 K
LZMA2	-	157.5s	334.3s	452.7s	7.6%	63.6%	94.8%	90.6 K	1.6 K	1.6 K	1.6 K	93.1 M	93.1 M	93.1 M	93.2 M	93.2 M	93.2 M
LZO1	-	2.0s	5.5s	10.3s	5.6%	50.8%	83.6%	5.5 K	376	396	416	64.0 K	64.0 K	64.0 K	69.9 K	69.9 K	69.9 K
LZO1X-1	-	0.6s	1.4s	2.1s	5.5%	51.2%	86.6%	5.2 K	408	408	408	16.0 K	16.0 K	16.0 K	21.6 K	21.6 K	21.6 K
LZOmini	-	0.6s	2.1s	9.7s	5.5%	51.7%	86.9%	13.3 K	432	432	432	16.0 K	16.0 K	16.0 K	29.7 K	29.7 K	29.7 K
LZW9	1 / 12	13.8s	24.7s	36.3s	8.6%	45.3%	76.5%	7.6 K	464	464	464	128.3 M	2.5 G	4.0 G	128.3 M	2.5 G	4.0 G
LZW12	3 / 12	32.1s	63.5s	108.9s	-4.7%	43.0%	76.5%	7.6 K	464	464	464	130.0 M	1.6 G	4.0 G	130.0 M	1.6 G	4.0 G
LZW15	3 / 12	37.9s	90.8s	156.8s	-8.9%	41.0%	82.2%	7.6 K	464	464	464	144.0 M	2.0 G	4.0 G	144.0 M	2.0 G	4.0 G

Table 4: The (averaged) results of the compression tests for each tested algorithm. The duration is normalised to how long it took to compress one gigabyte. The gain gives the percentage of how much the compression reduced the original size.

[22] LZ77 (and also RLE-LZ77-Huffman) was removed because not a single test was finished within 3 hours. Huffman, Rice and LZW have unknown issues that result in the corruption of the data with certain memory dumps. As most tests failed, Rice, LZ77 and RLE-LZ77-Huffman are removed from further testing.

bigger compressed size. Only LZ4 compresses better, but at the cost of almost ten times the memory usage of LZF.

Most of the tested (implementations of the) algorithms are not able to compress more than four gigabytes of data in a single run. For this reason only the first four out of eight gigabytes are used dump. Several of the tested algorithms (Huffman, LZ4, LZW, Rice, Shannon-Fano) had even a slightly lower maximum input size. When an algorithm wasn't able to compress the full four gigabytes, those images were slightly reduced in size (ten bytes) to make the compression possible.

4.3 Acquiring Algorithm

Based on the goals and operational conditions mentioned in the approach, a generic acquiring algorithm has been developed that will be described now. The algorithm in pseudo code can be found in Listing 2.

First, the memory map is retrieved from the system firmware. The region entries are sorted based on their size, with the biggest first. The reason for this is that for the first region a place must be found where the compressed output can be stored. While on average this should not be an issue, if by accident the compression is started with a small memory region with a (very) high entropy the compressed output could be larger than the region itself.

Each of the regions is then compressed. Before the compression can start, the location where the output can be stored must be determined. If there is space left in a previous region that is already compressed, this is used. However, if this is not the case, for example because this is the first region to compress, this region is checked if it is reserved or contains (parts of) the application.

If the region is reserved (i.e. the region is readable but can't be overwritten) or is within the first megabyte of address space, the entry is put back to the end of the list of regions that still have to be processed. If a region contains application code, a new region for this part is created, marked as reserved and also added to the list. The idea of this is that such regions are compressed to a different, already compressed region, to prevent the overwriting of critical data like the compression module or interrupt vectors.

Because there is no space in an earlier compressed region to store the compressed output of this region, there is no option left then to write the compressed data back to the location of the input that has already been processed. The first thing to determine is whether this region starts with data with a (very) high entropy. If this is the case, this part is skipped and a separate region is created for it.

Which part of the memory to compress and where to store the compressed data has now been determined. The data of the region to compress is divided in multiple blocks. Each block is independently compressed and stored and contains up to one hundred megabytes of input data. This way when the data is stored in a different region and this region has no space left anymore, it can continue on a different location and if for some reason a part of the data gets overwritten, not the full memory dump will get lost. The limit of one hundred megabytes of data as input is arbitrarily chosen. However if the blocks are too small the overhead (80 bytes per block) of the header and compression algorithm starts to matter and if the blocks are too big more data than needed is lost when a block cannot be recovered. When there is some practical experience with this system, this parameter should be fine-tuned further.

During the compression of a block the algorithm carefully monitors the current output location. If the output is written to the same region as where the input is read from, the algorithm will skip small parts (in the current implementation one hundred bytes per occurrence) of the input and replace them with null bytes[23] if continuing results in writing output past the current read pointer. If the input or output pointer reaches the end of the region or the maximum amount of data for a single block has been reached, the block is finished by writing the header to the beginning of the block.

When all regions are compressed the remaining available space is determined. As the minimal amount of available memory to boot the operating system is known, the system has to decide what to do when this is not available. In the current design of the algorithm "random"[24] blocks are marked as available until the minimal amount is reached. Further research is required to determine an optimal strategy, such as assigning a score to the compressed blocks based on location, origin and entropy, to reduce the chance critical information will be overwritten.

Then all data blocks are moved to the end of the region in which they reside, which is usually the biggest "usable" region in the 32-bit address space. This is the easiest solution to make space for the Linux kernel that must be loaded on a fixed address in the default configuration. As the total amount of required memory for the kernel loader is much less than the space that is required to fully boot the operating system, this should guarantee a successful boot. Overwriting critical locations (e.g. application code) should be prevented by forcing the compression of it to a different region as is done during one of the earlier steps.

```
memregions ← GetMemoryMapFromFirmware()

SortByRegionSize(memregions)
ReverseOrder(memregions)              /* start with the biggest region */

foreach memory region m in memregions do
    if IsReadableRegion(m) then
        CompressRegion(m)   /* resulting in 1 or more compressed blocks */
    end
end

if not HasEnoughSpaceToBootOperatingSystem() then
    repeat
        MarkRandomCompressedDataBlockAsAvailable()
    until HasEnoughSpaceToBootOperatingSystem()
end

foreach compressed block b do          /* without the blocks of prev step */
    MoveCompressedBlockToEndOfRegion(b)
end

foreach compressed block b do
    MarkCompressedBlockInFirmwareMemoryMapAsReserved(b)
end

BootOperatingSystem(memregions)
```

Listing 2: A high-level overview of the developed algorithm. The implementation of the most important functions can be found on the next page.

[23] If it is important that the analyst can recognize that data is skipped on a certain location, a repeating known pattern (like "SKIPSKIPSKIP" etc.) could be used instead of null bytes.

[24] Truly random is almost non-existing in computer systems. A PRNG could be used, seeded by reading (a small part of) the data of a region.

```
function CompressRegion(memory region m)
    if not HasSpaceInAlreadyCompressedRegion() then
        if IsReservedRegion(m) then                /* retry this region later */
            AddRegionToList(m)
            return
        else if RegionContainsApplicationCode(m) then
            mNew, m ← SplitRegionAfterApplicationCode(m)
            AddRegionToList(mNew)
        end
        if StartsWithHighEntropy(m) then
            mNew, m ← SplitRegionAfterHighEntropy(m)
            AddRegionToList(mNew)
        end
        // write output to the location of the processed input
        outputAddress ← StartAddress(m)
    else
        outputAddress ← AddressOfFirstAvailableLocation()
    end
    sourceAddress ← StartAddress(m)
    repeat
        sourceAddress, outputAddress ← CompressBlock(sourceAddress, outputAddress)
        if IsEndOfRegion(outputAddress) then
            // started output in other region but no space is left
            // continue output from the start of this region
            outputAddress ← StartAddress(m)
        end
    until IsEndOfRegion(sourceAddress)
end

function CompressBlock(startAddress, outputAddress)
    outputStartAddress ← outputAddress
    p ← sizeHeader
    repeat
        if startAddress + p < outputAddress or IsDifferentRegion(outputAddress)
        then
            outputAddress ← Compress(ReadByte(startAddress + p), outputAddress)
            p ++
        else                                        /* output reached input */
            // compress null byte and skip input to prevent overwriting it
            outputAddress ← Compress(0x00, outputAddress)
            p ++
        end
    until p equals to the amount of bytes to compress per block or reached end of region
    WriteHeader(outputStartAddress)
    return startAddress + p, outputAddress
end
```

```
BOOT_IMAGE=/openwrt/openwrt-ramfs.bzImage memmap=42156K$3102485K
   ↪ memmap=1048576K$4194304K m3mcomp=43165831@3176945529 m3mraw=1073741824@4294967296
```

Listing 4: An example of how the compression module instructs the kernel how to handle the acquired data. The `memmap` parameter is used to hide certain parts of the address space. Two custom parameters, `m3mcomp` and `m3mraw`, are used to instruct the script where the compressed blocks and uncompressed data respectively can be found.

The last step is to hide the compressed data for the operating system. By adding the location and length of the compressed data blocks as reserved space to the memory map of the system, the Linux kernel will not use these parts and the memory content will survive the start of the operating system. The acquisition script can now extract the compressed blocks and store them to the preferred location on a attached hard drive or to a mounted network volume. Listing 4 shows an example of a modified kernel command line as is generated by the proof of concept.

4.4 Proof of Concept

The proof of concept, called memcompress, has been successfully implemented as Syslinux module, combined with a OpenWRT based operating system that is booted after the compression phase and a Python script to extract the memory content.

Time limitations prevented the implementation of several of the proposed error handling mechanisms of the algorithm. This will not impact the testing of the proof of concept as they are used to handle failure conditions. The not implemented features are the searching for a low entropy zone to start compressing instead of starting at the beginning of the region (or a fixed offset in the case when the region starts at 0x100000 as this is the location of the Syslinux code) and the dropping of compressed regions to make enough space available for the operating system to boot.

Testing revealed that the used operating system, OpenWRT 15.05-rc1 with 64-bit kernel, requires 82 megabytes of available memory as absolute minimum. Less than this amount results in a kernel panic during the booting of the system. The reason for this is most likely that the kernel tries to relocate itself as part of the boot procedure. At a certain moment during the relocation it needs space for both the old and the new location which is not available.

When the minimal amount of memory is available, the system boots successfully. At the end of the initialisation of the kernel almost 30 megabytes of memory is released that is no longer needed. This is the space that is available to run the scripts to acquire the gathered data.

It is very likely that the memory usage of the kernel can be reduced even further. For the proof of concept the default kernel configuration (besides the patches to remove the restrictions from the `/dev/mem` device) of the OpenWRT project was used. Because the kernel imposes the current minimum, removing the hardware support for the more exotic systems and disabling the kernel features that are not used would lower it.

Because in the worst-case there is only 10-20 megabytes available for the script that extracts the data, the decompression and validation steps are separated. This way these steps can be executed from a different system. The decompression script will also produce a memory map as can be seen in Listing 5. This memory map gives an overview of what was successfully recovered and which parts are lost. The system is not able to

```
[              0] - [          9F7FF] OK
[          9F800] - [          FFFFF] MISSING
[         100000] - [         FFFFFF] OK
[        1000000] - [        73FFFFF] OK
[        7400000] - [        D7FFFFF] OK
[        D800000] - [       13BFFFFF] OK
[       13C00000] - [       19FFFFFF] OK
[       1A000000] - [       1FEEFFFF] OK
[       1FEF0000] - [       1FEFEFFF] OK
[       1FEFF000] - [       1FEFFFFF] OK
[       1FF00000] - [       1FFFFFFF] OK
```

Listing 5: An overview of a reconstructed memory map (of a virtual machine with 512 megabytes of RAM), which was generated during the extraction of the compressed data, based on the information in the headers of the compressed blocks. It is expected that the region between 0x9F800 and 0xFFFFF is missing because this range is used for the EBDA[25], video memory and ROM area on BIOS based systems. When the decompressed data don't match the checksum that was generated during the compression, the region will be marked with "Checksum INVALID!" instead of "OK".

detect what is lost as a result of loading the proof of concept itself, only what is lost during or after the compression of the data which under normal circumstances should be nothing.

4.4.1 Comparison with Existing Solutions

To assess the performance of the proof of concept, it is compared against msramdmp, bios_memimage and using OpenWRT with the extraction script, but without shielding the memory from the kernel. This should cover most existing solutions that can be used to retrieve the memory content of a system that must be rebooted to a different operating system before it can be retrieved. The tests were conducted with a virtual machine with 1024 megabytes of RAM filled with a known pattern. Table 5 shows the results of the different tests, consisting of the amount of data that could be recovered and the memory ranges that were not recoverable.

The initial idea was to use a generic virtualisation solution, for example VMWare Fusion or VirtualBox, fill the memory with a known pattern and execute the test. However, this was not successful. The memory was filled by pausing the virtual machine and directly modifying the memory content that is stored to disk when a virtual machine is paused. The moment the virtual machine was resumed, it crashed and the content was cleared during the reboot.

Then, an attempt was made to create a Syslinux module that fills the RAM. This is a very tricky approach as it is very easy to also overwrite the code of the module or one of its dependencies such as a libc function. This was solved by carefully writing an independent subroutine that could function as long as a region of around two hundred bytes around the location of the code was preserved. However, the moment the first megabyte of the RAM was overwritten, the virtual machine crashed.

This was traced back to the interrupt vectors that are installed by the firmware and reside in the first megabyte of RAM. The crashes were solved by disabling interrupts using the `cli` instruction. The consequence of this was that it was no longer possible

[25]Extended BIOS Data Area

Method	Recovered		Not recoverable	
msramdmp	1022.8 M	99.883%	1.2 M	0.117%
bios_memimage	1022.7 M	99.872%	1.3 M	0.128%
Proof of Concept	1019.5 M	99.556%	4.5 M	0.444%
Unmodified OpenWRT x64	878.0 M	85.746%	146.0 M	14.254%

Table 5: An overview of how much of the data the existing solutions and the new developed proof of concept are able to retrieve.

to reboot the virtual machine and execute the test. The bare minimum that had to be preserved to still be able to reboot the virtual machine was a total of around 350 kilobytes.

A solution was found in modifying the QEMU[26] system emulator. The subsystem that allocates the virtual machine memory was modified in such a way that instead of overwriting the memory with null bytes, it writes the required pattern. This way the whole memory of the virtual machine is filled the moment the firmware is loaded and the execution starts.

At first sight, it appears the proof of concept requires significant more memory than, for example, msrampdmp, that is also developed as Syslinux module. However, a major part of this additional memory usage can be attributed to the newer Syslinux version. Msramdmp was developed as Syslinux 4 module while the proof of concept was developed using the latest version, Syslinux 6. The major difference is that the latest version uses almost 2.7 megabytes on a location which was not used at all in Syslinux 4.

To find out what the components are that cause this significant increase in memory usage, the memory layout of the binary as is generated by the linker during the building of Syslinux is examined. There are two parts of Syslinux that contribute significantly to this increased memory usage. First, the way modules are implemented has been rewritten from DOS compatible COM executables (and an improved version of this called COM32 for 32-bit executables) to an ELF based system that includes support for shared dependencies and many other improvements. This new loader is much more complex and its implementation is bigger. However, the biggest increase is caused by `__file_info`. This is a globally allocated array of `file_info` structs. It contains the state of the file descriptors which are used to implement file support in Syslinux. Each file descriptor has its own sixteen kilobytes buffer, resulting in a memory usage of two megabytes because there is a maximum of 128 file descriptors. This array is part of the global data and is actively cleared during the initialization of Syslinux, even when not all file descriptors are used, which is usually the case. By lowering the maximum number of file descriptors or dynamically allocating the corresponding struct when it is needed, the memory usage could be reduced to 2.5 megabytes. Further reduction would require the removal of unused functionality. However, any change to the code of Syslinux results in a modified version which must be maintained.

4.4.2 Limitations

The proof of concept in its current state has several limitations. One of the limitations is that because Syslinux is used, the code runs in 32-bit protected mode with linear addressing. As such, it is not able to reach the memory with an address of 0x100000000 or higher. In practical terms this means that only the lower three gigabytes of RAM

[26] http://www.qemu.org

can be compressed. To be able to extract the memory above this limit, the extraction script is signalled using the kernel command line which region it should dump as raw data. However, this requires a 64-bit kernel.

While it should also work when PAE (Physical Address Extension) is used, during the tests it consistently failed after 3.9 gigabytes of physical memory. While this means that the PAE system must be used as only the first three gigabytes resided in the first four gigabytes of the address space (and therefore the physical address is 4.9G instead of 3.9G), something prevented the kernel to modify the page tables in a way to make the higher memory regions accessible. Most likely this is a side effect of hiding that part of the memory from the kernel.

Another limitation of Syslinux is that there is no control over where the modules are loaded. While the core of Syslinux is loaded on a fixed address, the modules are loaded on the heap. The observed behaviour of the heap allocator used by Syslinux is that the first region it uses is the last usable region with an address lower than 0x100000000. It starts to allocate from the beginning of the region. It is observed[27] that[28] many[29] machines have a small usable region at the end of the first four gigabytes of the address space, even when there is more than four gigabytes of RAM available. The firmware allocates some space before this region for its own usage or to store the ACPI configuration. If the compression module fits in this region, this is the location where it will be loaded.

If this small region is missing or above the first megabyte of address space only a single usable region is found, the module is loaded directly after the Syslinux code. Because the location of the Syslinux code is fixed, the modules are loaded starting from address 0x330000.

Syslinux is known to work on both BIOS and UEFI based systems. However, during this project only BIOS based systems have extensively been tested. While the module should work without modification on the UEFI build of Syslinux, this is yet to be confirmed.

[27] http://www.linuxquestions.org/questions/linux-kernel-70/bios-e820-memory-map-4175535958
[28] http://blog.fpmurphy.com/2012/08/uefi-memory-v-e820-memory.html
[29] https://communities.intel.com/thread/60382?start=15&tstart=0

5 Discussion

The last step of this project is to analyse and discuss the results and their meaning. For this a similar order as with the approach and results will be used, starting with the results of the created memory dumps based on the scenarios that are given in the approach.

A side effect of creating artificial scenarios and use them instead of collecting a memory dump from many different machines is that there is no way of knowing if they are representative. Because for a large part the content of the RAM of a system is influenced by the applications that are started and the files that are opened, the scenarios can only be used to give an impression of what can be found instead of data that can be statistically analysed.

An interesting observation that can be made is that there doesn't appear to be a relation between the entropy and the size of the system memory. It is a very real possibility this is an anomaly caused by having only a single 256 megabytes and 512 megabytes sample. However, there is a possibility this can be explained by looking at the behaviour of the memory management in operating systems.

When an operating system doesn't use its full RAM for data and applications, the system tries to use it for other purposes such as file caches. This explains why dumps of scenarios with hard to compress data such as images or that use an encrypted file system have many regions with a very high entropy. At the same time Windows systems actively clear the content of small groups of pages [46] to ensure it can quickly respond to the need for a new page by a process.

The dumps also support the initial assumption that even when there is a lot of hard to compress data in the RAM, there are always certain locations that can be compressed.

The question is if it is always possible to create enough space to safely boot an operating system and what the best algorithm is to do this. Table 6 shows for each algorithm and scenario if the operating system has enough space to boot after the compression.

LZF is chosen because it gives the best overall performance, but the LZO variant LZO1X-1 is a close call. It has the downside that it uses the heap instead of the stack for its memory requirements, but compresses significantly better (5.5% instead of 3.2% for the webserver with images) in the worst-case scenario. While the heap allocations are most likely easy to change to use the stack as it appears to be two fixed size buffers that are allocated, the downside of the heap is that every allocation is equal to more data that is overwritten while the stack size is determined on compile time of Syslinux and fixed for all modules.

This worst-case scenario demonstrates the limit of what is possible with the current proof of concept. With 3.2% gained space on a four gigabytes memory dump, there is around 131 megabytes available to boot the operating system. Testing revealed that the proof of concept needs 82 megabytes as bare minimum. If the same compression gain is reached on a two gigabytes system, booting the system will simply fail. Table 7 shows for each algorithm and several RAM sizes if it would be able to compress the RAM under the worst-case conditions sufficiently enough to be able to boot the operating system.

The proposed solution for the algorithm is to drop compressed blocks until the minimum available space is reached. While this has not yet been implemented in the proof of concept, there is also a different approach possible. If instead of only measuring the entropy to find the starting point, the entropy of the whole memory is measured, a

Algorithm	Windows XP 512 MiB (Desktop)	Windows 7 x86 2 GiB (Desktop)	Windows 7 x64 2 GiB (Desktop)	Windows 8.1 x64 4 GiB (Desktop w. FDE)	Windows 8.1 x64 8 GiB (Desktop, couple hours of usage)	Ubuntu 14.04 x86 2 GiB	Ubuntu 14.04 x64 4 GiB (Desktop w. FDE)	Tails 1.4 2 GiB (Browsing w. Tor)	Tails 1.4 2 GiB (Encrypted folder w. images)	Debian x86 7.8 2 GiB (File server)	Debian 8.1 x64 4 GiB (Web server)	OpenWRT 256 MiB
RLE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Huffman	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
Shannon-Fano	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
BWT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DEFLATE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LZ4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LZF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LZMA2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LZO1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LZO1X-1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LZOmini	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LZW9	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
LZW12	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓
LZW15	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓

Table 6: An overview if the proof of concept would be able to acquire the data and boot the operating system for each combination of compression algorithm and scenario.

better compression algorithm could be used when the overall entropy is very high. For example, LZMA2 has a significantly better compression, at the cost of a much slower compression speed and a memory usage of more than 90 MiB. As this memory usage is higher than the required memory to start the proof of concept, it is most likely not worth the effort. The algorithm with the second best compression results, DEFLATE, could be interesting to explore further as its memory usage is much lower (312 kilobytes) compared to LZMA2.

While a prediction would be preferred of how likely it is this algorithm will be able to recover the whole memory on a certain system, it is very hard to give one. Not only are there way too few samples, the observed distribution of the gained compression doesn't follow one of the typical statistical distributions. It is possible to have a negative compression, while at the same time the ability to compress data is limited by the entropy, according to the theory of Shannon.

In advance it is (very) hard to predict how the content of the memory of a system will look like. For this reason two examples are given with the minimum amount of RAM that the system must have to be able to successfully compress it and boot the operating system. In the worst-case scenario with a webserver filled with images, the system must have almost 2700 megabytes of RAM. In the case of an encrypted volume on a system that runs Tails the minimum amount of RAM would be almost 500 megabytes. In all other scenarios it would work with even less as compression gains of up to 84.7% (Windows 7 64-bit desktop) have been observed. Looking at those numbers and computer systems multiple gigabytes of RAM have become common, it is likely it will succeed.

Algorithm	256.0 M	512.0 M	1.0 G	2.0 G	4.0 G	8.0 G
RLE	✗	✗	✗	✗	✓	✓
Huffman	✗	✗	✗	✗	✗	✗
Shannon-Fano	✗	✗	✗	✗	✗	✓
BWT	✗	✗	✗	✓	✓	✓
DEFLATE	✗	✗	✗	✓	✓	✓
LZ4	✗	✗	✗	✓	✓	✓
LZF	✗	✗	✗	✗	✓	✓
LZMA2	✗	✗	✗	✓	✓	✓
LZO1	✗	✗	✗	✓	✓	✓
LZO1X-1	✗	✗	✗	✓	✓	✓
LZOmini	✗	✗	✗	✓	✓	✓
LZW9	✗	✗	✗	✗	✗	✗
LZW12	✗	✗	✗	✗	✗	✗
LZW15	✗	✗	✗	✗	✗	✗

Table 7: An overview that shows if an algorithm will be able to compress the data of a system with a certain amount of RAM when the data has the same compression ratio as the worst-case scenario.

It is very likely that the RAM usage of the used operating system can be further reduced. While OpenWRT is chosen because it is a ready to use minimal Linux distribution, there is still room for improvement. Currently the root file system is embedded in the kernel binary for ease of use, that has to be decompressed during the boot phase. Also many kernel features that are not used and many drivers for uncommon hardware are still enabled.

The proof of concept has the same limitation that many of the existing solutions have, which is the lack of support to acquire the memory content that is mapped above the address 0x100000000. Because the content is compressed a normal 64-bit kernel can be booted where the acquisition script was used to read the content above the 4G limit, effectively circumventing the limitation. The reason for this is that Syslinux modules run in 32-bit protected mode. An improvement would be if PAE could be used or a way to run certain modules in 64-bit mode is implemented in Syslinux. This way the additional space could be used to create more space for the kernel below the 4G limit.

It is important to remember that when a system has more than three gigabytes of RAM, only thee gigabytes is mapped in the first four gigabytes of address space. Everything more is mapped to higher addresses. The scenario with the worst compression gain that has been observed in this project required 2700 megabytes of RAM to create enough space to boot the kernel. If a situation occurs that is even slightly harder to compress, not everything can be recovered.

6 Conclusion

This research project introduced a new method to extract the content of the RAM of a running system by rebooting it to a special operating system with a modified boot loader. The boot loader compresses the RAM, hides the locations with the compressed data and loads the operating system.

First, the question is answered how compressible the content of the RAM is. This is done by measuring the Shannon entropy of memory dumps that have been created for this project and each represents a realistic scenario. The Shannon entropy limits the theoretical compressed size. None of the scenarios reached the theoretical maximum and the entropy (measured over the full memory dump) is between 3.34 and 5.82 bits per byte, except for scenarios with mostly encrypted or already compressed files such as images. The highest entropy found was 7.86 bits per byte.

The same dumps have been used to select a suitable compression algorithm, based primarily on the memory usage during the compression and secondarily the compression time and gained space. Multiple suitable algorithms have been found, but the LZF algorithm has been chosen to be used in the proof of concept.

The proof of concept has been implemented as Syslinux module. It executes a custom developed acquisition algorithm, compresses the data and hides the compressed data for the operating system by modifying the memory map as is provided by the system firmware. Then the operating system is started, which is a slightly modified version of OpenWRT. This way the operating system has no knowledge of the existence of those parts of the memory. To extract the compressed data, a Python script has been developed that uses a modified version of `/dev/mem` of which the (security) restrictions are removed and has the ability to access memory addresses that do not exist according to the kernel.

The forensic integrity is ensured by generating a cryptographic hash of the input before the data is compressed that acts as checksum. This checksum is added to a custom developed header that is prepend before every compressed block, together with meta information about the origin of the data.

A comparative test with the existing solutions found that at the cost of an (slightly) increased memory usage (4.5 megabytes instead of 1.2 - 1.3 megabytes), the memory can be compressed and a Linux based operating system booted. How much of the memory usage can be attributed to the system firmware could not be established, but existing literature suggests that UEFI firmware could have a significant bigger overhead, however testing this (especially on real system) is difficult.

Compared to booting a Linux based operating system without compressed memory (resulting in 146 megabytes of overwritten data) a significant improvement has been realised. Based on this the conclusion is that the developed technique is feasible and working, and pre-boot memory compression is effective in reducing the amount of data that will be overwritten by a booting system.

7 Future Work

First and foremost extensive testing with the proof of concept should be conducted on a representative set of real machines. Current tests have been limited to several virtual machine solutions which will not cover the tricky corner cases found in real systems. Specially machines with UEFI firmware should be tested. While Syslinux has specialized versions for UEFI machines and the modules should work on every version, the proof of concept manipulates the system on such low level that breaking compatibility is a real risk.

Next, the proposed but not yet implemented features of the algorithm should be implemented in the proof of conept. Currently, the proof of concept uses a fixed starting point, but a dynamic method based on entropy measurement is proposed. Also the method of last resort to drop compressed blocks to make space for the operating system to boot if the minimum is not reached is not implemented.

The ability to acquire memory above the 4G address limit is implemented in the script that extracts the compressed data instead of in the pre-boot compression module itself. This limits the space gained by the compression. The implementation of PAE or a 64-bit mode in Syslinux modules could solve this limitation.

Future research could be conducted to build a large database with memory samples from different machines. This makes it possible with some statistical analysis to predict the likelihood of success for a certain device that is used for a certain purpose.

Another interesting topic could be to find a better algorithm to drop compressed blocks when not enough space is available based on the actual location of operating system objects instead of simply selecting a random block. An example of this is to identify a memory region that will never contain the crypto keys on the common operating systems with the most used encryption solutions.

However, this could make anti-forensics easier by modifying an encryption solution and force it to use exactly that location. When a heuristics based approach is used, it could be circumvented by artificially creating a block in memory that (has a high chance to be) matched by the algorithm and is dropped but in fact contains the critical information.

Acknowledgements

I would like to thank the employees of The Netherlands Forensic Institute for a great time and the insight they gave me in the world of digital forensics. A special mention is for my supervisor, Ruud Schramp, for his endless patience and feedback. I would like to thank Ranieri Argentini for proofreading this paper and his help with interpreting the data.

References

[1] *Getting started with Windows drivers - Virtual address spaces.* Microsoft. http://msdn.microsoft.com/en-us/library/windows/hardware/hh439648 Accessed on: 2015-06-30.

[2] *Memory Management - Virtual Address Space.* Microsoft. http://msdn.microsoft.com/en-us/library/windows/desktop/aa366912 Accessed on: 2015-06-30.

[3] *Physical Memory Acquisition with WindowsSCOPE CaptureGUARD.* BlueRISC Inc., 2012. http://www.windowsscope.com/index.php?option=com_docman&task=doc_download&gid=47 Accessed on: 2015-06-30.

[4] *Unified Extensible Firmware Interface Specification Version 2.5.* Unified EFI, Inc., 2014.

[5] CONFIG_STRICT_DEVMEM: Filter access to /dev/mem (Linux Kernel Driver DataBase), 2015. http://cateee.net/lkddb/web-lkddb/STRICT_DEVMEM.html Accessed on: 2015-06-30.

[6] K. Amari. Techniques and tools for recovering and analyzing data from volatile memory. *SANS Institute*, 2009.

[7] K. Brandenburg. MP3 and AAC explained. In *Audio Engineering Society Conference: 17th International Conference: High-Quality Audio Coding*. Audio Engineering Society, 1999.

[8] R. Carbone, C. Bean, and M. Salois. An in-depth analysis of the cold boot attack. *DRDC Valcartier, Defence Research and Development, Canada, Tech. Rep*, 2011.

[9] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, 2004.

[10] E. Casey, G. Fellows, M. Geiger, and G. Stellatos. The growing impact of full disk encryption on digital forensics. *digital investigation*, 8(2):129–134, 2011.

[11] E. L. Cashin. Subject: Re: Why is a PA converted to a VA in pmd_page? - msg#00139, 2003. http://osdir.com/ml/linux.kernel.kernelnewbies/2003-07/msg00139.html Accessed on: 2015-06-30.

[12] D. Cortjens. Bootable Linux CD / PXE for the remote acquisition of multiple computers. 2014.

[13] S. David. Data compression: The complete reference. *Springer, USA., ISBN*, 10:1846286026, 2007.

[14] L. P. Deutsch. DEFLATE compressed data format specification version 1.3. 1996.

[15] R. Dillon. How Broken is SHA-1?, 2014. http://killring.org/2014/01/05/how-broken-is-sha1 Accessed on: 2015-06-30.

[16] M. Geelnard. Basic Compression Library - Manual, 2006.

[17] V. Georgiev and I. Pavlov. Worst-case LZMA compression size increase?, 2012. http://sourceforge.net/p/sevenzip/discussion/45797/thread/b6bd62f8 Accessed on: 2015-06-30.

[18] M. Gorman. *Understanding the Linux virtual memory manager.* Prentice Hall Upper Saddle River, 2004.

[19] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[20] M. Hale. Information on using the Firewire Address Space, 2012. http://code.google.com/p/volatility/wiki/FirewireAddressSpace Accessed on: 2015-06-30.

[21] P. Hannay and A. Woodward. Cold Boot Memory Acquisition: An Investigation Into Memory Freezing and Data Retention Claims. In *Security and Management*, pages 620–622, 2008.

[22] T. Holwerda. SuperFetch: How it works & myths, 2009. http://www.osnews.com/story/21471/SuperFetch_How_it_Works_Myths Accessed on: 2015-06-30.

[23] C. Hosmer. Proving the integrity of digital evidence with time. *International Journal of Digital Evidence*, 1(1):1–7, 2002.

[24] D. A. Huffman et al. A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[25] I. Kollár and I. Kollár. Forensic RAM dump image analyser. *Department of Software Engineering, Charles University, Prague*, 2010.

[26] D. L. Lewis. The Hash Algorithm DilemmaHash Value Collisions, 2008. http://www.forensicmag.com/articles/2008/12/hash-algorithm-dilemma%E2%80%93hash-value-collisions Accessed on: 2015-06-30.

[27] A. Lineberry. Malicious Code Injection via /dev/mem. *Black Hat Europe*, 2009.

[28] R. Love. *Linux kernel development*. Pearson Education, 2010.

[29] M. Marcos. Without a Trace: Fileless Malware Spotted in the Wild, 2015. http://blog.trendmicro.com/trendlabs-security-intelligence/without-a-trace Accessed on: 2015-06-30.

[30] E. Martignetti. *What makes it PAGE? - The Windows 7 (x64) Virtual Memory Manager*. CreateSpace Independent Publishing, 2012.

[31] A. Martin. FireWire memory dump of a windows XP computer: a forensic approach. *Black Hat DC*, pages 1–13, 2007.

[32] W. McGrew. msramdmp: McGrew Security RAM dumper. *McGrew Security*, 2008.

[33] G. Osbourne. Memory Forensics: Review of Acquisition and Analysis Techniques. Technical report, DTIC Document, 2013.

[34] N. Phamdo. Theory of Data Compression. http://www.data-compression.com/theory.shtml Accessed on: 2015-06-30.

[35] C. Prosise, K. Mandia, and M. Pepe. *Incident response & computer forensics*. McGraw-Hill/Osborne, 2003.

[36] G. G. Richard and A. Case. In lieu of swap: Analyzing compressed RAM in Mac OS X and Linux. *Digital Investigation*, 11:S3–S12, 2014.

[37] L. Rizzo. A very fast algorithm for RAM compression. *ACM SIGOPS Operating Systems Review*, 31(2):36–45, 1997.

[38] M. Russinovich. Pushing the Limits of Windows: Paged and Nonpaged Pool, 2009. http://blogs.technet.com/b/markrussinovich/archive/2009/03/26/3211216.aspx Accessed on: 2015-06-30.

[39] M. Russinovich, D. Solomon, and A. Ionescu. Windows Internals Part 2, 2012.

[40] A. Said. Introduction to arithmetic coding-theory and practice. *Hewlett Packard Laboratories Report*, 2004.

[41] D. Salihun. *BIOS Disassembly Ninjutsu Uncovered (Uncovered series)*. A-List Publishing, 2006.

[42] R. Schramp. Ram Memory Acquisition Using Live Bios Modification. *Presentation, OHM2013*, 2013. http://archive.org/details/D3T312201308022200... Accessed on: 2015-06-30.

[43] J. Seward. bzip2, 2007. http://www.bzip.org/1.0.5/bzip2.txt Accessed on: 2015-06-30.

[44] C. Shannon. A Mathematical Theory of Communication. 1948.

[45] M. Simpson, S. Biswas, and R. Barua. Analysis of Compression Algorithms for Program Data. *University of Maryland*, 2003.

[46] T. Sneath. PDC10: Mysteries of Windows Memory Management Revealed: Part Two, 2010. http://blogs.msdn.com/b/tims/... Accessed on: 2015-06-30.

[47] S. Turner and L. Chen. Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms. 2011.

[48] E. van den Haak. Remote data acquisition on block devices in large environments. 2014.

[49] R. van Riel and D. V. Lehn. HighMemory, 2004-2009. http://linux-mm.org/HighMemory Accessed on: 2015-06-30.

[50] E. C. Ventura. Memory Layout for Windows XP, 2005. http://www.openrce.org/reference_library/files/reference/Windows%20Memory%20Layout,%20User-Kernel%20Address%20Spaces.pdf Accessed on: 2015-06-30.

[51] S. Vogl, J. Pfoh, T. Kittel, and C. Eckert. Persistent data-only malware: Function Hooks without Code. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[52] G. K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1991.

[53] J. Yao and V. J. Zimmer. White Paper A Tour beyond BIOS Memory Map Design in UEFI BIOS. 2015.

[54] J. Yao, V. J. Zimmer, E. Li, and C. Li. White Paper A Tour Beyond BIOS Implementing the Tiny Quark Design. 2014.

[55] P. Zijlstra. HIGH MEMORY HANDLING. http://www.kernel.org/doc/Documentation/vm/highmem.txt Accessed on: 2015-06-30.

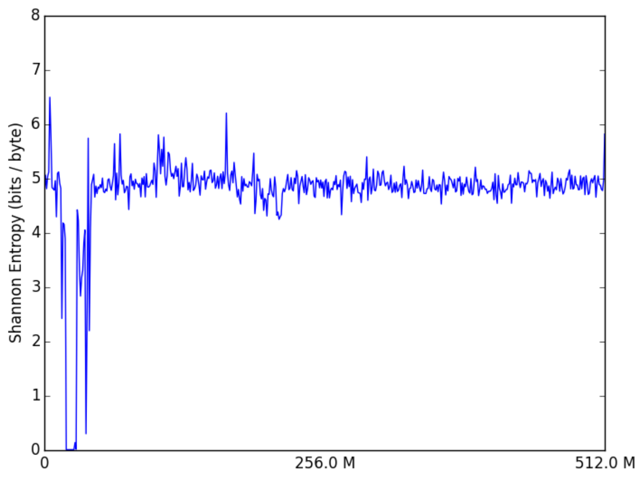
[56] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[57] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978.

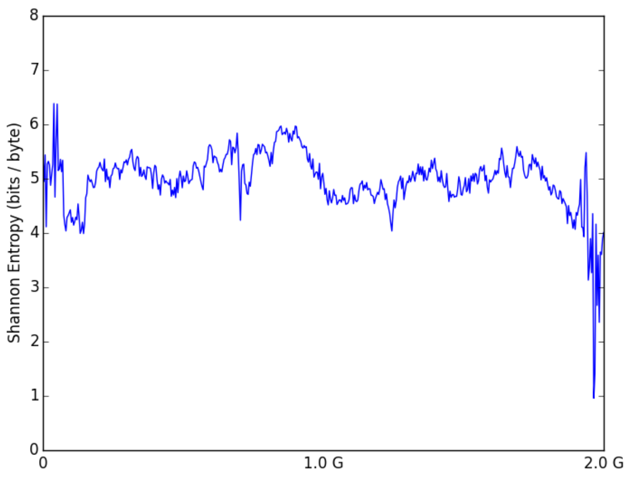
Appendices

A RAM Content Entropy Graphs

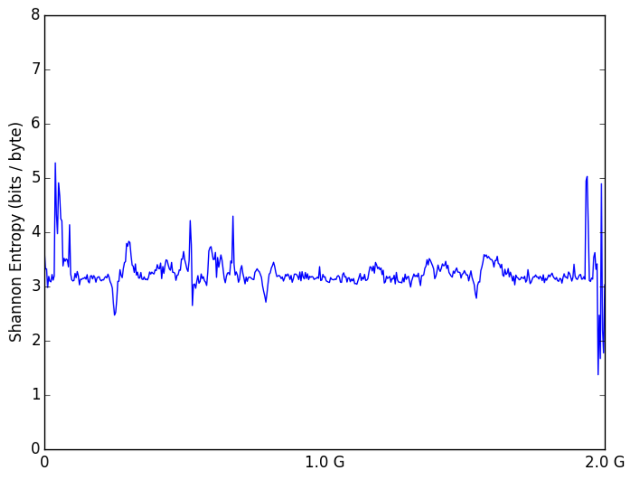
This appendix contains the graphs of the entropy for each of the memory dumps created during this project. First, the entropy is plotted as function over the size of the image (measured over 512 equally sized blocks). The second group of graphs is much more detailed. Each pixel corresponds with a block of data with a size between 2 and 64 kilobytes, depending on the size of the memory dump. The pixels are grouped in vertical rows, with each row corresponding to $\frac{1}{1024}$ of the dump. The colours are used to indicate the entropy (blue for (very) low, red for (very) high) on a certain location.



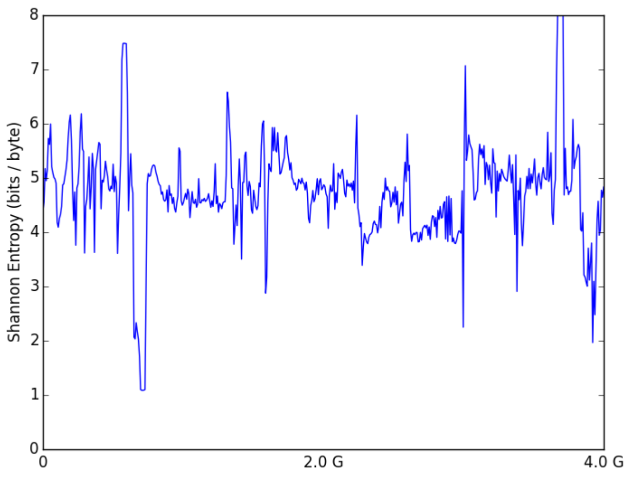
(a) Windows XP 512 MiB (Desktop)



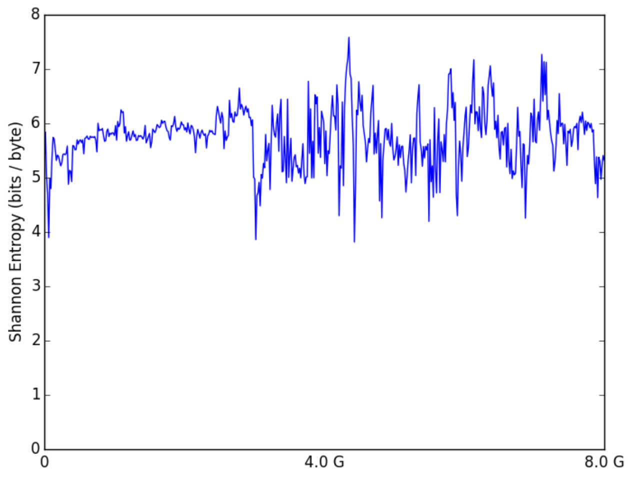
(b) Windows 7 x86 2 GiB (Desktop)



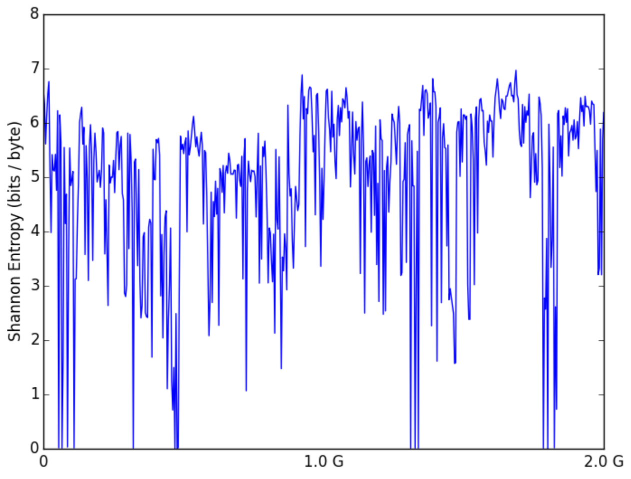
(c) Windows 7 x64 2 GiB (Desktop)



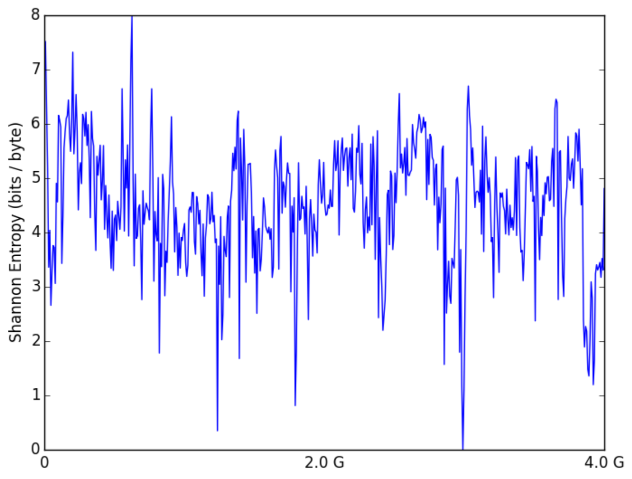
(d) Windows 8.1 x64 4 GiB (Desktop w. FDE)



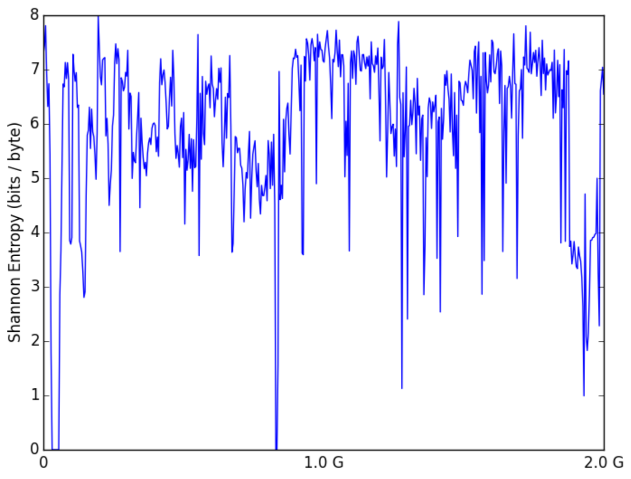
(e) Windows 8.1 x64 8 GiB (Desktop, after couple hours of normal usage)



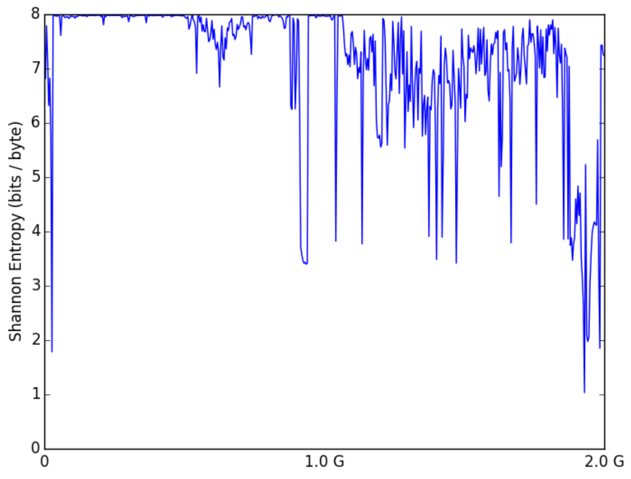
(f) Ubuntu 14.04 x86 2 GiB



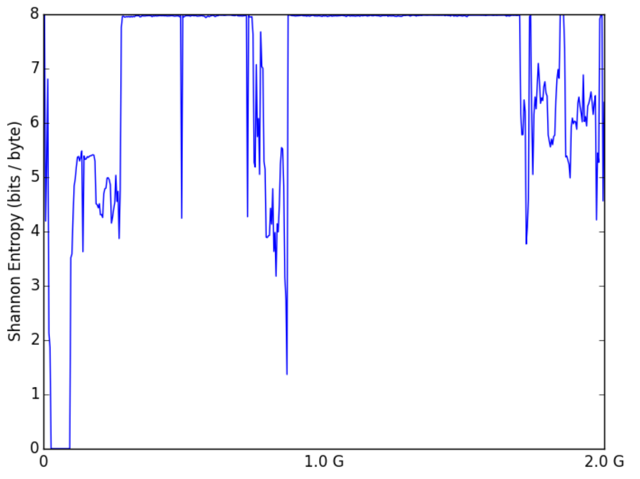
(g) Ubuntu 14.04 x64 4 GiB (Desktop w. FDE)



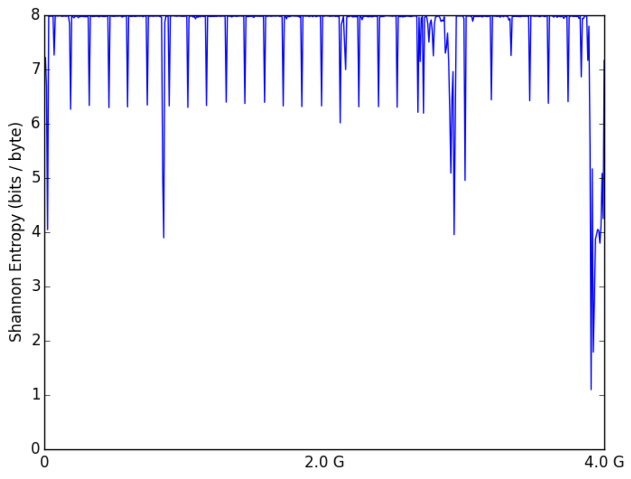
(h) Tails 1.4 2 GiB (Browsing w. Tor)



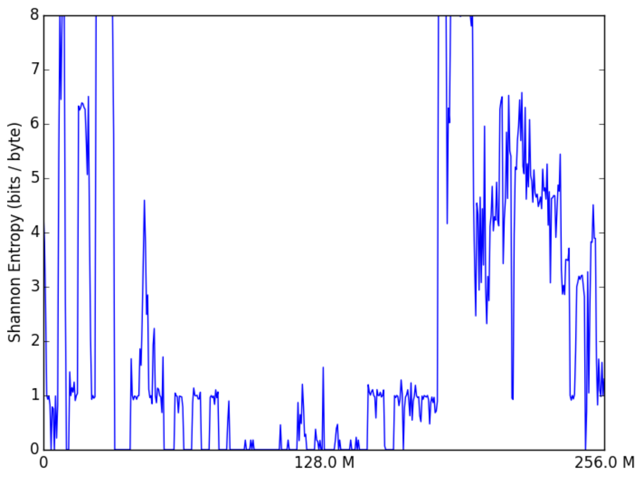
(i) Tails 1.4 2 GiB (Encrypted folder w. images)



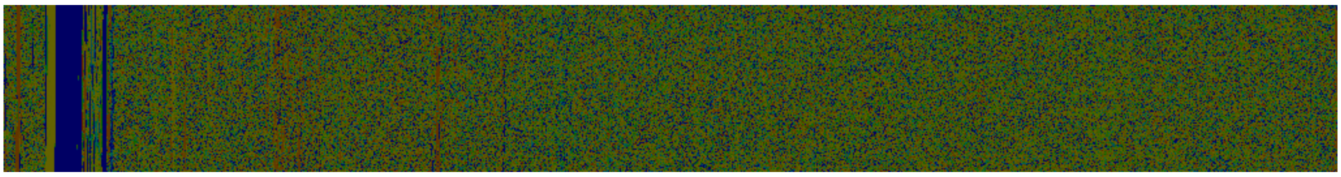
(j) Debian x86 7.8 2 GiB (File server)

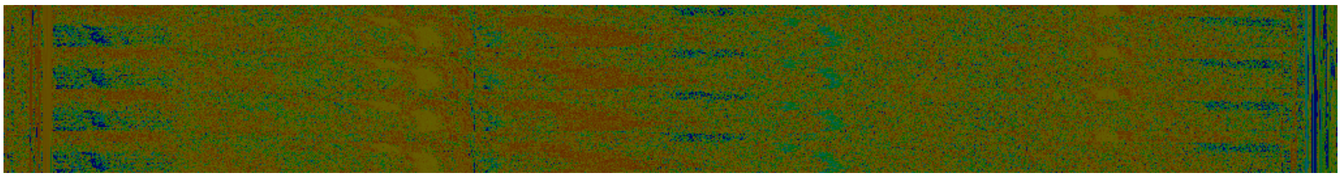


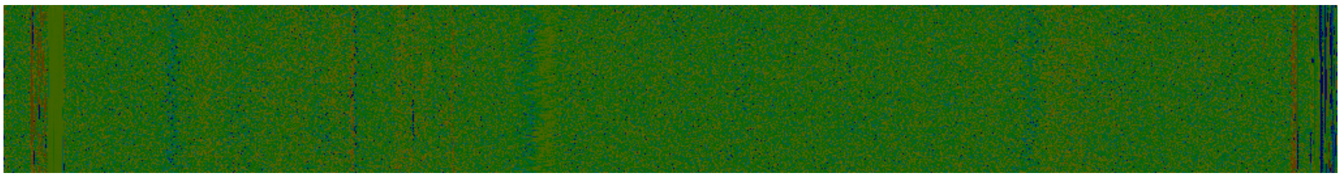
(k) Debian 8.1 x64 4 GiB (Web server)

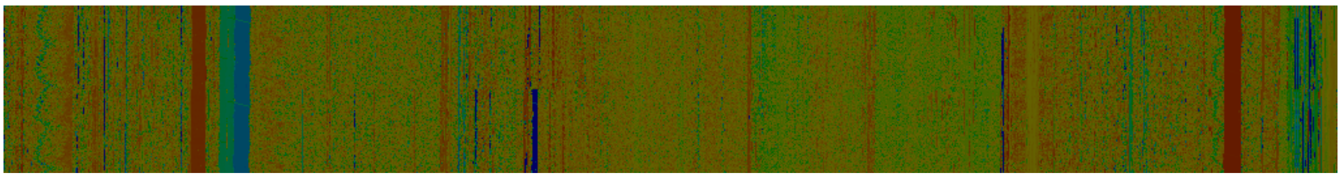


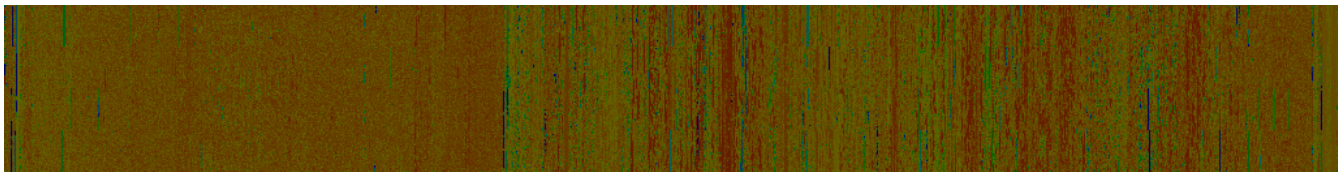
(l) OpenWRT 256 MiB

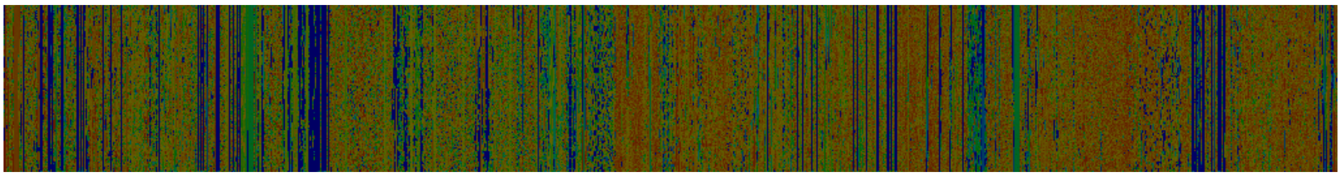

(a) Windows XP 512 MiB (Desktop)

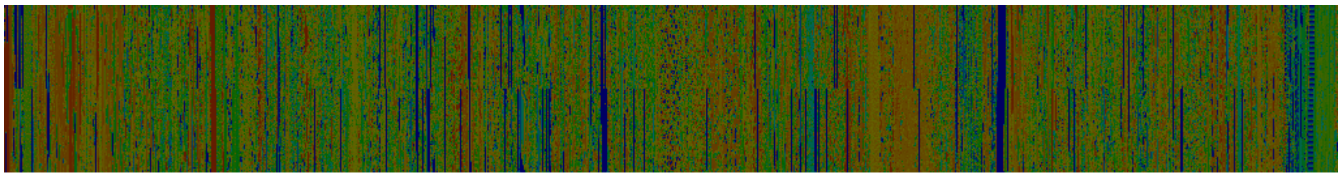

(b) Windows 7 x86 2 GiB (Desktop)

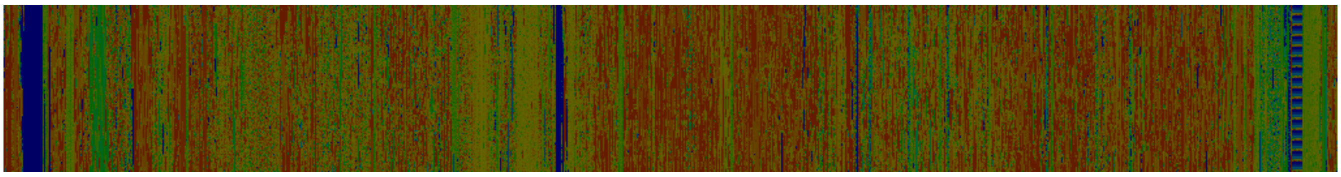

(c) Windows 7 x64 2 GiB (Desktop)

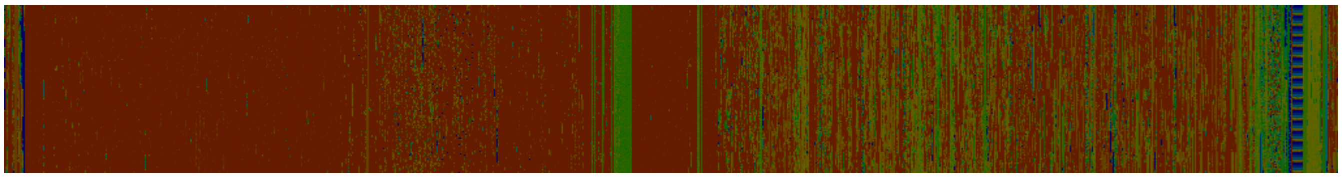

(d) Windows 8.1 x64 4 GiB (Desktop w. FDE)

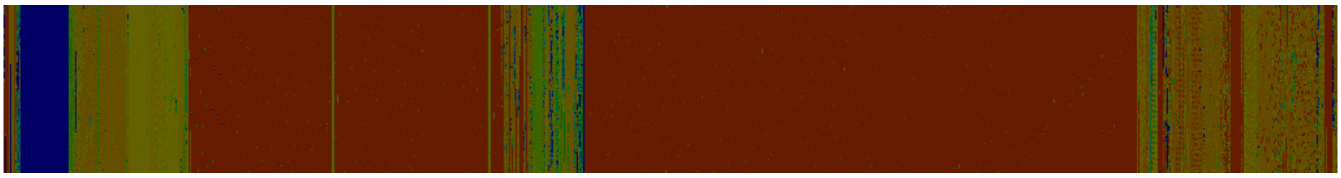

(e) Windows 8.1 x64 8 GiB (Desktop, after couple hours of normal usage)

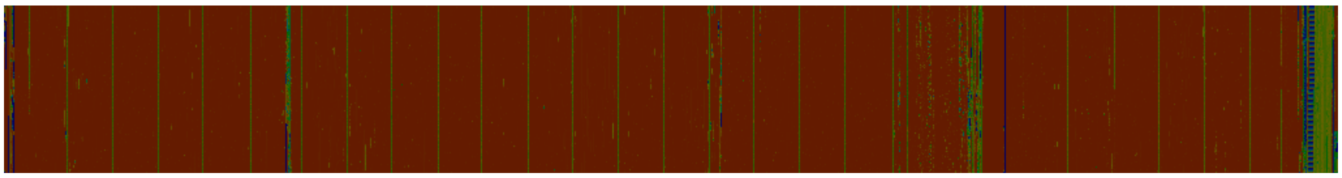

(f) Ubuntu 14.04 x86 2 GiB

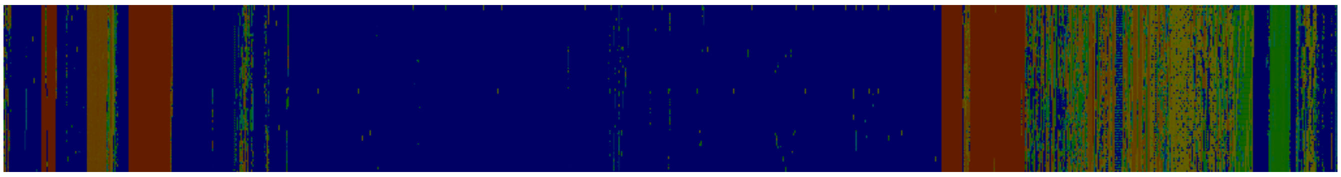

(g) Ubuntu 14.04 x64 4 GiB (Desktop w. FDE)


(h) Tails 1.4 2 GiB (Browsing w. Tor)


(i) Tails 1.4 2 GiB (Encrypted folder w. images)


(j) Debian x86 7.8 2 GiB (File server)


(k) Debian 8.1 x64 4 GiB (Web server)


(l) OpenWRT 256 MiB

B Detailed Compression Algorithm Benchmark Results

Algorithm	Windows XP 512 MiB (Desktop)	Windows 7 x86 2 GiB (Desktop)	Windows 7 x64 2 GiB (Desktop)	Windows 8.1 x64 4 GiB (Desktop w. FDE)	Windows 8.1 x64 8 GiB (Desktop, couple hours of usage)	Ubuntu 14.04 x86 2 GiB	Ubuntu 14.04 x64 4 GiB (Desktop w. FDE)	Tails 1.4 2 GiB (Browsing w. Tor)	Tails 1.4 2 GiB (Encrypted folder w. images)	Debian x86 7.8 2 GiB (File server)	Debian 8.1 x64 4 GiB (Web server)	OpenWRT 256 MiB
RLE	1.6s	5.9s	5.5s	47.8s	11.4s	5.9s	93.3s	5.6s	5.0s	4.8s	59.7s	0.8s
Huffman	9.2s	37.0s	26.4s	61.3s	46.3s	38.1s	58.7s	44.2s	49.0s	51.7s	n/a	3.3s
Rice8	30.5s	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	13.2s
Rice16	19.9s	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	8.2s
Rice32	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	14.9s
LZ77	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Shannon-Fano	9.3s	38.5s	26.3s	76.1s	83.7s	39.3s	74.1s	43.4s	49.4s	51.7s	126.6s	3.5s
RLE-LZ77-Huffman	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
BWT	105.9s	249.6s	250.2s	514.9s	480.3s	225.4s	354.0s	233.2s	276.9s	267.6s	914.2s	10.9s
DEFLATE	15.8s	57.8s	33.4s	113.7s	117.0s	62.6s	121.9s	73.5s	70.1s	63.8s	134.2s	4.4s
LZ4	1.0s	3.6s	2.0s	7.3s	7.1s	3.6s	7.1s	3.8s	2.2s	1.7s	2.5s	0.2s
LZF	1.9s	7.6s	4.0s	14.5s	16.2s	7.8s	14.0s	9.3s	11.6s	11.1s	25.9s	0.6s
LZMA2	177.2s	628.5s	322.9s	1313.7s	1353.5s	639.2s	1497.6s	781.7s	845.4s	793.0s	1810.7s	39.4s
LZO1	2.1s	9.6s	4.0s	17.4s	23.0s	9.2s	15.4s	12.1s	17.6s	17.0s	41.1s	0.7s
LZO1X-1	1.0s	3.3s	1.9s	6.6s	8.5s	3.5s	7.2s	3.6s	2.0s	1.7s	2.2s	0.2s
LZOmini	1.0s	3.5s	2.0s	7.0s	38.7s	3.7s	7.4s	3.8s	2.1s	1.8s	2.4s	0.2s
LZW9	10.5s	44.9s	40.4s	96.1s	106.9s	43.9s	82.5s	57.2s	72.6s	71.7s	n/a	3.5s
LZW12	23.6s	109.4s	65.2s	n/a	263.8s	101.5s	n/a	143.4s	217.8s	215.7s	n/a	8.0s
LZW15	34.6s	167.3s	75.7s	n/a	391.8s	147.4s	n/a	210.3s	313.6s	300.3s	n/a	10.7s

Table 8: The time in seconds to compress a memory dump with each of the algorithms.

	Windows XP 512 MiB (Desktop)		Windows 7 x86 2 GiB (Desktop)		Windows 7 x64 2 GiB (Desktop)		Windows 8.1 x64 4 GiB (Desktop w. FDE)		Windows 8.1 x64 8 GiB (Desktop, couple hours of usage)		Ubuntu 14.04 x86 2 GiB		Ubuntu 14.04 x64 4 GiB (Desktop w. FDE)		Tails 1.4 2 GiB (Browsing w. Tor)		Tails 1.4 2 GiB (Encrypted folder w. images)		Debian x86 7.8 2 GiB (File server)		Debian 8.1 x64 4 GiB (Web server)		OpenWRT 256 MiB	
Algorithm	CS	G	CS	G	CS	G	CS	G	CS	G	CS	G	CS	G	CS	G	CS	G	CS	G	CS	G	CS	G
RLE	352.6 M	31.1%	1.3 G	36.7%	1.4 G	32.0%	2.9 G	27.8%	2.9 G	26.8%	1.3 G	33.8%	2.5 G	37.8%	1.6 G	20.5%	1.8 G	8.9%	1.8 G	9.2%	3.9 G	3.1%	68.3 M	73.3%
Huffman	310.6 M	39.3%	1.3 G	36.9%	864.1 M	57.8%	2.0 G	50.4%	1.2 G	70.9%	1.3 G	34.6%	1.9 G	51.8%	1.6 G	21.7%	1.9 G	7.1%	1.9 G	5.6%	n/a	n/a	107.0 M	58.2%
Rice8	430.1 M	16.0%	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	168.3 M	34.3%
Rice16	410.0 M	19.9%	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	157.4 M	38.5%
Rice32	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	256.0 M	-0.0%
LZ77	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Shannon-Fano	325.9 M	36.3%	1.3 G	32.8%	869.7 M	57.5%	2.6 G	34.1%	2.9 G	26.5%	1.4 G	31.8%	2.6 G	35.5%	1.6 G	21.4%	1.9 G	6.8%	1.9 G	4.5%	4.0 G	1.1%	115.6 M	54.9%
RLE-LZ77-Huffman	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
BWT	152.0 M	70.3%	744.9 M	63.6%	162.7 M	92.1%	1.1 G	72.1%	1.7 G	58.1%	656.7 M	67.9%	997.5 M	75.6%	959.6 M	53.1%	1.5 G	22.9%	1.5 G	26.6%	3.7 G	6.8%	42.1 M	83.5%
DEFLATE	144.1 M	71.9%	704.9 M	65.6%	174.1 M	91.5%	1.1 G	73.0%	1.6 G	60.0%	642.2 M	68.6%	1018.2 M	75.1%	950.9 M	53.6%	1.5 G	23.3%	1.5 G	26.4%	3.7 G	6.9%	42.9 M	83.2%
LZ4	196.4 M	61.6%	869.1 M	57.6%	247.7 M	87.9%	1.4 G	65.1%	1.8 G	53.5%	819.2 M	60.0%	1.3 G	66.2%	1.1 G	44.7%	1.6 G	18.9%	1.5 G	23.3%	3.8 G	4.6%	51.5 M	79.9%
LZF	195.8 M	61.8%	869.4 M	57.5%	313.3 M	84.7%	1.4 G	64.0%	1.9 G	52.8%	825.3 M	59.7%	1.4 G	66.0%	1.1 G	43.8%	1.7 G	17.3%	1.6 G	19.9%	3.9 G	3.2%	53.1 M	79.3%
LZMA2	103.6 M	79.8%	543.1 M	73.5%	107.1 M	94.8%	795.9 M	80.6%	1.3 G	67.4%	508.5 M	75.2%	757.2 M	81.5%	780.4 M	61.9%	1.5 G	26.2%	1.4 G	29.2%	3.7 G	7.6%	37.4 M	85.4%
LZO1	200.6 M	60.8%	881.8 M	56.9%	335.9 M	83.6%	1.5 G	63.0%	1.9 G	52.5%	837.4 M	59.1%	1.4 G	65.3%	1.1 G	43.7%	1.6 G	18.7%	1.6 G	21.3%	3.8 G	5.6%	53.1 M	79.3%
LZO1X-1	196.2 M	61.7%	892.6 M	56.4%	273.9 M	86.6%	1.5 G	63.7%	2.0 G	50.8%	849.1 M	58.5%	1.3 G	66.5%	1.1 G	43.6%	1.6 G	18.3%	1.6 G	22.1%	3.8 G	5.5%	51.2 M	80.0%
LZOmini	193.4 M	62.2%	878.8 M	57.1%	268.0 M	86.9%	1.4 G	64.5%	1.9 G	51.6%	833.5 M	59.3%	1.3 G	67.2%	1.1 G	44.3%	1.6 G	18.6%	1.6 G	22.4%	3.8 G	5.5%	50.5 M	80.3%
LZW9	217.0 M	57.6%	996.4 M	51.3%	842.7 M	58.9%	2.0 G	49.9%	2.4 G	39.9%	961.5 M	53.1%	1.7 G	58.1%	1.3 G	35.5%	1.8 G	8.6%	1.8 G	9.2%	n/a	n/a	60.2 M	76.5%
LZW12	207.0 M	59.6%	992.0 M	51.6%	511.6 M	75.0%	n/a	n/a	2.4 G	40.8%	905.8 M	55.8%	n/a	n/a	1.3 G	34.6%	2.1 G	-4.7%	2.0 G	-2.4%	n/a	n/a	60.0 M	76.5%
LZW15	225.7 M	55.9%	1.1 G	47.4%	365.1 M	82.2%	n/a	n/a	2.5 G	36.4%	964.4 M	52.9%	n/a	n/a	1.4 G	31.7%	2.2 G	-8.9%	2.1 G	-4.4%	n/a	n/a	61.4 M	76.0%

Table 9: This table shows the remaining size (or compressed size, CS) and how much space is gained (G) by the compression for every memory dump with each of the tested compression algorithms.

	Windows XP 512 MiB (Desktop)		Windows 7 x86 2 GiB (Desktop)		Windows 7 x64 2 GiB (Desktop)		Windows 8.1 x64 4 GiB (Desktop w. FDE)		Windows 8.1 x64 8 GiB (Desktop, couple hours of usage)		Ubuntu 14.04 x86 2 GiB		Ubuntu 14.04 x64 4 GiB (Desktop w. FDE)		Tails 1.4 2 GiB (Browsing w. Tor)		Tails 1.4 2 GiB (Encrypted folder w. images)		Debian x86 7.8 2 GiB (File server)		Debian 8.1 x64 4 GiB (Web server)		OpenWRT 256 MiB	
Algorithm	Stack	Heap	Stack	Heap	Stack	Heap	Stack	Heap	Stack	Heap	Stack	Heap	Stack	Heap	Stack	Heap	Stack	Heap	Stack	Heap	Stack	Heap	Stack	Heap
RLE	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0
Huffman	16.8 K	0	16.8 K	0	16.9 K	0	16.9 K	0	16.8 K	0	16.8 K	0	16.8 K	0	16.9 K	0	16.8 K	0	16.8 K	0	n/a	n/a	16.8 K	0
Rice8	424	0	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	424	0
Rice16	424	0	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	424	0
Rice32	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	424	0
LZ77	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Shannon-Fano	4.9 K	0	4.9 K	0	4.8 K	0	4.9 K	0	4.9 K	0	4.9 K	0	4.7 K	0	4.9 K	0	4.9 K	0	4.9 K	0	4.9 K	0	4.9 K	0
RLE-LZ77-Huffman	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
BWT	5.3 K	1.1 M	5.3 K	1.1 M	5.3 K	1.1 M	5.3 K	1.1 M	5.3 K	1.1 M	5.3 K	1.1 M	5.3 K	1.1 M	5.3 K	1.1 M	5.3 K	1.1 M	5.3 K	1.1 M	5.4 K	1.1 M	5.3 K	1.1 M
DEFLATE	672	261.8 K	672	261.8 K	672	261.8 K	672	261.8 K	672	261.8 K	672	261.8 K	672	261.8 K	672	261.8 K	672	261.8 K	672	261.8 K	672	261.8 K	672	261.8 K
LZ4	16.4 K	0	16.4 K	0	16.4 K	0	16.4 K	0	16.4 K	0	16.4 K	0	17.1 K	0	16.4 K	0	16.4 K	0	17.1 K	0	17.1 K	0	17.1 K	0
LZF	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0	3.2 K	0
LZMA2	1.6 K	93.1 M	1.6 K	93.1 M	1.6 K	93.1 M	1.6 K	93.1 M	1.6 K	93.1 M	1.6 K	93.1 M	1.6 K	93.1 M	1.6 K	93.1 M	1.6 K	93.1 M	1.6 K	93.1 M	1.6 K	93.1 M	1.6 K	93.1 M
LZO1	408	64.0 K	408	64.0 K	376	64.0 K	376	64.0 K	416	64.0 K	376	64.0 K	376	64.0 K	408	64.0 K	408	64.0 K	408	64.0 K	416	64.0 K	376	64.0 K
LZO1X-1	408	16.0 K	408	16.0 K	408	16.0 K	408	16.0 K	408	16.0 K	408	16.0 K	408	16.0 K	408	16.0 K	408	16.0 K	408	16.0 K	408	16.0 K	408	16.0 K
LZOmini	432	16.0 K	432	16.0 K	432	16.0 K	432	16.0 K	432	16.0 K	432	16.0 K	432	16.0 K	432	16.0 K	432	16.0 K	432	16.0 K	432	16.0 K	432	16.0 K
LZW9	464	512.3 M	464	2.0 G	464	2.0 G	n/a	n/a	n/a	n/a	464	2.0 G	464	4.0 G	464	4.0 G	464	4.0 G	464	4.0 G	n/a	n/a	464	128.3 M
LZW12	464	514.0 M	464	2.0 G	464	1.0 G	480	4.0 G	n/a	n/a	464	2.0 G	480	4.0 G	464	4.0 G	n/a	n/a	n/a	n/a	n/a	n/a	464	130.0 M
LZW15	464	528.0 M	464	4.0 G	464	1.0 G	480	4.0 G	n/a	n/a	464	2.0 G	480	4.0 G	464	4.0 G	n/a	n/a	n/a	n/a	n/a	n/a	464	144.0 M

Table 10: This last table shows the runtime memory usage (for both the stack and the heap) for each of the compression algorithms.

C Overview Source Code Components

Component	Description	Repository URL[30]
Compression Test Suite	The developed tools to benchmark the compression algorithms.	GitHub
Syslinux modules	Contains the following Syslinux modules: - meminfo: Prints the memory map of a system; - memcompress: Compress the current RAM content and boot the Linux kernel; - memchunkfinder: Locate compressed blocks in the RAM of the current system; - memread: Read the content of the RAM, show how much of it is filled with a known pattern and the memory ranges where it was (not) found; - memwrite: Fill the RAM with a known pattern.	GitHub
Acquisition scripts	The Python scripts that are used to extract and decompress the compressed blocks.	GitHub
Kernel Patch	The required modifications to remove the restrictions from `/dev/mem`.	GitHub
MemCarve.bt	Carve a memory dump for the block header marker, so it can be manually extracted and decompressed.	GitHub
QEMU Patch	The required modifications to fill the VM memory with a known pattern.	GitHub

Table 11: An overview of the developed components and the URL to the repository where they can be found.

[30]Main project repository: `https://github.com/MartijnB/memcompress`