MemScrimper: Time- and Space-Efficient Storage of Malware Sandbox Memory Dumps

Michael Brengel[1] and Christian Rossow[2]

CISPA, Saarland University
{michael.brengel|rossow}@cispa.saarland

Abstract. We present MemScrimper, a novel methodology to compress memory dumps of malware sandboxes. MemScrimper is built on the observation that sandboxes always start at the same system state (i.e., a sandbox snapshot) to analyze malware. Therefore, memory dumps taken after malware execution inside the same sandbox are substantially similar to each other, which we can use to only store the differences introduced by the malware itself. Technically, we compare the pages of those memory dumps against the pages of a reference memory dump taken from the same sandbox and then deduplicate identical or similar pages accordingly. MemScrimper increases data compression ratios by up to 3894.74% compared to standard compression utilities such as `7zip`, and reduces compression and decompression times by up to 72.48% and 41.44%, respectively. Furthermore, MemScrimper's internal storage allows to perform analyses (e.g., signature matching) on compressed memory dumps more efficient than on uncompressed dumps. MemScrimper thus significantly increases the retention time of memory dumps and makes longitudinal analysis more viable, while also improving efficiency.

1 Introduction

As of 2018, the number of new and potentially malicious files has risen to over 250,000 per day [2]. Naturally, defenders strive to analyze as many of these files as possible. Malware sandboxes play a key role in this process and represent *the* prevalent mechanism to study the behavior of unknown programs. To cope with the daily flood of unknown files, malware sandboxes can utilize hardware virtualization (e.g., Intel VT) to speed up parallel processing. While significant progress has been made in scaling malware analysis [13, 9, 14, 6, 5], mechanisms how to efficiently store the results of malware analysis has received little attention so far. Obviously, without detailed planning and well-tuned mechanisms, long-term storage of analysis results hardly scales. This is particularly challenging for unstructured and space-consuming outputs such as memory snapshots ("images" or "dumps") taken during or after malware analysis. Although storing memory images seems unfeasible at first, a persistent storage is appealing. First, a long-term storage of memory dumps enables for a large variety of promising forensic analyses, such as on the evolution of malware over time (e.g., lineage), extracting malware configuration files (e.g., to see targets of banking trojans), or

group similar malware files by clustering them on unpacked malware code [10]. Second, a long-term storage of memory artifacts aids to grasp the concept drift in malware, e.g., to reevaluate the accuracy of code signatures (e.g., YARA) over a long-term data set. Yet, as compelling as it sounds, naïvely storing memory dumps for each malware analysis run does not scale. Given that the storage footprint of regular memory dumps equals to the virtual memory given to a sandbox, merely analyzing 10,000 malware samples requires already about 5 TiB of disk space (assuming 512 MiB per dump) to persistently store malware dumps.

To tackle this problem, we propose time- and space-efficient storage methodologies for dumps stemming from malware sandboxes. We first study the straw man solution to the problem and assess how regular file compression algorithms perform. While existing compression utilities reduce storage demands by an order of magnitude, they (i) have a relatively poor computation performance, and (ii) do not achieve compression ratios that would allow for persistent data storage. We thus propose novel methodologies that (i) better utilize the structure of memory dumps (i.e., memory pages), and (ii) leverage the fact that we are actually only interested in *changes* caused by the malware under analysis.

Our main idea is borrowed from genome data processing in biology, which faces similar scaling problems for persistently storing genomes. While a human's genome consists of about 3 *billion* pairs, only very few (around 0.1%) of these pairs are actually different per human. Thus, by just storing these *mutations*, one can significantly reduce the space footprint of a genome. Transposing this idea to malware, we observed that malware changes the memory only slightly compared to a system *before* infection. Our general idea is thus to store only these differences compared to a base snapshot that is taken before analyzing a particular file. That is, given that sandboxes usually use snapshots to restore to a clean state after analyzing a file [22], we first save a so called reference memory dump from this state. This reference dump is identical for all files analyzed with a given sandbox and thus needs to be persisted only once. Then, after having analyzed a malware in the sandbox, we compare the memory dump of the malware-infected system with the reference dump and only store the differences between the two. Furthermore, we can leverage page-wise memory deduplication, as deployed in operating systems or incremental checkpoints of hypervisors [20, 16, 17], to reduce the dump size. As a final optimization, we search for pages that have only slightly been altered compared to the reference dump, and if so, we store only the changes instead of the entire page.

We implement MemScrimper, a prototype for this methodology. Our results show that MemScrimper outperforms standard compression utilities such as `7zip` in our sandbox context. MemScrimper improves the data compression ratio by up to 3894.74%, while reducing compression and decompression times by up to 72.48% and 41.44%, respectively. Storing a 2 GiB memory dump with MemScrimper requires roughly 5 MiB ($< 0.3\%$) on average. Finally, we measure if forensic analysis can be carried out on such compressed dumps by matching YARA [1] rules on them. This demonstrates how we can speed up

the forensic analysis with our methodology due to the smaller file size of the compressed memory dumps.

2 Methodology

We first motivate our idea and give technical preliminaries which are required to grasp the concepts of our approach. We then describe the two main compression ideas, namely *intra-deduplication* and *inter-deduplication*, which are implemented by MemScrimper (Section 3) and which will be evaluated in Section 4.

2.1 Motivation and High-Level Idea

Our proposed methodology is designed to be used in large-scale malware analysis systems. In such a setting, the following high-level workflow is usually deployed:

1. **Set up an analysis environment.** This involves installing an operating system and configuring it to run the malware. The latter includes disabling security mechanisms such as firewalls or built-in anti-virus solutions to increase the likelihood that the malware will expose its malicious behavior.
2. **Create a snapshot.** To ensure that malware exposes its malicious behavior and to ease the analysis, the malware should be executed in an untampered environment. This allows to eliminate side effects where one malware might hinder another malware from executing and also reduces noise in the behavioral data. To guarantee such a clean state, snapshots can revert the state of the system to the one immediately after the setup process (step 1). Snapshots are common in both virtualization-based and bare-metal sandboxes [12].
3. **Analyze the malware.** The first two steps will be executed once per analysis machine. After that, the malware is executed for a predefined amount of time after which the analysis data (including the memory dump) is collected, and finally, the system is restored to its original snapshot state. This procedure is repeated for every malware sample that will be executed.

The core design principle of MemScrimper is based on two intuitive assumptions. First, the snapshot guarantees that the memory will be bit-by-bit identical every time the execution of a malware sample starts. Second, it is unlikely that a malware will modify huge parts of the memory of the analysis machine. This follows the intuition that malware usually has a small memory footprint to operate stealthily and to ensure that it runs on a variety of systems including systems where memory is scarce. These two observations imply that the memory dumps resulting from executions of different malware samples on the same analysis environment will share a significant amount of their memory contents. By simply storing memory dumps in plain, the space is not efficiently used, as the same memory content is stored over and over again. This is also true if the memory dump is deflated with standard compression utilities such as `gzip` or `7zip`, since the same memory content is redundantly compressed. With MemScrimper we fill this gap.

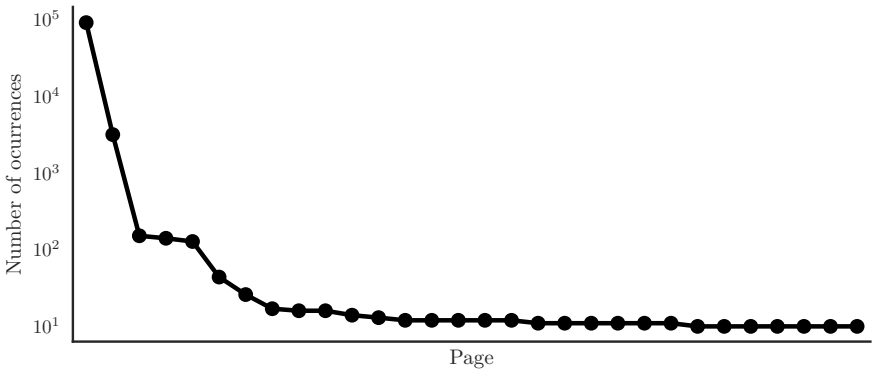

Fig. 1. The number of occurrences of the 30 most common pages of a 512 MiB memory dump taken from a Windows XP analysis machine.

2.2 Terminology

The central objects that MemScrimper works on are *memory dumps*. A memory dump is a bit-by-bit copy of the *physical memory* of a system taken at some point in time. Modern operating systems and hardware use the concept of *virtual memory*, which gives each running process the impression of having a large contiguous space of memory. This virtual memory space is then mapped to the physical memory space, which does not have to be contiguous. The granularity of this mapping is determined by the *page size*. The page size is an important piece of structural information that dictates the way we process memory dumps. We will focus on the x86 platform, which has a page size of 4,096 bytes. Formally, we consider a memory dump $\mathcal{D}$ as an ordered sequence of pages, i.e., $\mathcal{D} = [P_1, \ldots, P_n]$ where $P_i \in \mathcal{D}$ is the ith page of $\mathcal{D}$ — we will also call i a *page number*. A page is an ordered sequence of bytes, i.e. $P_i = [b_{i,1}, \ldots, b_{i,ps}]$ where $b_{i,j} \in [0, 255]$ and ps is the page size.

2.3 Intradeduplication

As a first mean of compressing a memory dump, we leverage two observations. First, sandboxes usually have a small memory footprint, since they do not have a lot of software executing on them. Hence, the majority of the physical memory is unused, which usually means that the vast majority of the pages contain only 0-bytes. Second, some of the pages inside the memory dump besides the 0-byte pages are likely to appear multiple times. This is because of technical issues like the usage of different library versions or memory-mapped files.

To illustrate this, Figure 1 shows the number of occurrences of the 30 most-common pages of a 512 MiB memory dump of a Windows XP analysis system we created. The most common page is the page which contains only 0-bytes and occurs 91,698 times. While this is expected, also other, non-zero pages occur

Page Number	Distinct	Total
Same	23,695	118,685
Different	365	8,500
$\sum$	23,991/27,776	127,185/131,072

Table 1. A breakdown of the inter-deduplicatable pages of a Carbanak dump with respect to whether or not those pages have the same page number in both $\mathcal{R}$ and $\mathcal{D}$.

several times. For example, the second most common page also contains mostly 0-bytes with a few exceptions and occurs 3,173 times. In total, the 512 MiB memory dump consists of 131,072 pages ($131{,}072 \cdot 4$ KiB = 512 MiB) out of which 942 distinct pages occur more than once and the number of occurrences sum up to 97,663. This means that 97,663 out of 131,072 pages (74.51%) can be represented by storing only 942 (0.72%) pages plus some metadata overhead.

We call this way of compressing a memory dump *intra-deduplication*, as it deduplicates pages *within* a given memory dump. While intra-deduplication seems promising, we will see that this compression idea is inferior to the one we will describe next—given that common compression methods work similarly. However, it is very efficient and gives a basic understanding of the modus operandi of MemScrimper. In particular, it shows how we can leverage the structure of memory dumps, i.e., the concept of pages, to achieve a compression with simple means. We just need to read the memory dump page-wise, keep track of pages that occur more than once, and finally, write a data structure which leverages this deduplication idea (cf. Section 3.2 for more details).

2.4 Interdeduplication

We now describe *inter-deduplication*, which is a second, more evolved compression method implemented in MemScrimper. The idea of inter-deduplication is to compare a memory dump $\mathcal{D}$ that we want to compress with the help of a reference dump $\mathcal{R}$ of the analysis machine. That is, we parse the sequence of pages $[P_1, \ldots, P_n]$ of $\mathcal{D}$ and check for each P_i if it is present in the reference dump $\mathcal{R}$, i.e., we check if $P_i \in \mathcal{R}$ is true. If this is the case, it means that we do not have to store P_i. Instead, we only need to store referential information that links the deduplicated page from $\mathcal{D}$ to the correct page number of $\mathcal{R}$.

To illustrate the effectiveness of this idea, Table 1 shows how many pages of a memory dump resulting from an execution of a Carbanak APT malware can be deduplicated when compared against the reference dump. In total, 23,991 distinct pages occur more than once and the number of occurrences sum up to 127,185. This means that 127,185 out of 131,072 pages (97.03%) can be represented by storing only referential information. In fact, as we will see later in the implementation details in Section 3.3, we only need to store referential information if the deduplicated page has a different page number in $\mathcal{D}$ than it has in $\mathcal{R}$. The vast majority of deduplicatable pages ($118{,}685/127{,}185 \approx 93.32\%$) share the page number between $\mathcal{R}$ and $\mathcal{D}$, which means that we do not need to store a

single byte to represent those pages. We will only need to store referential information for the 8,500/127,185 (6.68%) pages which have a different page number in $\mathcal{R}$ than they do have in $\mathcal{D}$.

Combining Interdeduplication with Intradeduplication: In our example, inter-deduplication leaves 3,887 (2.97%) that cannot be deduplicated as they do not appear in $\mathcal{R}$. To further reduce the amount of information required to represent those pages one could apply the intra-deduplication technique between those 3,887 pages. In this example, 101 (2.60%) of the distinct pages occur more than once and the number of total occurrences sums up to 202. This means that intra-deduplication might only add a slight improvement, and as we will show in Section 4, might even reduce the compression ratio if we add standard compression utilities on top.

Differential Interdeduplication: So far we have deduplicated *identical* pages. We now search for *similar* pages, and if possible, only store differential information to a similar page. Such a differential view is particularly helpful if pages are only slightly modified when the malware executed. While a strict page comparison would interpret such slight changes as failure to deduplicate, a differential deduplication likely requires less space than saving the entire updated page. Thus, for each page P_i, we check if we can find a similar page $P'_j \in R$ and only store the diff $\delta(P_i, P'_j) = [b'_{j,k} \mid 1 \leq k \leq ps, P_i \ni b_{i,k} \neq b'_{j,k} \in P'_j]$, i.e., the byte-wise difference of both pages. The diff function δ captures the type of modifications we expect, i.e., patches which just replace a few bytes as opposed to more complex modifications which completely move data inside a page. Additionally, we expect the closest candidate to reside at the same page number in the reference dump. Hence, we let $j = i$, which also makes the candidate selection less costly. Since we need some metadata for storing a diff, i.e., the offsets of the bytes which will need to be patched, we only store a diff if the number of bytes plus the metadata overhead is smaller than the page size.

In our running Cabernak example, 3,811 out of 3,887 pages (98.04%) yield a diff (including overhead) which is smaller than the page size. On average, the size of the diff without overhead, i.e. $|\delta(P_i, P'_i)|$, is 891.39 bytes (median 317, std 1,099.4). The sum of all $\delta(P_i, P'_i)$ is 3,397,094 bytes, which is $3{,}811 \cdot 4{,}096 - 3{,}397{,}094 = 12{,}212{,}762 \approx 11.65$ MiB less than what would be needed to store those pages without our diffing idea (ignoring metadata overhead). To foreshadow a bit, however, the overhead of storing diffs is not negligible and would account for 527.21 KiB in this example.

After performing all of these steps, only a negligible fraction of pages, i.e., 76/131,072 (0.06%), need to be stored without any of the previous optimizations being applied to them. The remaining 130,996/131,072 pages (99.94%) could either be deduplicated page-wise, or a similar page was found in $\mathcal{R}$ such that storing the diff is less costly than storing the entire new page.

3 Implementation

We will now describe the implementation details of MemScrimper. We implemented a reference prototype in Python, which includes the design of a file format for memory dumps compressed with MemScrimper.

3.1 File Format Overview

The compressed memory dump files of MemScrimper consist of two components, the *header* and the *body*. The header contains metadata which is shared among the different compression methods, while the body contains method-specific metadata as well as the actual compressed memory dump. We first define the header, which contains the following information:

Magic Number A zero-terminated string (*str*) (currently "`MBCR`"), which aids file identification.

Method The name of the method (*str*) that the memory dump has been compressed with (e.g., "`intradedup`" for intra-deduplication)

Major Version Number A two-byte integer describing the version of MemScrimper, which can, e.g., be used to track changes to the header format.

Minor Version Number A two-byte integer version number that can be used to track changes to the compression method implementation.

Page Size The page size (in bytes, usually 4096) that has been assumed for the compression represented with a four-byte integer (4B).

Uncompressed Size The size of the uncompressed dump in bytes represented with an eight-byte integer (8B).

3.2 Intradeduplication

Intra-deduplication finds pages which are shared *within* the same memory dump. Technically, we read the memory dump page-wise and keep track of duplicate appearances. To reduce the memory footprint of pages that already appeared previously in the memory dump, we can store hashes instead of the entire page contents. The file format that we use for intra-deduplication memory dumps is depicted in Figure 2. The integer n stores the number of distinct pages used as reference during deduplication and corresponds to the number of page contents that appear in more than one page in the given dump. For each such page, we can deduplicate all other identical pages. After the number n, we write all these n reference pages next to each other in the file (ps is the page size in bytes).

Once we have stored all reference pages, we now also have to encode how these pages were referred to in the original memory dump. That is, we need to denote at which offset(s) each reference page was stored such that we can (during decompression) reconstruct the original file. A naïve solution would be storing a list of original page numbers for each reference page. Yet such lists waste space if we face large contiguous ranges where the same page occurs repeatedly, as for example in the case of 0-pages. To account for this, for each reference page, we

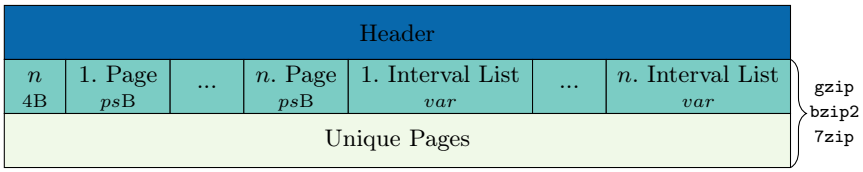

Fig. 2. File format of a memory dump, which was compressed with intra-deduplication.

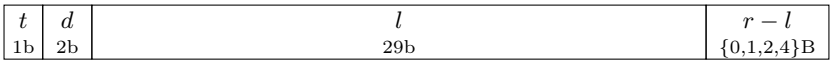

Fig. 3. Encoding of an interval $[l,r]$.

store a list of page number intervals. A list of page number intervals is an ordered sequence of intervals $I = [l,r]$, in which each interval represents the page numbers at which a deduplication page occurs. Figure 3 depicts how we encode an interval. First, we store the termination bit t, which indicates whether this interval is the last item of the contiguous interval list. This enables the decompressor to detect the end of an interval list without the need to store the length of the interval. After that we store a two bit number d, which describes the number of bytes that is required to store the size of the interval (the four possible states are mapped to an interval length size of 0, 1, 2 or 4 bytes). To encode the actual interval, we store the left side of the interval l, i.e., the page number at which the interval starts. Finally, we encode the size of the interval in number of pages (minus 1), i.e., $r - l$. We omit the size encoding if the interval is just one page (determined by d), such that we can store single-page intervals in just four bytes.

To parse an interval list, the decompressor would proceed as follows: First, t, d and l are read. Then, the size of the interval can be deduced depending on the value of d. If $d = 0$, the interval is $[l, l]$ and the $r - l$ field is omitted. For $d = 1$, the $r - l$ field is a single byte long and can indicate a page range of maximum 256 pages. For the maximum $d = 3$, the $r - l$ field is four bytes long and can indicate a size of up to 2^{32} pages. With this information, the decompressor can reconstruct the interval $[l, r]$. If $t = 0$, this process is repeated until the final entry that has $t = 1$ and thus terminates the list of page intervals.

The notion of intervals serves as a zero-overhead alternative to plainly storing page numbers. If a deduplicated page occurs only twice at non-consecutive page numbers, for example, we would represent this with two interval lists which would be 4-bytes each, which is the same number of bytes that we would require if we simply stored the page numbers directly. Since we use 29 bits to denote the left side of the interval l, the maximum number of pages that MemScrimper can handle with this method is $2^{29} - 1$, which roughly corresponds to 2 TiB memory dumps if we consider a page size of 4 KiB. Any changes to size constraints made in the file format can be reflected by the minor version field of the header.

After the interval lists, the unique pages, i.e., the pages which cannot be deduplicated, are written next to each other in the same order as they appear

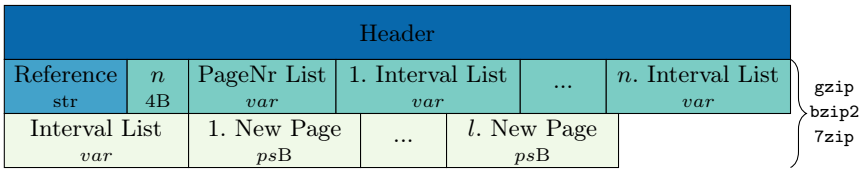

Fig. 4. File format of memory dump, which was compressed with inter-deduplication.

in the memory dump. We do not need to store any referential information for those pages. To reconstruct the memory dump, the decompressor would first parse all the pages and intervals. The page numbers which are not contained in those intervals are then chronologically mapped to those unique pages.

Finally, to further compress the inter-deduplicated memory dump, we can optionally add standard compression utilities like `gzip`, `bzip2` or `7zip` on top of this method. This is achieved by simply compressing the body of the file with those utilities, while the header is left untouched to give the decompressor the necessary metadata for reconstructing the memory dump.

3.3 Interdeduplication

To compress a memory dump $\mathcal{D}$ with inter-deduplication, we need to compare the pages of $\mathcal{D}$ against the pages of a reference memory dump $\mathcal{R}$. To achieve this, we first parse the pages of $\mathcal{R}$ and then iterate over all pages in $\mathcal{D}$ and check if a page occurs in $\mathcal{R}$. If so, we can inter-deduplicate this page. If the page number of such a page is identical in $\mathcal{R}$ and $\mathcal{D}$, we do not need to store any information about the page. Only if the page numbers differ, we need to store the respective page number in the compressed memory dump.

The file format of the inter-deduplication method is depicted in Figure 4. We first store the path of $\mathcal{R}$ so that the decompressor knows against which memory dump the compressed memory dump was compared. Similar to the intra-deduplication implementation, we again store the number of distinct reference pages n that are used as basis for (inter-)deduplication. Again, similarly, we store n page number interval lists which denote for each of the n reference pages where it occurs in $\mathcal{D}$. The missing bit of information, i.e., where the deduplication page occurs in $\mathcal{R}$ (its page number), is stored in a *page number list* that maps the n interval lists to n offsets in $\mathcal{R}$.

For space-efficiency reasons, the page number list is a sequence of n variable-sized integers. The first page number is absolute and all subsequent page numbers are relative to the page number given by the previous entry. If such a relative page number (or the first absolute page number) fits into 7 bits, the page number is encoded as a single-byte integer and the most significant bit is set to 1. This gives the decompressor the information that the page number can be recovered by only reading a single byte. Otherwise, the most significant bit is 0 and the page number is encoded with 4 bytes.

Finally, we need to store the new pages, i.e., pages which are not covered by inter-deduplication. We thus store a page number interval list that encodes the page numbers of all new pages, followed by the sequence of new pages. The number of page numbers encoded by the interval list is the same as the number of new pages, i.e., l in Figure 4, which tells the decompressor where to put each new page. If a page number is not covered by this interval list or the page number list of the deduplicated pages, then this page is identical to the page in $\mathcal{R}$ at the same page number. This is also why we need to store referential information for the new pages, which we did not need to do for intra-deduplication.

Combined Inter- and Intradeduplication The new pages in the basic inter-deduplication format are not deduplicated within themselves. That is, if a new page occurs multiple times, we still store that page redundantly multiple times. This can be fixed by applying intra-deduplication to those pages, similar as described before by storing an interval list for each of the l new pages. This idea is depicted in Figure 5, where we also need to store the integer l to tell the decompressor how many new pages exist, which was previously given implicitly by the single interval list for the new pages. However, we noticed in our experiments that intra-deduplication applied to the new pages adds little to no benefit, since the new pages usually do not occur multiple times. In fact, if we use standard utilities to compress the body, this combined inter- and intra-deduplication might result in worse compression ratios due to locality issues.

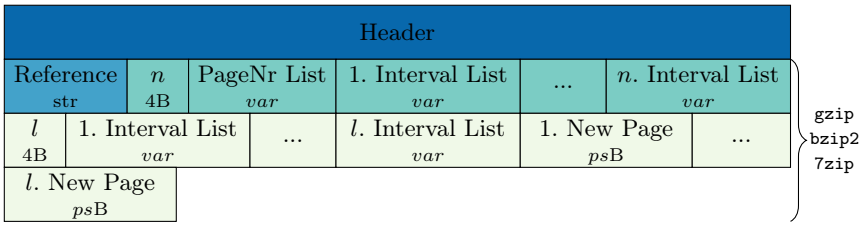

Fig. 5. File format of a memory dump, which was compressed with inter-deduplication and intra-deduplication applied to the new pages.

Differential Interdeduplication A more effective optimization is to look for *similar* pages in $\mathcal{R}$ and $\mathcal{D}$ and store only their differences. To do this, we compare each page P_i of $\mathcal{D}$ with the corresponding page P'_i of $\mathcal{R}$, which resides at the same page number i. We then compare each byte $b_{i,j} \in P_i$ with the corresponding byte $b'_{i,j} \in P'_i$ at the same offset and remember the byte if it is different. This yields the difference $\delta(P_i, P'_i) = [b_{i,k} \mid 1 \leq k \leq ps, P_i \ni b_{i,k} \neq b'_{i,k} \in P'_i]$.

To encode the difference such that we can use it later to restore the actual page P_i, we also need to store the offsets of the patched bytes. A simple solution for this problem would be to store pairs of offsets and bytes. While this

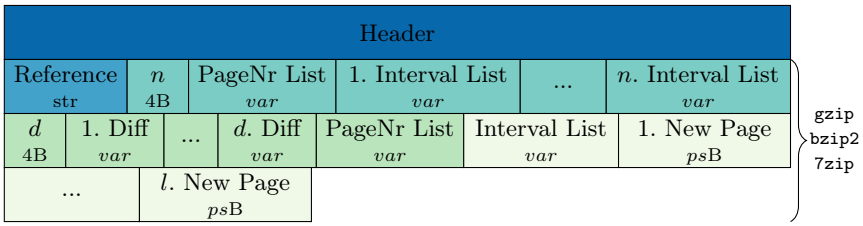

Fig. 6. File format of a compressed memory dump with inter-deduplication and diffing.

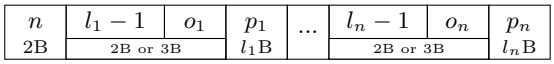

Fig. 7. Encoding of a diff $d = [(o_1, l_1, p_1), \ldots, (o_n, l_n, p_n)]$

would work, it would incur a significant overhead, as we would have overhead for each byte. Instead, we reshape the difference of a page to look as follows: $D = [(o_1, l_1, p_1), \ldots, (o_n, l_n, p_n)]$, i.e., a a sequence of triples (o, l, p) that each represents a patch. A patch is a sequence of bytes p of length l which needs to be applied at offset o to restore the original page content at a particular offset in the page. It is straight forward to compute D from δ by simply looking for long streaks, i.e., bytes $b_{i,k}$, which appear consecutively. Additionally, the offsets o_i are relative to the offset given by the previous entry plus the length of the previous entry, i.e., $o_{i-1} + l_{i-1}$, with the exception of o_1, which is absolute.

Figure 7 shows the diff encoding. An integer n stores the number of patches of the diff. This is followed by n patches, i.e., (o, l, p) triples. The offset o and the length l are packed into a single two-byte integer if they both fit into 7 bits each, otherwise they are packed into a 3-byte integer and the most significant bit of this integer is set to 1. To ensure that the length and the offset always fit in 3 bytes, we guarantee that no streak is longer than 2048 bytes and partition longer streaks. Since we store $l - 1$, as shown in Figure 7, we ensure that the decompressor can recover the correct value of l since $l - 1$ fits in 11 bits and therefore the most significant bit of the three-byte integer does not overlap.

Figure 6 shows how this diffing idea is incorporated into the file format. The file format is similar to the previous one with the only addition being that the interval list of deduplicated page numbers is followed by a list of d diffs. Again, we add a page number list to encode the position of the d pages that can be recovered by applying those diffs. The page number list and the diffs are ordered so that the nth diff corresponds to the nth page number of the page number list. To recover the original pages, the decompressor reads the page numbers from the page number list, read the corresponding pages from the reference dump and apply the corresponding diffs. Similar to before, intra-deduplication can optionally be applied to the new pages, which is depicted in Figure 8.

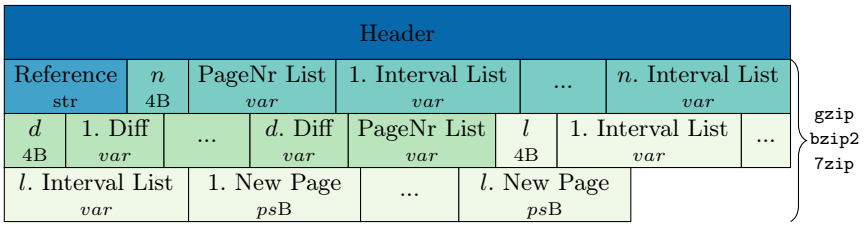

Fig. 8. File format of a compressed memory dump with inter-deduplication, diffing and intra-deduplication applied to the new pages.

4 Evaluation

In the following, we will now evaluate MemScrimper with respect to several aspects. First, we will evaluate the *data compression ratio*, i.e., the ratio between compressed size and uncompressed size of a memory dump and we will compare this ratio against the ratios of the standard compression utilities `gzip`, `bzip2` and `7zip`. For each of these utilities, we add a compression method to MemScrimper where the body contains the memory dump after compressing it with the respective utility. For each utility we used the strongest compression flags, e.g., `mx=9` for `7zip` or `-9` for `gzip`. Second, we will evaluate the compression time, i.e., how long it takes to compress and decompress a memory dump with MemScrimper.

As discussed in Section 3, there are several different methods that we will need to compare. First, there are the three standard compression methods `gzip`, `bzip2` and `7zip`. Then, we have the intra-deduplication method, for which we have an additional method if we add the above mentioned standard compression utilities, which gives us 4 additional methods. We will refer to this method and its variations with Intra^x where $x \in \{\texttt{gz}, \texttt{bz}, \texttt{7z}, -\}$, to denote the compression method that was used on top[1]. In the case of the inter-deduplication method, we have 4 different variations. The one which applies no intra-deduplication on the new pages, the one which does and for both of them an additional variation which applies diffing. On each of those four variations, we can again apply the above standard compression utilities as before, which results in 16 additional methods. We will refer to this method and its variations with Inter^x_y where $x \in \{\texttt{gz}, \texttt{bz}, \texttt{7z}, -\}$ and $y \subseteq \{\circlearrowleft, \delta\}$, i.e., $\circlearrowleft$ denotes intra-deduplication applied to the new pages and δ indicates that diffing was used[2]. In total, this gives us 23 methods, which we will need to consider.

[1] For the sake of brevity, we denote the absence of an additional compression layer with Intra instead of Intra$^-$.

[2] For the sake of brevity, we write Inter^x instead of Inter$^x_\varnothing$.

Windows XP (512 MiB)			
Method	`gzip`	`bzip2`	`7zip`
–	9.54	10.71	16.15
Intra	10.20	11.15	16.55
Inter	121.72	134.37	206.18
$\text{Inter}_{\circlearrowleft}$	117.41	121.04	195.12
Inter_{δ}	427.53	459.90	645.15
$\text{Inter}_{\circlearrowleft,\delta}$	446.60	469.65	644.75

Windows 7 (2 GiB)			
Method	`gzip`	`bzip2`	`7zip`
–	9.34	9.51	14.54
Intra	9.73	9.74	15.27
Inter	140.08	147.93	232.61
$\text{Inter}_{\circlearrowleft}$	132.15	128.19	216.55
Inter_{δ}	351.73	361.11	505.82
$\text{Inter}_{\circlearrowleft,\delta}$	358.91	363.49	503.14

Table 2. The average data compression ratio of the different methods (larger is better, best ratio is highlighted) grouped by the analysis machine the memory dump was taken from (Windows XP and Windows 7).

4.1 Experimental Setup

For evaluation we collected a data set of 236 labelled samples from 20 malware families in total (Foreign, Tedroo, Fareit, Ghost, Kelihos, Kuluoz, Nitol, NJRat, Nymaim, Virut, ZeusP2P, Kronos, Pushdo, Carbanak, LuminosityLink, SpyEye, Dridex, ISRStealer, Palevo, Tinba). We executed each sample for 2 minutes in a virtual machine (VM) starting from a snapshot using the VirtualBox hypervisor. Additionally, we ran an idle execution on the VM for 2 minutes as well to collect the reference dump $\mathcal{R}$. Using manual forensic analysis, we carefully verified that each sample became active and that it indeed belongs to the labelled family. By doing so, we ensure that MemScrimper will operate on actually infected memory dumps. If malware for example shows evasive behavior and will not start on the virtual machine, MemScrimper would operate on a memory dump where the malware has left little to no memory footprint, which would in turn skew our results as the resulting memory dump would be more similar to the reference memory dump $\mathcal{R}$. By verifying the family and the activity of the sample, we therefore simulate a worst case behavior where MemScrimper operates on memory dumps with a real malware footprint.

We performed this whole process for a Windows XP and a Window 7 VM where 512 MiB of memory was assigned to the former and 2 GiB of memory was assigned to the latter. We did so as we want to evaluate how MemScrimper scales for larger memory dumps.

4.2 Data Compression Ratio

The data compression ratio is defined as $\frac{u}{c}$ where u is the size of the uncompressed memory dump in bytes and c is the size of the compressed memory dump in bytes. To compute the compression ratio, we ran all methods of MemScrimper on all collected memory dumps and compared the sizes. The average compression ratio of each method is depicted in Table 2. A first (rather unsurprising) observation is that `7z` is superior to both `gz` and `bz2`, which is also why we will only consider `7z` for the remaining data compression ratio evaluation.

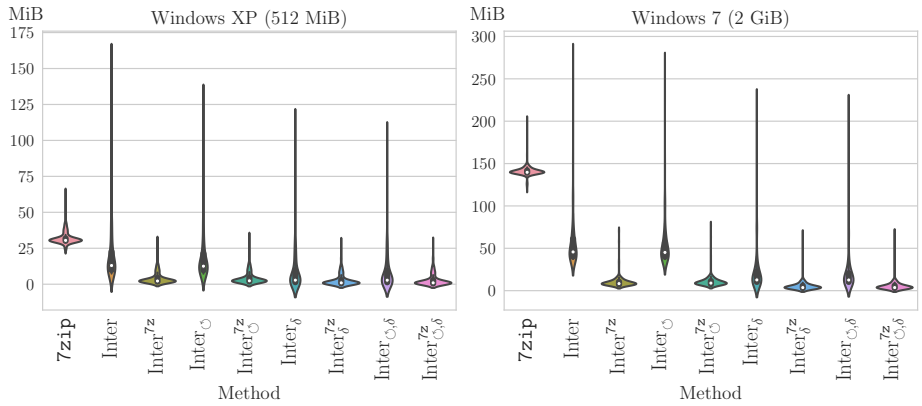

Fig. 9. A violin plot depicting the distribution of the actual file sizes of memory dumps compressed with MemScrimper (lower is better).

Another observation is that while Intra^{x} yields a better compression ratio than the individual compression methods alone, the improvement is not significant. The standalone `7z` method yields a compression ratio of 16.15 on the Windows XP memory dumps and 14.54 on the Windows 7 memory dumps, while Intra$^{\texttt{7z}}$ achieves a ratio of 16.55 (2.42% improvement) and 15.27 (5.02% improvement) respectively. The Inter$_{\delta}$ method on the other hand yields compression ratios as high as 645.15 (3894.74% improvement) in the case of Windows XP and 505.82 (3378.82% improvement) if we use `7z` as the inner compression. Another observation is the fact that the usage of diffing (δ) greatly improves the compression ratio. In the case of Inter$^{\texttt{7z}}$ and Inter$^{\texttt{7z}}_{\delta}$, it increases the compression ratio from 206.18 to 645.15 (212.91% improvement) in the Windows XP case and from 232.61 to 505.82 (117.45% improvement) in the Windows 7 case. The usage of intra-deduplication within the new pages (↻) adds little to no improvement and even more interesting, it might also decrease the compression ratio, which we can see if we compare Inter$^{\texttt{7z}}_{\delta}$ against Inter$^{\texttt{7z}}_{\circlearrowleft,\delta}$ for example or Inter$^{\texttt{7z}}$ against Inter$^{\texttt{7z}}_{\circlearrowleft}$. We suspect that the reason for this is that by intra-deduplicating the new pages, we remove locality information, which can be used by the `7z` utility to achieve a better compression.

The violin plot in Figure 9 depicts the distribution of the file sizes of the compressed memory dumps. We focus here only on `7z` and the relevant inter-deduplication variations as the previous discussion has shown that these are the most relevant candidates. The plot also contains the file sizes for the Inter$_{x}$ variations, i.e., without any inner compression applied – something we have omitted in Table 2. An interesting observation here is that these variants are already better than `7z` alone on average. For example, in the Windows 7 case, `7z` yields an average size of 141.12 MiB (median 139.72, std 6.65) whereas the Inter$_{\circlearrowleft,\delta}$ method yields an average size of 21.26 MiB (median 12.58, std 24.10). However, as shown in the violin plot, the methods without any inner compression applied to them suffer from outliers, which results in a high standard deviation. This standard deviation can be greatly reduced by applying `7z` as the inner

Compression

Windows XP (512 MiB)					Windows 7 (2 GiB)				
Method	–	`gzip`	`bzip2`	`7zip`	**Method**	–	`gzip`	`bzip2`	`7zip`
–	–	23.99	22.08	35.68	–	–	84.66	86.04	53.02
Intra	1.80	20.43	13.82	32.37	Intra	7.40	73.21	54.26	75.63
Inter	2.54	6.87	4.67	6.06	Inter	10.73	22.51	17.73	20.94
$\text{Inter}_{\circlearrowleft}$	2.45	6.57	4.03	6.16	$\text{Inter}_{\circlearrowleft}$	10.97	21.92	15.71	21.91
Inter_{δ}	8.34	8.78	9.11	9.82	Inter_{δ}	24.11	27.15	26.15	27.82
$\text{Inter}_{\circlearrowleft,\delta}$	8.16	8.66	9.15	9.86	$\text{Inter}_{\circlearrowleft,\delta}$	24.14	26.98	26.38	27.90

Decompression

Method	–	`gzip`	`bzip2`	`7zip`	**Method**	–	`gzip`	`bzip2`	`7zip`
–	–	2.35	6.19	2.69	–	–	8.99	24.66	10.86
Intra	0.52	1.41	4.90	2.60	Intra	1.63	5.23	19.87	10.41
Inter	1.12	1.27	1.82	1.46	Inter	4.74	5.18	6.68	5.07
$\text{Inter}_{\circlearrowleft}$	1.04	1.18	1.71	1.44	$\text{Inter}_{\circlearrowleft}$	4.36	4.86	6.28	5.14
Inter_{δ}	1.90	2.94	4.22	2.19	Inter_{δ}	5.94	8.03	10.71	6.36
$\text{Inter}_{\circlearrowleft,\delta}$	1.90	2.87	4.16	2.17	$\text{Inter}_{\circlearrowleft,\delta}$	5.92	8.00	10.63	6.37

Table 3. Average compression and decompression times of the different methods in seconds (lower is better, best is highlighted).

compression, which takes care of compressable data that we missed with our methodology. To put this into perspective consider the Inter_{δ} method, which has a standard deviation of 25.34 which can be reduced to 6.27 by applying `7z` as the inner compression, which gives the $\text{Inter}_{\delta}^{\texttt{7z}}$ method. The best performing method in both cases, i.e. $\text{Inter}_{\delta}^{\texttt{7z}}$, yields an average size of 2.81 MiB (median 0.94, std 4.26) in the case of Windows XP and an average size of 5.57 MiB (median 3.63, std 6.27) in the case of Windows 7, which is a large improvement over `7z` alone (WinXP: 32.21 avg, 30.49 med, 4.61 std, Win7: 141.12 avg, 139.72 med, 6.66 std).

4.3 Compression and Decompression Time

To further assess the performance of MemScrimper, we measured how long it takes to compress and to decompress a memory dump. To this end, Table 3 depicts the average compression and decompression time of the different methods in seconds. To collect this data, we randomly sampled 4 memory dumps per family (i.e., a pair of a Windows XP and a Windows 7 memory dump for 2 samples per family), and ran all methods on the resulting memory dumps. We had to sample, as a precise compression and decompression time analysis required us to run the methods sequentially. Parallel processing would introduce variance of our results due to concurrent disk reads/writes, caching issues and scheduling and load problems. Since running all methods on all memory dumps sequentially would take too long, we decided to follow this sampling approach, which still gives us a reasonable number of $20 \cdot 4 = 80$ memory dumps per method.

Table 3 shows that the fastest compression method by far is Intra with average compression times of 1.8 seconds for Windows XP and 7.4 seconds for Windows 7. This is followed by Inter with compression times of 2.54 seconds for

Windows XP and 10.73 seconds for Windows 7. Although Intra and Inter are conceptually quite similar in that they only look for deduplicatable pages, Inter is slightly slower, since it has to process two memory dumps, i.e., the one we want to compress $\mathcal{D}$ and the reference memory dump $\mathcal{R}$. In a real world deployment of MemScrimper one could pre-process the reference memory dump $\mathcal{R}$ once and put the result (i.e., a hashmap of the pages) in memory to eliminate overhead. Due to the bad data compression ratio, Intra also gets significantly slower as soon as we add inner compression to the method, which becomes evident if we compare Intra with Intra$^{\texttt{7z}}$ where we observe a slowdown of 922.03% in the Windows 7 case—the costly inner compression has to work on large data, which is inevitably slow. Conversely, if we consider Inter$_{\delta}$, i.e., the method with the best data compression ratio, we can see that the overhead of adding inner compression is less significant with the largest being the difference to Inter$_{\delta}^{\texttt{7z}}$ with 3.71 seconds (15.39%) in the case of Windows 7. Another observation is that intra-deduplicating the new pages adds little to no overhead, since the difference between Inter and Inter$_{\circlearrowleft}$ and between Inter$_{\delta}$ Inter$_{\circlearrowleft,\delta}$ can be considered negligible. The usage of diffing on the other hand incurs a significant overhead of up to 124.70% if we compare Inter against Inter$_{\delta}$ in the case of Windows 7. This can be explained by the fact that diffing adds more complexity to the compression, as it has to compare each page that cannot be directly deduplicated with the corresponding page in the reference memory dump $\mathcal{R}$. As we have seen in the compression ratio analysis, this turns out to be very effective, which means that many diffs have to be created and serialized – all of which contributes to additional complexity.

If we compare the numbers of Windows XP to the numbers of Windows 7, we can see that the methods scale linearly with the size of the memory dump, which confirms our expectations given the way we have implemented those methods (cf. Section 3). All of the Inter$_{y}^{x}$ methods perform better than any of the standalone compression methods in both compression and decompression. In particular, Inter$_{\delta}^{\texttt{7z}}$, i.e., the method with the best data compression ratio, has an average compression and decompression time of 9.82/2.19 seconds (Windows XP) and 27.82/6.36 seconds (Windows 7) as opposed to `7zip` alone, which yields average compression and decompression times of 35.68/2.69 seconds (Windows XP, 72.48%/18.59% improvement) and 53.02/10.86 (Windows 7, 47.53%/41.44% improvement). This also shows that decompression is much faster than compression, which also stems from the fact that we have written the decompressed memory dumps to memory in our decompression runs as opposed to writing them to disk. We did so, because we envision a workflow where a memory dump is compressed once for storage and then decompressed multiple times over time for historical analysis, e.g., consider a case where a sandbox operator retrieves new signatures and wants to apply them on archived memory dumps. In our evaluation, the Inter$_{y}^{x}$ methods also benefited from the fact that we put the reference memory dump $\mathcal{R}$ for both Windows XP and Windows 7 in memory rather than on disk, which we believe is a realistic option for sandbox vendors (cf. Section 5.1 for a detailed discussion on this matter).

Finally, note that while the standalone compression tools `gzip`, `bzip2` and `7zip` are all written in C or C++ and are heavily optimized, our prototype of MemScrimper is written in Python and can therefore not achieve the same level of optimization. It is therefore likely that an optimized implementation in a non-interpreted language would further amplify the performance benefit.

4.4 Soundness and Efficiency of Analyses on Compressed Dumps

In the previous evaluation step, we have measured how long it takes to decompress an entire memory dump. In fact, however, for several analyses such as signature matching it is not actually required to work on the entire dump, but just on the memory parts that were changed by the malware. To demonstrate this, we collected a total of 17 YARA [1] signatures for 17 out of 20 malware families of our data set using Malpedia [18] and various other online resources. We then matched all those YARA signatures against the previously sampled 80 memory dumps and verified that 24/80 memory dumps matched the correct signature for the given family. Since our methodology only stores memory pages in plain if they cannot be deduplicated or stored differentially, it is reasonable to assume that only these pages contain the relevant memory footprint of the given malware. To verify this, we matched all the YARA signatures against the memory dumps of the Intra and Inter_x methods without inner compression. We discovered that the compressed memory dumps matched all YARA rules perfectly, i.e., the matching yielded the same results as the completely uncompressed ones.

Performance-wise, the matching was also faster than on uncompressed memory dumps. While the uncompressed ones took 6.6 seconds on average to match all signatures, Intra yielded an average matching time of 1.57 seconds, Inter took 0.14 seconds followed by $\text{Inter}_{\circlearrowleft}$ (0.14 seconds). The best results were yielded by the Inter_{δ} method with 0.064 seconds and the $\text{Inter}_{\delta,\circlearrowleft}$ method with 0.067 seconds on average. These results nicely reflect the data compression ratio, i.e., the better the compression, the smaller the file size, the faster the matching. Even if we consider the overhead of removing the inner compression, the matching is still faster than on raw uncompressed memory dumps. Consider, for example, the $\text{Inter}_{\delta}^{\texttt{7z}}$ method in the Windows 7 case (i.e., the worst case) where `7zip` adds compression time overhead of $27.82 - 24.11 = 3.71$ seconds (cf. Table 3) in which case the matching is still $6.6 - (0.064 + 3.71) = 2.826$ seconds (43.36%) faster.

5 Discussion

In this secion we briefly discuss some aspects of MemScrimper regarding its limitations, use cases, deployment and future work.

5.1 Use Case & Limitations

MemScrimper is not a general purpose compression tool and is primarily meant to compress memory dumps of snapshot-based sandboxes. The underlying as-

sumption of the inter-deduplication methods is that dumps share a common "predecessor" which contains a substantial amount of similar data, i.e., the reference memory dump $\mathcal{R}$. In other settings, intra-deduplication is still applicable, which does however not yield the same compression ratio as inter-deduplication.

Furthermore, we assume that the malware execution starts from a well-defined snapshot. This snapshot was taken with great care to ensure that the system was in a stable idling state to limit the forensic noise that subsequent malware executions would create. This means that we verified no background process was executing and disabled services that had a large memory or performance footprint, as this would hinder differential analysis and would ultimately increase the size of the resulting memory dumps. We did not evaluate how MemScrimper would perform in a setting where the snapshot was not taken at such a stable point, e.g., when the snapshot would be created during the boot process.

One could question whether it is practical that MemScrimper keeps the reference dump in memory to speed up inter-deduplication. If the number of analysis environment and their memory footprint grows, sandbox operators might not have sufficient memory to store all dumps. However, note that (i) the number of environments and their assigned memory is usually small, (ii) intra-deduplication of the reference dump can be applied, and (iii) hashing the pages (except for diffing methods) significantly reduces the footprint. Therefore, maintaining reference dumps on the heap should always be feasible. Even if not, disk caches could partially mitigate this problem and maintain the substantial performance improvement compared to standard compression utilities.

5.2 Deployment

Adding MemScrimper to existing infrastructures should not pose a major challenge. At its core, MemScrimper can be considered a black box which receives a memory dump as input and yields a compressed memory dump as output, which can easily be integrated into an existing pipeline. The only manual effort which needs to be done is to take a reference memory dump per analysis machine and per snapshot and optionally putting this reference memory dump in memory to speed up the inter-deduplication process. Additionally, fine-tuning the snapshot as described above to ensure an optimal data compression ratio might be required. However, this is a one-time effort per analysis machine and per snapshot, after which MemScrimper operates completely automatic. To foster a wide-spread deployment of MemScrimper in sandbox environments, we publish the source code of our reference implementation[3].

5.3 Future Work

We have left open a few extensions open for future work. First, in our current inter-deduplication implementation, we compare against a single referene dump. However, it might be beneficial to also compare against other memory dumps

[3] Available at `https://github.com/mbrengel/memscrimper`

in general, as similar malware instances will likely create similar (or identical) memory changes. Adding this to MemScrimper is simple and promising to gain more space, yet it might incur a performance penalty.

Second, the presented methodology is *lossless* and does not discard any information from memory dumps. Depending on the actual use case, such as signature matching, it might be sensible to disregard small changes to memory pages (e.g., changes of pointers or small data items) in a lossy compression method. Lossy compression has the potential to even further reduce memory footprint, would however not allow the analyst to entirely restore the original memory dump.

6 Related Work

To the best of our knowledge, we are the first to leverage the similarity of malware sandbox memory dumps for compression. While the concept of data deduplication is well-known and extensively studied [3, 8, 21, 11], we do not know of any prior work which applied similar methodologies in the context of sandbox malware analysis. The closest work we found in this area is given by Park et al. [15], who discuss fast and space-efficient virtual machine checkpointing. The authors propose a deduplication mechanism for shared pages between the memory of a virtual machine and its page cache on disk, which is different from our approach and also not as effective.

We have introduced several new deduplication aspects, including (i) a file format for compressed memory dumps, (ii) the possibility to compare memory dumps against reference snapshots, and (iii) differential deduplication. In the malware context, the closest works to ours study the differential behavior (e.g., system activity) of malware on multiple sandboxes to discover evasive malware [12, 4]. While the objective of these works is not to compress data, and both do not consider memory dumps, they also extract an essential footprint of the malware by considering the differences in behavioral profiles—similar to how we only store the differences introduced by the malware in the forensic sense.

Despite the lack of otherwise related work in this area, we argue that our results are highly relevant. Apart from the primary use case of our work—saving memory dumps time- and space-efficiently—it also allows for other interesting insights. By only storing the forensic differences introduced by a malware, we enable a more targeted and more efficient analysis, which can only focus on those differences. Forensic analysis techniques such as digital forensic text string searching as proposed by Beebe et al. [7], or data structure content reverse engineering such as proposed by Saltaformaggio et al. [19], can significantly benefit from our approach. For example, after inter-deduplication, these techniques only have to consider the new pages, i.e., the ones which could not be (differentially) deduplicated—similar to how we matched YARA signatures more efficiently.

7 Conclusion

MemScrimperis a novel methodology for compressing malware sandbox memory dumps by exploiting their snapshot mechanism and the resulting similarity of the memory dumps. MemScrimper achieves data compression ratios which are one order of magnitude better than the ones yielded by standalone compression utilities such as `7zip`, while at the same time significantly improving their performance. We believe that MemScrimper is a promising addition to existing malware sandbox analysis infrastructures as it is easy to deploy and enables a longer storage time and a more viable longitudinal analysis of memory dumps.

Acknowledgements. This work was supported by the European Union's Horizon 2020 research and innovation programme, RAMSES, under grant agreement No. 700326 and by the European Union's Horizon 2020 research and innovation programme, SISSDEN, under grant agreement No. 700176.

References

1. YARA - The pattern matching swiss knife for malware researchers. `https://virustotal.github.io/yara/`, Accessed: 2018-02-27
2. Malware Statistics & Trends Report | AV-TEST. `https://www.av-test.org/en/statistics/malware/` (2018), Accessed: 2018-02-27
3. Aronovich, L., Asher, R., Bachmat, E., Bitner, H., Hirsch, M., Klein, S.T.: The Design of a Similarity Based Deduplication System. In: Proc. of the The Israeli Experimental Systems Conference (SYSTOR) (2009)
4. Balzarotti, D., Cova, M., Karlberger, C., Christopher, K., Kirda, E., Vigna, G.: Efficient Detection of Split Personalities in Malware. In: Proc. of the Annual Network and Distributed System Security Symposium (NDSS) (2010)
5. Bayer, U., Comparetti, P.M., Hlauschek, C., Kruegel, C., Kirda, E.: Scalable, Behavior-Based Malware Clustering. In: Proc. of the Annual Network and Distributed System Security Symposium (NDSS) (2009)
6. Bayer, U., Kirda, E., Kruegel, C.: Improving the Efficiency of Dynamic Malware Analysis. In: Proc. of the ACM Symposium on Applied Computing (SAC) (2010)
7. Beebe, N.L., Clark, J.G.: Digital Forensic Text String Searching: Improving Information Retrieval Effectiveness by Thematically Clustering Search Results. In: Proc. of the The Digital Forensic Research Conference (DFRWS) (2007)
8. Bhagwat, D., Eshghi, K., Long, D., Lillibridge, M.: Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup. In: Proc. of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS) (2009)
9. Jacob, G., Comparetti, P.M., Neugschwandtner, M., Kruegel, C., Vigna, G.: A Static, Packer-Agnostic Filter to Detect Similar Malware Samples. In: Proc. of Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) (2013)
10. Jang, J., Brumley, D., Venkataraman, S.: BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In: Proc. of the Conference on Computer and Communications Security (CCS) (2011)

11. Jayaram, K.R., Peng, C., Zhang, Z., Kim, M., Chen, H., Lei, H.: An Empirical Analysis of Similarity in Virtual Machine Images. In: Proc. of the International Middleware Conference (MIDDLEWARE) (2011)
12. Kirat, D., Vigna, G., Kruegel, C.: BareCloud: Bare-metal Analysis-based Evasive Malware Detection. In: Proc. of the USENIX Security Symposium (2014)
13. Kolbitsch, C., Comparetti, P.M., Kruegel, C., Kirda, E., Zhou, X., Wang, X.: Effective and Efficient Malware Detection at the End Host. In: Proc. of the USENIX Security Symposium (2009)
14. Neugschwandtner, M., Comparetti, P.M., Jacob, G., Kruegel, C.: ForeCast - Skimming off the Malware Cream. In: Proc. of the Annual Computer Security Applications Conference (ACSAC) (2011)
15. Park, E., Egger, B., Lee, J.: Fast and Space-Efficient Virtual Machine Checkpointing. In: Proc. of the International Conference on Virtual Execution Environments (VEE) (2011)
16. Plank, J.S., Beck, M., Kingsley, G., Li, K.: Libckpt: Transparent Checkpointing under Unix. In: Proceedings of the USENIX Technical Conference (TCON) (1995)
17. Plank, J.S., Chen, Y., Li, K., Beck, M., Kingsley, G.: Memory Exclusion: Optimizing the Performance of Checkpointing Systems. Software: Practice and Experience (1999)
18. Plohmann, D., Clauß, M., Enders, S., Padilla, E.: Malpedia: A Collaborative Effort to Inventorize the Malware Landscape. In: Proc. of Botconf (2017)
19. Saltaformaggio, B., Gu, Z., Zhang, X., Xu, D.: DSCRETE: Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse. In: Proc. of the USENIX Security Symposium (2017)
20. h T. Meyer, D., Aggarwal, G., Cully, B., Lefebvre, G., hael J. Feeley, M., hinson, N.C.H., Warfield, A.: Parallax: Virtual Disks for Virtual Machines. In: Proc. of the European Conference on Computer Systems (EuroSys). Association for Computing Machinery (ACM) (2008)
21. Xia, W., Jiang, H., Feng, D., Hua, Y.: SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. In: Proc. of the USENIX Annual Technical Conference (USENIX ATC) (2011)
22. Yokoyama, A., Ishii, K., Tanabe, R., Papa, Y., Yoshioka, K., Matsumoto, T., Kasama, T., Inoue, D., Brengel, M., Backes, M., Rossow, C.: SandPrint Fingerprinting Malware Sandboxes to Provide Intelligence for Sandbox Evasion. In: Proc. of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID) (2016)