THE JOURNAL OF
DIGITAL FORENSICS,
SECURITY AND LAW

Journal of Digital Forensics,
Security and Law

Volume 12 | Number 1 Article 9

3-31-2017

Compression of Virtual-Machine Memory in Dynamic Malware Analysis

James E. Fowler Ph.D.
Mississippi State University, fowler@ece.msstate.edu

Recommended Citation

Fowler, James E. Ph.D. (2017) "Compression of Virtual-Machine Memory in Dynamic Malware Analysis," *Journal of Digital Forensics, Security and Law*: Vol. 12 : No. 1 , Article 9.
DOI: https://doi.org/10.15394/jdfsl.2017.1437
Available at: https://commons.erau.edu/jdfsl/vol12/iss1/9

COMPRESSION OF VIRTUAL-MACHINE MEMORY IN DYNAMIC MALWARE ANALYSIS

James E. Fowler
Department of Electrical and Computer Engineering
Distributed Analytics and Security Institute (DASI)
Mississippi State University, Starkville, Mississippi, USA
fowler@ece.msstate.edu

ABSTRACT

Lossless compression of memory dumps from virtual machines that run malware samples is considered with the goal of significantly reducing archival costs in dynamic-malware-analysis applications. Given that, in such dynamic-analysis scenarios, malware samples are typically run in virtual machines just long enough to activate any self-decryption or other detection-avoidance maneuvers, the virtual-machine memory typically changes little from that of the baseline state, with the difference being attributable in large degree to the loading of additional executables and libraries. Consequently, delta coding is proposed to compress the current virtual-machine memory dump by coding its differences with respect to a predicted memory image formed by loading the same executables and libraries into the baseline memory. Experimental results reveal a significant improvement in compression efficiency as compared to straightforward delta encoding without such predictive executable/library loading.

Keywords: compression, malware analysis, virtual machine, delta coding

1. INTRODUCTION

Malware—malicious computer code of all types, including viruses, worms, bots, and trojans—is an ever-increasing threat to personal, corporate, and government computing systems alike. Particularly in the corporate and government sectors, the attribution of malware—including the identification of the authorship of malware as well as potentially the malefactor responsible for an attack—is of growing interest. Such malware attribution is often enabled by the fact that malware authors build on the work of others through the use of generators, libraries, and borrowed code. Determining malware *phylogeny*—the evolutionary history of and the derivative relations between malware—is consequently an endeavor of increasing importance. In some cases, it may be possible to simply analyze the source code or binary executable program of a malware sample; however, such *static analysis* is easily defeated by more sophisticated code that actively avoids detection. Such malware often employs self-modifying code (Egele, Scholte, Kirda, & Kruegel, 2012), as in the case of an encrypted malware file that self-decrypts upon execution in memory. Consequently, there is a growing focus on the *dynamic analysis* of malware which involves executing a malware sample and determining the actions it takes after some period of operation (Egele et al., 2012). In most cases, such dynamic analysis occurs in a virtual machine, or "sandbox," in order to confine the malware to an environment in which it can do no harm to real systems (Farmer & Venema, 2005).

In sandbox-driven dynamic analysis of malware, a virtual machine is typically run starting from some known, malware-free baseline state. The malware is injected into the virtual machine, and the machine is allowed to run for some period of time during which the malware presumably activates. The machine is then suspended, and the current machine memory is dumped to disk. The process may then be repeated for other malware samples, each time starting from the baseline state. Subsequent analysis procedures may then attempt to identify, classify, or otherwise analyze the malware based on the dumped memory image.

Stored in raw form on the disk, the dumped memory file is the same size as the virtual-machine memory; for virtual machines running modern operating systems, such memory would likely be no less than 512 MB but could be up to several GBs. If the corresponding memory dumps are to be retained for repeated analysis—as is likely to be required in order to determine a phylogeny for a large database of malware samples—lossless compression of the memory dumps is necessary to prevent explosive disk usage. For example, the VirusShare project[1] maintains a database of over 19 million malware samples; running these in a virtual machine with 512 MB of memory would require of 9 petabytes (PB) of storage to retain the memory dumps.

In this paper, we develop a scheme for the lossless compression of memory dumps resulting from the repeated execution of malware samples in a virtual-machine sandbox. Rather than compress each memory dump individually, we capitalize on the fact that memory dumps stem from a known baseline virtual-machine state and code with respect to this baseline memory. Additionally, to further improve compression efficiency, we exploit the fact that a significant portion of the difference between the baseline memory and that of the currently running machine is the result of the loading of known executable programs and shared libraries. Experimental results on a collection of virtual-machine memory dumps demonstrate a significant improvement over the straightforward compression of each memory dump independently. We detail our proposed compression scheme in the remainder of the text.

2. BACKGROUND

Any number of generic lossless compression algorithms could be applied to a virtual-machine memory-dump file to significantly reduce its size. Some obvious choices include algorithms from the Lempel-Ziv (Ziv & Lempel, 1977, 1978) and Burrows-Wheeler (Burrows & Wheeler, 1994) families of algorithms, as implemented by programs such as `gzip` (LZ77) and `bzip2` (Burrows-Wheeler). However, given the short amount of time that the virtual machine has typically been run in a dynamic-malware-analysis scenario—effectively just enough time for the malware under analysis to activate itself (including any self-extraction, self-decrypting, or self-decompression)—it is likely that memory has changed little from the baseline state. While there may have been a few new processes started and a few libraries loaded—along with corresponding memory allocations and data initializations—overall, the dumped memory file will likely have most memory locations unchanged from the baseline machine's memory. Indeed, Fig. 1 depicts a map of all memory locations that have changed when an example malware sample is run in a 512-MB virtual machine; specifically, we see that only 15% of the memory has changed, although we do observe that the differences between the current and baseline memories are widely distributed throughout the entire memory space. In such a situation, *delta encoding*—the compression of the differences between two files—is likely to significantly outperform any single-file compression approach.

Although there have been a number of lossless delta-encoding algorithms proposed in the past, perhaps those that are in the most widespread use are based on the VCDIFF (Korn, MacDonald, Mogul, & Vo, 2002) standard. Effectively, the VCDIFF standard prescribes delta encoding using an LZ77 variant in which the reference (or "source" dataset), which is available to both the encoder and the decoder, prepends the dataset to be encoded (the "target" dataset), such that LZ77 string matching can reference into the source dataset. More specifically, the target dataset is partitioned into non-overlapping target windows, and each target window is encoded by prepending a source window and performing LZ77-style string matching on the concatenated string starting from the beginning of the target dataset. The source window can come from either the source dataset or earlier in the target dataset; since the VCDIFF standard specifies only a file format, specific methodology for string matching and window selection are left to the encoder implementation to determine. Here, we focus on the open-source `xdelta3`[2] implementation of VCDIFF.

The `xdelta3` encoder relies on a substantial degree of similarity between the source and target datasets in order to outperform the LZ77 compression of the target dataset alone. As is evident from Fig. 1, we expect that, in our malware-analysis application, `xdelta3` encoding of a virtual-machine memory dump using the baseline machine's memory as the source dataset will result in a compressed file significantly smaller than that of `gzip` applied directly to the memory dump by itself. Indeed, for the specific dataset considered in Fig. 1, `gzip` produces a compressed file of size 139 MB, while `xdelta3` yields a 25-MB file. In the next section, we consider steps that may be taken to improve the performance of `xdelta3` in our malware-analysis application even further.

3. PROPOSED APPROACH

Some of the differences between the memory dump after a period of malware execution and the starting baseline memory state can be attributed to data

[1] `http://virusshare.com/`

[2] `http://xdelta.org/`

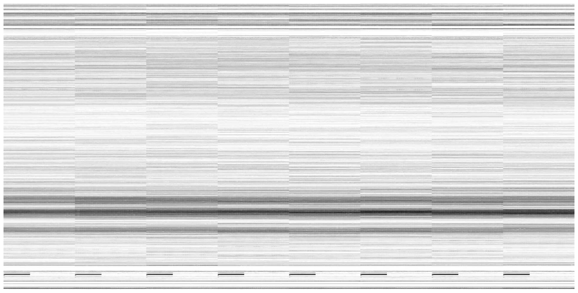

Figure 1. Map of differences between the baseline memory map and the current memory map after malware execution (baseline virtual machine #41, malware dataset #139992); number of bytes of difference = 80,661,129 (15.0% of the 512-MB memory). White = byte unchanged from baseline; black = byte different from baseline.

that was effectively created by the various processes, including the malware, running on the system. However, some of the memory differences are due to the loading of additional executable and shared-library files into the system. On Windows-based systems, such executable programs (EXEs) and dynamic-link libraries (DLLs) are stored in the Portable Executable (PE) format (Microsoft, 2013), a modification of Unix's Common Object File Format (COFF). We can make the memory of the baseline machine more closely resemble that of the currently running machine—thereby increasing the efficiency of `xdelta3` coding—by simulating the loading of these PEs into the baseline memory, a process that can be done identically in both the encoder and the decoder.

Specifically, the open-source tool for memory forensics, `volatility`[3] (Ligh, Case, Levy, & Walters, 2014), is used to determine the running EXEs and loaded DLLs in both the baseline and current memory dumps. Parsing the process and library lists produced by `volatility`, an encoder can determine which programs and libraries are new to the current machine memory with respect to the baseline, load these new PEs into the baseline memory, and finally use this updated memory as the source for `xdelta3` coding of the current memory dump. Detailed operation of the resulting encoder and decoder is described below.

3.1 Encoder

The encoder compresses the memory dump (the *current memory*) from the currently running virtual machine using delta encoding with respect to the *predicted memory*, the latter of which is produced by loading *new PEs* into the baseline virtual machine's memory dump (the *baseline memory*). The *new PEs* are those that are in the current memory but not in the baseline memory. The new PEs are loaded from the *virtual disk* which is the virtual hard drive shared by both the baseline and current virtual machines; the loading is accomplished by copying virtual-memory *pages* from the PE file into physical-memory pages in the baseline memory. More specifically, the encoder follows the following steps:

1. Run `volatility` commands `pslist` and `dlllist` on the baseline memory to determine lists of baseline EXEs and DLLs, respectively.
2. Run `volatility` commands `pslist` and `dlllist` on the current memory to determine lists of current EXEs and DLLs, respectively.
3. Parse the EXE/DLL lists to determine the new PEs that are in the current memory but not in the baseline memory.
4. To produce the predicted memory from the baseline memory, for each new PE (EXE or DLL) do:
   (a) Determine the ID of the process corresponding to the new PE (for a DLL, this is the process into which the DLL has been loaded; for an EXE, it is the process assigned to the EXE itself) along with the base address where the new PE is loaded into the virtual-memory address space of the corresponding process.
   (b) For the process corresponding to the new PE, run the `volatility` command `memmap` on the current memory to extract the virtual-to-physical memory map of the process.
   (c) Copy the new PE from its corresponding file on the virtual disk into the baseline memory; specifically, for each virtual-memory page in the PE file:
      i. If the page is resident in the current memory, copy the page from the PE file to the baseline memory using the virtual-to-physical mapping retrieved in Step 4(b).
      ii. Record the source page location in the PE file, the destination page location in physical memory, and the page length (the *page copy* information).
5. Output header information, including pathnames of new PEs to load and a list of all page copies for each PE.

[3] `http://www.volatilityfoundation.org/`

6. Perform `xdelta3` coding using the current memory as the target and the predicted memory as the source.

In Step 4(c), we assume that the virtual disk used for both the baseline as well as the currently running virtual machine is available to the encoder so that it can access the PEs to perform the page copies. Normally, this virtual disk will be a file stored at some known location alongside the baseline-memory and current-memory dumps. For the VMware virtual machines used here, the encoder employs the `vmware-mount` command to mount the VMDK-format virtual disk via a loopback device, permitting the encoder to read PE files from the Windows 7 installation resident on the virtual disk.

3.2 Decoder

The decoder produces the same predicted memory as used by the encoder by loading the new PEs into the baseline memory. While the decoder has access to the baseline memory, it does not know the current memory, or, consequently, the virtual-to-physical map of the processes corresponding to the new PEs. Therefore, to duplicate the page copies that the encoder used to produce the predicted memory, the decoder relies on the list of page copies stored in the header of the compressed file. Like the encoder, the decoder loads the new PEs from the virtual-disk file stored alongside the baseline memory. The specific process is as follows:

1. Read the header from the compressed file.
2. To produce the predicted memory from the baseline memory, for each new PE (EXE or DLL) do:
   (a) Copy the new PE from its corresponding file on the virtual disk into the baseline memory; specifically, for each page copy listed in the header, do:
      i. Copy the corresponding page in the PE file into the designated location in the baseline memory.
3. Perform `xdelta3` decoding of the current memory (the target) using the predicted memory as the source.

3.3 Implementation

Both the encoder and decoder of the proposed approach, which we call VMMZ, are implemented primarily in C with a small portion written in Perl to handle parsing of output from `volatility`. In addition to dependence on `volatility` and `xdelta3`, VMMZ is built on QccPack[4] (Fowler, 2000).

4. EXPERIMENTAL RESULTS

Our test dataset consists of 14 different baseline virtual machines, each possessing 512 MB of RAM and installed with a 32-bit version of Windows 7. Using the `cuckoo` malware-analysis sandbox[5], each virtual machine was run for approximately 4 minutes with a malware sample injected into the virtual machine. Afterwards, the virtual-machine memory was dumped as a 512-MB file to disk for malware analysis at some subsequent time. Our test dataset consists of a total of 67 memory dumps corresponding to 67 different malware samples, with between 3 and 8 different samples being run in each of the 14 baseline machines. Total storage required for the resulting 67 memory dumps is 33.5 GB.

Table 1 tabulates the results of various compression approaches applied to these memory dumps. The memory dumps are given a 6-digit number ("File #") corresponding to which malware sample was run in the virtual machine; Table 1 also indicates the number ("Base #") of the corresponding baseline machine which served as the starting point for the malware execution.

Table 1 indicates the sizes of files output by various compression algorithms applied to the virtual-machine memory dumps. In these results, `gzip` is simply applied directly to the memory dump in question, while the proposed approach (VMMZ) and `xdelta3`, both being delta encoders, compress each memory dump with respect to its corresponding baseline virtual-machine memory. Table 1 also indicates the average compressed-file size across all 67 memory dumps as well as execution times for both encoding and decoding. We see that, while the proposed VMMZ approach is slower, it significantly outperforms the other two compressors in terms of compression: at 34 MB, the average file size for VMMZ is approximately 20% smaller than that of `xdelta3` (42 MB on average), and 79% smaller than that of `gzip` (163 MB on average).

5. RELOCATABLE CODE AND FIXUPS

Typically, PE files are divided into multiple sections, some of which have special meanings that are recognized by linkers and loaders (Microsoft, 2013). By convention, these special sections are designated by known section names in the PE header, for example, the `.data` section (initialized data), the `.rdata` section (read-only initialized data), the `.text` section

[4] `http://qccpack.sourceforge.net/`

[5] `http://cuckoosandbox.org/`

Table 1. Compressed-file sizes and execution times for the proposed algorithm (VMMZ) as well as `xdelta3` (XD3) and `gzip` (GZIP). Results conducted on a 2.1-GHz i7-4600U system with 8 GB memory.

		Size (MB)		
File #	*Base #*	*VMMZ*	XD3	*GZIP*
139992	41	18	25	139
140106	41	30	38	150
140403	41	41	52	162
140537	41	34	44	156
140389	45	33	42	160
140458	45	12	18	141
140484	45	13	20	143
139996	46	47	55	179
140054	46	46	52	177
140515	46	39	44	175
140061	47	23	31	157
140080	47	29	39	164
140126	47	18	24	150
140384	47	19	29	156
140456	47	23	31	156
140534	47	33	41	164
140545	47	17	22	149
140151	49	31	41	160
140522	49	30	41	160
140538	49	28	37	157
139991	50	50	60	175
140032	50	45	55	171
140453	50	47	57	173
140489	50	49	59	175
140514	50	55	66	180
140082	51	25	29	161
140122	51	13	16	150
140154	51	65	73	193
140388	51	59	69	188
140459	51	59	68	189
140490	51	61	69	190
140525	51	56	64	187
140039	52	15	23	145
140058	52	27	37	157
140105	52	14	25	143
140390	52	28	38	157
140485	52	29	52	158
140541	52	13	44	142
139997	53	57	42	189
140035	53	59	18	190
140083	53	25	20	161
140150	53	33	55	166
140402	53	33	52	166
140488	53	61	44	191
140516	53	69	31	200
140491	54	23	39	150
140524	54	31	24	157
140544	54	29	29	155
140057	55	36	31	161
140394	55	32	41	158
140494	55	31	22	159
140513	55	32	41	159
139993	57	36	41	161
140399	57	42	37	168
140495	57	26	60	155
140546	57	31	55	156
140395	58	29	57	156
140517	58	32	59	158
140547	58	32	66	156
139995	59	29	29	158
140079	59	26	16	152
140107	59	30	73	157
140127	59	15	69	145
140397	59	31	68	158
140461	59	28	69	156
140520	59	28	64	158
140542	59	31	23	159

	VMMZ	*XD3*	*GZIP*
Average size (MB)	34	42	163
Encoding time (sec)	172	8.94	14.7
Decoding time (sec)	22.7	1.06	3.41

(executable code), and the .reloc section (image relocations). While sections hold special meaning for linkers and loaders, the proposed VMMZ compression framework as described above is agnostic to section type, loading every physically resident page from every section into the baseline memory. However, in the case of relocatable executable code, such section-agnostic loading causes some compression inefficiency.

Specifically, PEs are rarely loaded at the virtual-memory addresses for which they are compiled, requiring a .reloc section that lists all the memory addresses (or "fixups") in the executable code (the .text section) which must be relocated during loading before the PE is executed. In our delta-encoding application, performing these fixups when loading a PE into baseline memory would result in a predicted memory that would be closer to the current memory, resulting in improved xdelta3 encoding. Unfortunately, while the decoder has access to the fixups in the .reloc section of the PE file, these fixups are expressed in terms of virtual-memory addresses which must be translated to physical-memory addresses during loading. Permitting the decoder to be able to perform these fixups would consequently require storing the virtual-to-physical mapping for the fixups in the header of the compressed file. Our empirical investigations have revealed that this additional header overhead would likely outweigh any improvement in xdelta3 encoding that would result from including the fixups in the predicted memory. Consequently, our proposed VMMZ coder loads executable-code .text sections as is without performing the fixups that would be done in a real system by a linker/loader.

6. CONCLUSIONS

In this paper, we have considered the lossless compression of virtual-machine memory dumps for a target application of dynamic malware analysis. Typically, in such dynamic analysis, malware samples are run in a virtual machine just long enough to activate; consequently, memory dumps from the currently running virtual machine are substantially identical to that of the baseline machine, with the difference being attributable in a large degree to the loading of various executable programs and dynamically linked libraries. By duplicating the loading of these executables and libraries into the baseline memory, our proposed approach produces a prediction of the current memory from which delta encoding is performed, resulting in a significant improvement in compression performance over straightforward delta coding alone. In experimental results for a body of malware samples, the proposed approach outperformed the widely used xdelta3 delta coder by approximately 20% and the popular generic gzip coder by 79%.

REFERENCES

Burrows, M., & Wheeler, D. J. (1994, May). *A block-sorting lossless data compression algorithm* (Technical Report No. 124). Digital Equipment Corporation.

Egele, M., Scholte, T., Kirda, E., & Kruegel, C. (2012, February). A survey on automatic dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, *44*(2).

Farmer, D., & Venema, W. (2005). *Forensic discovery.* Addison-Wesley.

Fowler, J. E. (2000, August). QccPack: An open-source software library for quantization, compression, and coding. In A. G. Tescher (Ed.), *Applications of digital image processing xxiii* (p. 294-301). San Diego, CA.

Korn, D. G., MacDonald, J. P., Mogul, J. C., & Vo, K.-P. (2002, June). *The VCDIFF generic differencing and compression data format.* RFC 3284.

Ligh, M. H., Case, A., Levy, J., & Walters, A. (2014). *The art of memory forensics: Detecting malware and threats in Windows, Linux, and Mac memory.* Wiley.

Microsoft. (2013, February). *Microsoft portable executable and common object file format specification.* (Rev. 8.3)

Ziv, J., & Lempel, A. (1977, May). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, *23*(3), 337-343.

Ziv, J., & Lempel, A. (1978, September). Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, *24*(5), 530-536.