2018

mrsh-mem: Approximate Matching on Raw Memory Dumps

Lorenz Liebler
University of Applied Sciences, Darmstadt, Germany

Frank Breitinger
University of New Haven, fbreitinger@newhaven.edu

Publisher Citation

Lorenz Liebler and Frank Breitinger. "mrsh-mem: Approximate Matching on Raw Memory Dumps". In: 2018 11th International Conference on IT Security Incident Management IT Forensics (IMF). 2018, pp. 47–64. Available from IEEE Xplore.

Comments

This is the authors' accepted version of the paper published by IEEE. The version of record can be found at http://dx.doi.org/10.1109/IMF.2018.00011.

mrsh-mem: Approximate matching on raw memory dumps

Lorenz Liebler
University of Applied Sciences, Darmstadt
da/sec - Biometrics and Internet Security Research Group
Darmstadt, Germany
Email: lorenz.liebler@h-da.de

Frank Breitinger
University of New Haven, USA
Cyber Forensics Research and Education Group (UNHcFREG)
New Haven, USA
Email: FBreitinger@newhaven.edu

***Abstract*—This paper presents the fusion of two subdomains of digital forensics: (1) raw memory analysis and (2) approximate matching. Specifically, this paper describes a prototype implementation named `MRSH-MEM` that allows to compare hard drive images as well as memory dumps and therefore can answer the question if a particular program (installed on a hard drive) is currently running / loaded in memory. To answer this question, we only require both dumps or access to a public repository which provides the binaries to be tested. For our prototype, we modified an existing approximate matching algorithm named `MRSH-NET` and combined it with `approxis`, an approximate disassembler. Recent literature claims that approximate matching techniques are slow and hardly applicable to the field of memory forensics. Especially legitimate changes to executables in memory caused by the loader itself prevent the application of current bytewise approximate matching techniques. Our approach lowers the impact of modified code in memory and shows a good computational performance. During our experiments, we show how an investigator can leverage meaningful insights by combining data gained from a hard disk image and raw memory dumps with a practicability runtime performance. Lastly, our current implementation will be integrable into the volatility memory forensics framework and we introduce new possibilities for providing data driven cross validation functions. Our current proof of concept implementation supports Linux based raw memory dumps.**

***Keywords*-Memory analysis, Forensic analysis, Approximate matching, Fuzzy hashing**

I. Introduction

Over the past years memory became very affordable and most machines currently have 8 GB, 16 GB or even 32 GB of memory. As a consequence, software (benign and malicious) often operates in memory only, e.g., non persistent malware only exists in memory. Understanding and being able to analyze memory can give forensic investigators valuable and meaningful insights into a running system. The community and industry came up with different solutions for the analysis of acquired memory dumps which fall into one of two major categories:

- **Interpretation of structures:** Most tools and frameworks utilize structured analysis, i.e., the software interprets the complex system related structures, where two well known memory forensic suites are Rekall[1] and Volatility[2]. In detail, the frameworks deal with different formats of acquisition, the concepts of virtual memory management, the underlying architecture and the operating system related structures. Memory profiles are used to close the semantic gap and enable to perform such a structural examination. The examination with structured analysis techniques also yields new ways of evasion, which are encountered by examining and correlating different sources of OS related structures.
- **Memory carving:** There are also tools for unstructured analysis to extract information out of memory dumps. Those tools are important for different tasks like string or key extraction [7]. Carving memory has the major advantages of being more robust against malicious evasion or domain specific deallocations of important structures. In addition, those tasks can achieve a high IO throughput, are good parallelizable and offer a fast access to valuable insights. With the increasing size of memory, methods of data reduction (similar to those for hard drive forensics) are needed [30].

A core task of memory forensics is the enumeration of running or already terminated processes. The task of identifying processes in memory dumps is well discussed and different solutions have been proposed. Memory forensic frameworks and commercial software products interpret the structures of the operating system, which are responsible for process management, execution and allocations. The correlation of different characteristics during memory analysis is also denoted as cross validation, which empowers to detect malicious activities, e.g., a process with partially implemented evasion features. Infection vectors based on injection, process possession or the simple reuse of benign process names require additional steps of investigation. Beside malicious manipulations, legitimate alterations or deallocations by the operating system itself could also hinder a structural examination. This fact motivates to explore data-driven analysis techniques, i.e., memory based carving.

[1] https://github.com/google/rekall (last accessed 2018-02-10).

[2] https://github.com/volatilityfoundation/volatility (last accessed 2018-02-10).

In the course of binary (malware) analysis, different techniques have been proposed to identify and compare code related structures. As those two steps barely describe the process of memory carving, we mention them in their respective fields of application. The interpretation of statistical structures showed promising results to classify malware [1] or to identify code [32]. Additional steps of processing and feature extraction can encounter obfuscation or variances [18, 28]. The application of feature hashing [15, 16] or the compression based comparison [24, 25] also showed promising results for clustering and classification. Cohen and Havrilla proposed the reduction of shared code by normalizing and hashing disassembled code on a function level. Jin et al. extended the approach by the utilization of locality sensitive hashing and semantic hashes. Assembly code clone search detection systems have been proposed to detect similarities between malicious code samples [10, 11, 26].

Most of the mentioned approaches utilize statistical properties of the instruction sequences itself. They are discussed within the scope of processing executable code in a file context and not in the scope of carving file fragments contained within a raw memory dump, which bares additional pitfalls and considerations. As some of those techniques showed promising results in related fields of application, we leave the question unanswered if all of those approaches are adoptable for memory forensics. We further describe our field of application and its restrictions in Section III-A.

In the field of memory forensics signature-based analysis has been recently adopted [9]. The research underlines the idiosyncrasies, pitfalls and needed adaptations to apply signatures to this domain. On the other hand, [29, 30] proposed the utilization of cryptographic hash functions to perform code integrity checks, tamper detection, and do white- or blacklisting. The authors discuss the process of whitelisting normalized executables on a page level with the help of a golden image baseline. The work emphasize the obstacle changes caused even by the loader itself. Garfinkel and McCarrin presented an approach that covers the main considerations and pitfalls of our work in general terms. In contrary to a whole-file hashing, a concept called hash-based carving was introduced, which can identify files that are fragmented, files that are incomplete, or files that have been partially modified. The publication covers similar aspects of our work within a general scope. We propose the usage approximate matching techniques for solving this task in the domain of process-related memory forensics and outline our choice in Section V.

In this work we present a novel approach that allows detecting similarities between software stored on hard drives and loaded as modules into memory (Linux only). For processing the physical memory dumps and for damping possible loading-traces, we rely on `approxis` [20] an approximate disassembler performing carving of code-related structures. To compare the content of the dumps with the content of hard drives, we borrowed concepts from a subdomain of digital forensics called approximate matching. In a nutshell, these algorithms can be used to find similarities between different digital objects (e.g., compare the similarity between two text documents; details see Sec. II-B). We consider our approach as robust for memory based carving of code related fragments, as our implementation relies on the possibly scattered code structures itself. Thus, our approach does not depend on critical system related structures, the manual adaptation of signatures or the specification of any alignment properties. Using approximate matching for memory forensics is not new and was already discussed where most researchers questioned the applicability and runtime efficiency of those algorithms [9, 30, 22]. We discuss the application of approximate matching in the scope memory carving and release a prototype implementation which shows good computational performance. To the best of our knowledge, this is the first usable implementation of an approximate matching technique, which integrates an additional step of disassembling.

In detail, we introduce a derivative of a particular approximate matching algorithm (`MRSH-NET`[3]) and present `MRSH-MEM` which allows to detect fragments of code in the course of white- or blacklisting. Our paper has the following contributions:

- We interface approximate matching with an additional layer of approximate disassembling to process physical memory which is accomplished by integrating a recently published disassembler `approxis` into `MRSH-NET`.
- We demonstrate the capabilities of our approach to identify code structures in large amounts of raw data by the extraction of allocable code sections from different resources (e.g., online repositories or hard disk images).
- We demonstrate an acceptable runtime performance for processing memory dumps with a reasonable and realistic size.
- Besides our prototype implementation, we demonstrate a first application to identify kernel structures in memory, i.e., we profile the current running Kernel version inside a previously acquired raw memory dump.
- Lastly, we show the detection of code fragments of a running process in User Memory Space and show the capabilities to identify the currently running version.
- We publish our current proof of concept implementation[4] and will outline steps of further improvements.

The remainder of this paper is organized as follows: In Section II we formally introduce the Linux memory management system, the properties of approximate matching and the

[3] Note, `MRSH-NET` originated from the `MRSH`-family which has other derivations like `MRSH-v2`. The overall procedure of all approaches is very similar and therefore will will only talk about `MRSH-NET` for simplicity reasons.

[4] https://github.com/dasec/approximate-memory

recently published approximate disassembler approxis. In Section III we introduce our approach, its functionality and its basic layers of processing. Section IV discusses some practical applications and the possible integration into structured analysis. In Section V we introduce existing research, which focuses on physical memory examination. In Section VI we conclude our findings and finally introduce some further extensions to our current proof of concept implementation in Section VII.

II. Background

In this section we will introduce some memory management techniques of the Linux operating system, core functionalities of approximate matching algorithms and a recently introduced technique of approximate disassembling. The introduced foundations have to be considered in the following sections and play an important role in the remainder of this work.

A. Linux Virtual Memory Management

The next few paragraphs outline the Linux memory management from a high level few; this is by far not a complete description which would be beyond of scope for this paper. We will summarize some core primitives of the virtual memory management (VMM) as well as some important peculiarities of the Linux memory management. The knowledge of some fundamentals are indispensable for understanding further parameter settings and demonstrated use cases.

Linux implements the concept of VMM, which introduces a layer of indirection and maps one Virtual to one or more Physical addresses in RAM. This has two major benefits: First, each process can have its own address space, which improves process separation and security (the memory of other processes stays hidden). The second advantage is the possibility to swap out memory to hard disk and to store it independent of its underlying physical scheme. A process can share fragments with other processes to reduce the occupied physical memory. The concept called shared memory therefore maps physical memory, which could be used by different processes at the same time, into each of those processes. Thus, the physical memory gets mapped into multiple processes at once, as long as the fragments are not changed by a process. As soon as a process changes the shared memory area, the changed version gets copied into a independent memory area for the process itself (copy on write). The memory mappings could be additionally equipped with access permissions, which controls the access for reading, writing or executing memory. The mapping is performed with the help of the Memory Management Unit, which links the CPU with the memory itself. The Translation Lookaside Buffer implements a system for fast buffering.

The virtual address space is split into an upper part and lower part also named Kernel Space and User Space, respectively. The border of this split is denoted as CONFIG_PAGE_OFFSET and differs for different architectures and systems. For instance, on a 32-bit system the offset is normally at address 0xC0000000. Given that the Kernel address space is above CONFIG_PAGE_OFFSET it would be 1 GB of a 4 GB system. On 64-bit systems this offsets varies by architecture, e.g., common boundaries are ARM=0x8000000000000000 and x86_64=0xffff880000000000. Focusing on the upper part (the Kernel space), there is usually another separation as shown in Figure 1.

We could roughly differ the Kernel Address Space in three different areas: Kernel Logical Address Space, Kernel Virtual Address Space and the User Virtual Address Space. Note, the definitions and wordings slightly differ across sources and literature. The **User Space** is located below the defined PAGE_OFFSET and stores the user space programs, where each process manages its own mappings. Not all portions of the process need to be loaded and mapped to memory until they are actually needed. The memory is mapped to Physical Address Space in a non-contiguous fashion. In case of high memory usage, the fragments of memory can be additionally swapped to hard disk and moved inside the Physical Address Space. It should be clear, that the User Space is unsuitable for Direct Memory Access. The **Kernel Logical Addresses** (Low Memory) are stored about the PAGE_OFFSET and are contiguously mapped with a fixed offset to the Physical Address Space. The area itself can never be swapped on hard disk and thus could be used for DMA. The address space above the Kernel Logical Addresses contain the so called **Kernel Virtual Addresses** (High Memory). This address space is mapped with non-contiguous memory mappings and unsuitable for DMA transfers.

The Kernel Address Space starts with the mappings of the Linux Kernel and its allocated segments. Above, the loaded vmlinux image, the Loadable Kernel Modules are located. A Loadable Kernel Module is defined in the ELF (Executable and Linking Format) format and can be invoked to a running system. In the file system a LKM is normally denoted with the .ko extension and those modules provide a broad variety of extensions to a running system. They are also to be known as often attacked by malicious kernel-level rootkits.

The mapping between the virtual and the physical space is handled by the Memory Management Unit which operates on memory pages (i.e., it maps one or multiple virtual pages to one physical page). Thus, a page is a fixed size allocation of virtual memory, which is aligned to this size in memory. The size of a page can vary for different architectures as shown in Table I and is set during the kernel build time.

We will not describe the concept of page tables, the functionality of TLB and the translation process itself, as this would be out of the scope of this work. Additionally the

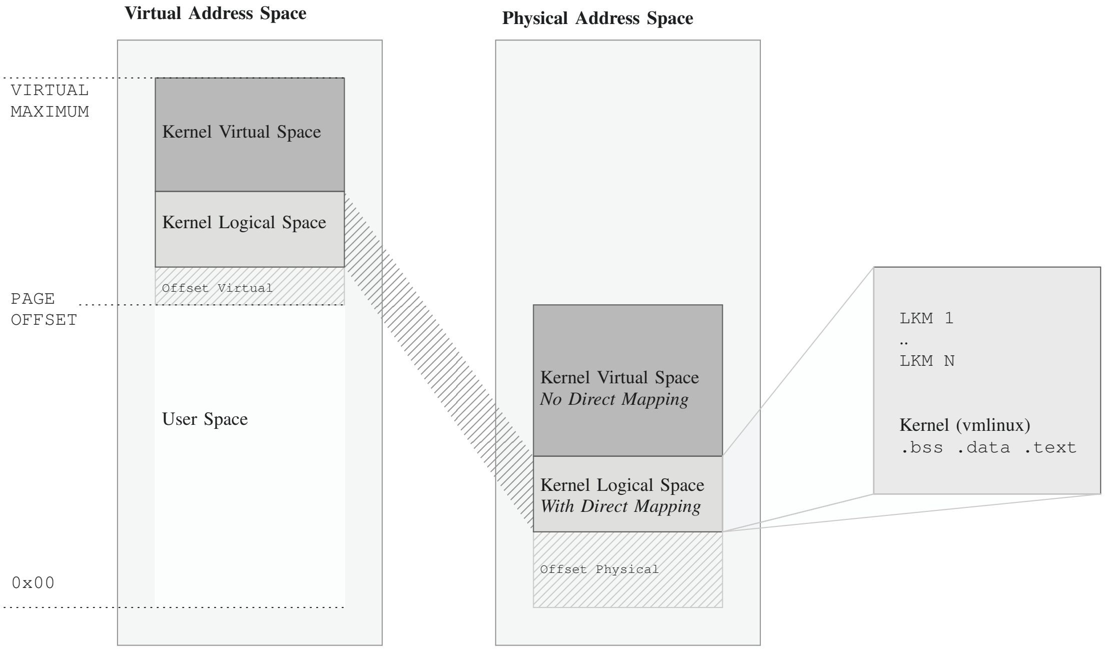

Figure 1. Simplified Linux Memory Address Layout

Architecture	Page Size
ARM	4 KB
ARM64	4 KB or 64 KB
MIPS	Configurable
x86	4 KB

Table I
OVERVIEW OF COMMON PAGE SIZES.

concept of Shared Memory, Lazy Allocation and (Kernel) Address Layout Randomization will not be described in this paper. However, we need to consider several implications caused by the above mentioned concepts as those concepts strongly differ from normally processed data of approximate matching techniques. We summarized the concepts and formalized general implications, which should be respected in our approach. After the introduction of some core primitives of approximate matching and approximate disassembling, we will give a comprehensive overview of our considerations in Subsection III-A.

B. Approximate Matching

Approximate matching (a.k.a. Fuzzy hashing or similarity (digest) hashing) is a rather new area of digital forensics and can be seen as the counterpart to traditional (cryptographic) hash functions, i.e., approximate matching algorithms return similar fingerprints for similar inputs. In the following we summarized the most important aspects; more comprehensive overviews are provided by [3] and [14]. From a high level perspective, approximate matching algorithms work as follow. First, the algorithm identifies features where a feature is usually a substring of the complete input (e.g., chunks of a particular length). These chunks are then shortened which is often done using (cryptographic) hash functions. Lastly, these shorter strings are then used to build a fingerprint / similarity digest.

For instance, let us have a closer look at algorithms for the MRSH family which form the basis for this work. These algorithms (e.g., [5]) consider only the underlying byte sequence of a given input (no interpretation of the byte sequence). The given sequence is divided into chunks of size b (common values are $64 \leq b \leq 320$ bytes) . To do so, the algorithm uses a sliding window that rolls through the sequence byte-by-byte and considers 7 consecutive bytes at a time. This window is then hashed using a *pseudo random function (PRF)* which returns a value between 0 and b. If $b == 0$, the end of a chunk is identified. As a consequence, if PRF behaves pseudo random, each chunk has approximately the size b bytes. Once the end of a chunk is identified, a *Chunk Hash Function (CHF)* is used to compress the sequence (common CHFs are MD5, SHA or FNV-1a). Lastly, all chunk hashes are translated into a final fingerprint where different algorithms use different concepts. In case of MRSH-NET, a single large Bloom filter is used which is explained in the following paragraph

A **Bloom filter** is a space-efficient, probabilistic data structured invented by Burton Howard Bloom in 1970 [2] that consists of an array of m bits all set to zero. In order to insert an element $s \in S$ into a Bloom filter, s is hashed using a hash function that returns values $|h(s)| \geq k \cdot log_2(m)$ bits[5]. Then, the first $log_2(m)$ bits are used to set the corresponding bit in the Bloom filter; the second $log_2(m)$ bits are used to set the corresponding bit in the Bloom filter; this is repeated k times. For instance, assuming Bloom filter size $m = 64 = 2^6$ and $k = 2$, $h(s)$ should return a hash value of at least $(2 \cdot log_2(64) =)$ 12 bits, e.g., 011011 101101. Given that $011011_{bin} = 27_{dec}$ and $101101_{bin} = 45_{dec}$, bits 27 and 45 of the Bloom filter are set to one. To verify whether an element s' is in a given Bloom filter, it is hashed with the same hash function h. If all corresponding bits in the Bloom filter are set to one, the element was inserted into the Bloom filter with a certain probability (there is a chance for a false positive). If one of the bits is zero, the element was never inserted into the Bloom filter (there are not false negatives). Specifically, the false positive rate of a Bloom filter is influenced by three parameters: the size of the filter m, the amount of elements which are inserted into the filter n and the number of set bits per element k. The probability for a false positive can then be estimated with the formulas illustrated in Eq. 1

$$\begin{aligned} P_{FP} &= \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{k} \approx \left(1 - e^{-kn/m}\right)^{k} \\ &= (1-p)^{k}, \text{with} \\ p &= 1 - \left(1 - \frac{1}{m}\right)^{kn}, \end{aligned} \tag{1}$$

where p is the probability of a bit being 0, after all n elements have been inserted.

In order to create the **final fingerprint**, the first $k \cdot log_2(m)$ bits of the chunk hashes are utilized to set the corresponding bits in the Bloom filter. In other words, for each chunk k bits are set in the Bloom filter. A summary of the parameters is provided in Table II.

b	Denotes the approximated chunk size
m	Denotes the Bloom filter size in bits
n	Number of elements inserted into a Bloom filter
k	Number of used sub-hashes; each sets a bit in the corresponding Bloom filter

Table II
PARAMETERS OF MRSH-NET AND THEIR DESCRIPTION.

The created fingerprints can be used to estimate the **similarity score** between two given files. Different approximate matching approaches create different fingerprints and thus utilize different techniques for similarity calculation. In the course of MRSH derivatives which utilize Bloom filters as similarity digest, the Hamming distance as metric is used. In the course of this work, we adopt approximate matching to identify chunks within an acquired memory dump. Similar to MRSH-NET, we can not expect a present file context when comparing extracted chunks against a database of files. We leave the question of better lookup strategies and chunk identification techniques open for further research. We discuss the details of our provisional solution to identify chunks in Section IV.

C. Approximate Disassembling

approxis is a fast approximate disassembler for unknown instruction sequences that was presented by Liebler and Baier [20] in 2017. In contrast to traditional linear sweep or recursive traversal approaches, approxis does not provide a full instruction decoding but focuses on computational efficiency which is accomplished using a pre-generated prefix tree. Note, we will use the terms of *disassembling* and *decoding* interchangeably in this paper. This was inspired by existing applications, like the distorm[6] stream disassembler. However, in contrast to distorm, approxis is less granular and does not operate on a bit level during disassembling an instruction.

Specifically, Liebler and Baier use a large ground-truth dataset of ELF files to generate a prefix tree (a.k.a. trie[7]) as simplified in Figure 2. All files in the corpus are parsed (e.g., the input byte sequence) and put into a trie structure where the focus lies on the mnemonics. For instance, 0x41 indicates a push and given that it is the ground truth it is known that the next byte (0x55) also belongs to the current instruction. Focusing on the next offset starting with 0x48, it is known that 0x48 can be lea, sub or mov and therefore the next byte (0x89) needs to be considered as well which then indicates a mov. Since the authors are only interested in approximate disassembling, approxis does not store the nodes and leaves colored in whited. Instead, the black colored decision node stores the representative mnemonic and the remaining length of the current instruction (length not shown in the figure). For instance, the black 0x64 would have stored a remaining instruction length of ‘2’. In the original trie implementation a lot of additional information is saved within each node: Frequency counts, most common instruction lengths, most

[5] Note, the original work suggests to use k different hash functions each returning a value between 0 and $m - 1$. However, we use a single hash functions and therefore our explanation differs slightly.

[6] https://github.com/gdabah/distorm/wiki/diStormInternals (last accessed 2018-02-10).

[7] https://en.wikipedia.org/wiki/Trie (last accessed 2018-02-10).

common mnemonic and several other information about the currently processed instructions. Those additional counts will be helpful for further statistical analysis described in the upcoming paragraphs. In order to disassemble an unknown ELF file, approxis parses the file and perform look ups in the trie. For instance, let us assume the byte sequence in Figure 2 is unknown. The opcode 0x41 55 will be translated into the mnemonic push 2 (as the approach is interested in the mnemonic and the length only). Given the same trie, the opcode 0x41 87 would end up in the identical mnemonic.

Obviously, not all byte sequences are decodable by the trie structure. For example, a sequence 0x48 ff e0 48 8d will stop at the undecidable intermediate node 0x48 as its successor 0xff is not present in the trie. To continue, the additional information (stored in the node) is utilized, e.g., frequency analysis of common instruction sequences and subsequent mnemonics. For the example in Figure 2 (right side), subsequent mnemonics would be [push,mov], [mov,sub], [sub,lea] and [lea,mov].

Based on these frequencies that are calculated over the complete ground truth corpus, the authors generate a confidence score λ which describes if two disassembled and subsequent instructions are plausible or not. In detail, the instructions are extracted as bigrams from the ELF ground truth and for each of those the probability p is saved as absolute logarithmic odds (logit). Coming back to the problem of the unknown byte (see Figure 3), the confidence score allows to predict the length of the instruction (i.e., where to continue the approximate disassembling). Precisely, 0xff is unknown and hence approxis does not know the length of the instruction. The algorithm depicts successively a candidate from the previously saved and most frequently occurred lengths. If the disassembler has success at a corresponding offset, the value of confidence counterchecks if the sequence of instructions is meaningful or not. In our example, the high confidence score (i.e., $\lambda = 17$) indicates that 0xe0 is most likely not a mnemonic and therefore the algorithm jumps to 0x48 8d which can be found in the try. Applying approxis on byte sequences that do not contain code fragments, the values of confidence will indicate that this snippet is nonsense. This behavior empowers to distinguish between fragments of code and data. Large blocks of continually similar byte sequences could lead to a wrong interpretation. An example could be long sequences of zero (0x00) padding bytes, which would be misleadingly decoded to an add instruction. To avoid this behavior the current implementation makes use of a running length counter, which penalizes and ignores such repeating sequences.

In the current implementation of approxis, the decoded instructions, the decoded instruction lengths and the determined values of confidence (λ) are stored in separated buffers. During the decoding of the input bytes the buffers store at the same offsets the corresponding informations of a specific sequence.

The buffers are listed in Table III. The raw image buffer buf_by with size INPUT_SIZE contains the raw byte sequence of an input stream (e.g. file, memory dump). The processing buffers can be adjusted manually by setting the corresponding parameter BUFFER_SIZE. The buffers contain decoded relative offsets of the instructions (buf_ro), the decoded (integerized) mnemonics of the instructions (buf_mn) and the values of local code confidence (buf_lo).

In the following paragraph we will summarize some major considerations. It should be clear that the field of application is not related to extended binary analysis and thus, the disassembler is not and should not be capable to proceed a full decoding of a present complex instruction set. The process should interface the steps of processing provided by MRSH-NET. The motivation and application of approximate disassembling is described in [20]. To summarize the characteristics of an approximate disassembler, we don't actually need a full decoding and only focus on the correct determination of instruction offsets and a mnemonic represent. We apply the disassembler on large amounts of raw data and do not expect a well formed executable with reasonable header information as input. The disassembler is applied in the field of memory analysis and not able to actually resolve virtual addresses. With the values of confidence the disassembler is able to discriminate between code and data. Additionally, approxis could differentiate the architecture of the code instructions [20].

III. Approach

As previously mentioned, this article presents MRSH-MEM which is a combination of approxis and an approximate matching algorithm MRSH-NET. The main goal is the possible comparison of memory images and hard-drive content, e.g., applications installed on the system and currently executed in memory. This will allow investigators to profile parts of the memory dump (whitelisting) or detect suspicious code patterns inside the memory dump (blacklisting). The proposed approach therefore enables to perform robust unstructured analysis of a memory dump. A strongly generalized overview of our use case is shown in Figure 4. The lefthand side outlines the already existing approximate matching techniques. The opposite side is the content of this article, i.e., how to modify / normalize a memory image so that we can generate a fingerprint / memory digest which then can be compared against a traditional approximate matching fingerprint.

A. Considerations

Before discussing our approach, we highlight some considerations which impacted our design decisions. In Section II we gave a brief introduction into the concepts of

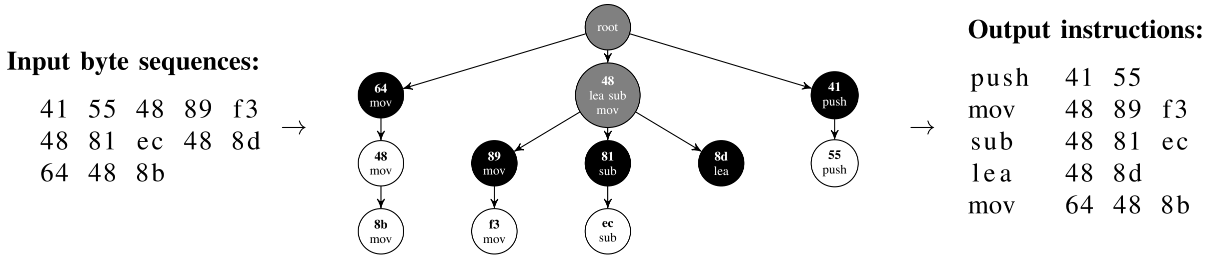

Figure 2. Oversimplified disassembling process with the help of a byte-trie structure.

Buffer	Size	Type	Description
buf_by	INPUT_SIZE	uint8_t	Stores the input file or stream, so it contains the raw bytes.
buf_lo	BUFFER_SIZE	uint8_t	Stores the value of confidence for two subsequent instructions.
buf_wi	BUFFER_SIZE	uint8_t	Stores the averaged confidence for WINDOW_SIZE instructions.
buf_ro	BUFFER_SIZE	uint8_t	Stores the relative offset of a instruction at the current offset.
buf_pe	BUFFER_SIZE	uint8_t	Stores the penalty value at a current offset.
buf_mn	BUFFER_SIZE	uint32_t	Stores the decoded mnemonic for a specific instruction.

Table III
USED BUFFERS OF THE CURRENT APPROXIS INTEGRATION.

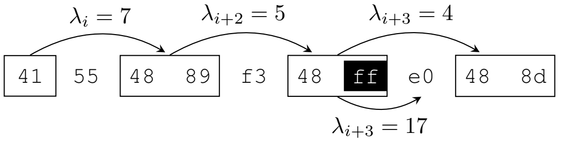

Figure 3. If the process of disassembling struggles, reasonable offsets could be selected by the examination of the saved values of confidence for different bigrams.

Virtual Memory Management, approximate matching and approximate disassembling. In the following listing we conclude some considerations, which have to be respected for processing raw physical memory:

I. **Mappings of virtually contiguous regions do not have to be physically contiguous.**
The fact that pages of a specific context do not have to be allocated contiguously in memory, is an important issue for the overall concept of transferring approximate matching to the field of memory forensics. Found features should be considered in a page-sized scope. This should lead to future research and concepts of composing separated page sizes.

II. **The page size can vary for different architectures and pages are aligned to its page size in memory.**
We expect the page size to be at least 4 KB which is the most common page size. This is important when selecting the block size b as it should be smaller than the page size. Explanation: a large b will reduce the amount of chunks within a page boundary of a physical memory dump. Considering non-contiguous physical pages, this could lead to producing features, that frequently overlap with neighboring pages (all details about b are discussed in Section III-D).

III. **Pages could be shared between processes, thus a physical could refer to multiple virtual pages but not the other way round. The concept is also called *Shared Memory*.**
This concept outlines, that virtual to physical mapping is actually not a one to one mapping. Especially in the case of shared libraries, it should be clear that those matches could not actually resolve a specific sample. Even if this is yet out of the scope of this work, we should consider it in further proceedings.

IV. **Each process context uses its own virtual mapping. We are not able to actually resolve the physical offset of a virtual address without translation or the analysis of system related structures.**
Our overall approach of processing is context unaware. Thus, we are not able to actually resolve a virtual reference. It should be clear, that examination on a higher level, e.g. the usage of recursive traversal or control flow graph analysis, are not applicable in a context unaware scenario.

V. **Not all requested pages of a process are allocated immediately. The concept is also called *Lazy Allocation*.**
In contrary to traditional hard drive forensics, the looked up data sample has not to be present in RAM

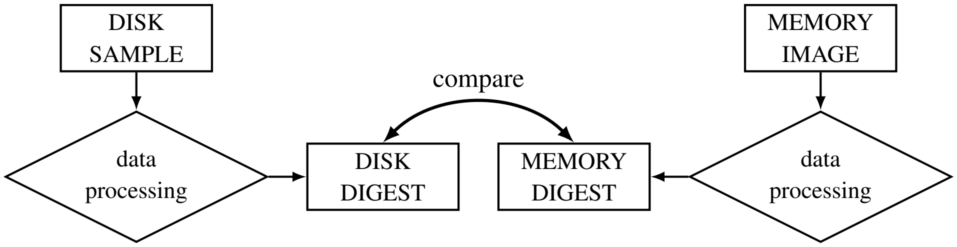

Figure 4. Overview of the overall approach and application of MRSH-MEM

completely during acquisition. We should consider this fact during examination of found chunks inside a target image.

VI. **In case of high memory usage, the kernel is able to swap content to hard disk, which is denoted as *Swapping*.**
Similar to *Lazy Allocations*, this concept should raise our awareness, that we should not expect a sample (i.e. in the case of loaded executables) which is fully loaded to memory at the time of acquisition.

VII. **We expect systems with 8 GB RAM to represent a reasonable upper bound for current consumer PC systems.**
Similar to previous publications, which have determined the required Bloom filter size for their field of application, we should determine the maximum required size for storing memory dumps. Even if it is common, that most of the acquired memory should be initialized with zero byte paddings anyways, we assume this value as possible upper bound, in case the acquired memory is well populated.

VIII. **Executables are changed during the process of loading to memory.**
Code on a hard disk should differ from its representation in memory. Legitimate changes to code would obviously cause the original fuzzy hashing techniques to fail. The PRF and CHF would possibly interpret even legitimate updates to the code structures. The possible pitfalls of applying traditional fuzzy hashes are twofold. First, the process of chunk extraction could be disturbed, as the PRF could trigger at different offsets. Second, the hash value of a extracted chunk could differ, as the CHF works on a byte-level of unnormalized code fragments.

B. Data Processing

In the following we will describe how we combined approxis with approximate matching. The workflow of data decoding and examination is depicted in Figure 5 and can be described as a multi-layered process. Note, even though the figure shows a clear separation between different steps, most of them are strongly interleaved and therefore it is hard to visualize the exact flow. A description of the steps depicted in the figure is given in the following (step one and two are nearly unmodified steps presented by Liebler and Baier [20]):

❶ The raw bytes from the memory are disassembled using approxis as discussed in Sec. II-C which will return the mnemonic as well as the length of the instruction. Especially the decoded mnemonic is important for further proceedings as the process of chunk extraction and chunk hashing.

❷ Using the confidence score λ and the concept of a simple running length counter allows to differentiate between code and data. The running lenght counter counts repeating mnemonics, e.g., a nop-slide, which should not be considered. Note, for our approach we will focus on code and neglect data.

❸ Having the approximate disassembled code, we now identify the chunk boundaries based on the mnemonics. Therefore, we utilize a sliding window approach on the mnemonics (precisely, the rolling hash runs over a C-buffer that contains the byte representations of the mnemonics). All details are provided in Sec. III-C.

❹ After identifying all chunks, an additional filter is applied to remove irrelevant chunks. To identify relevant chunks, we utilize the confidence score. For instance, the first three entries form chunk one (indicated by C_1) have a high confidence score (64, 64, 63), therefore we consider this chunk as not relevant (indicated by [0]).

❺ Lastly, the relevant chunks will be hashed and stored into a database. While this example focused on creating a chunk hash based on the mnemonic buffer, we can utilize other buffers as well for further comparisons, e.g., the raw byte buffer.

C. Implementation Details

The previous section outlined a high level perspective of the procedure where this section details about our concept. As mentioned, we are using a multi-layered process which is reflected by the usage of multiple buffers (buffers are summarized in Table III). Most of the working buffers are limited in its size and thus, have to be swapped during

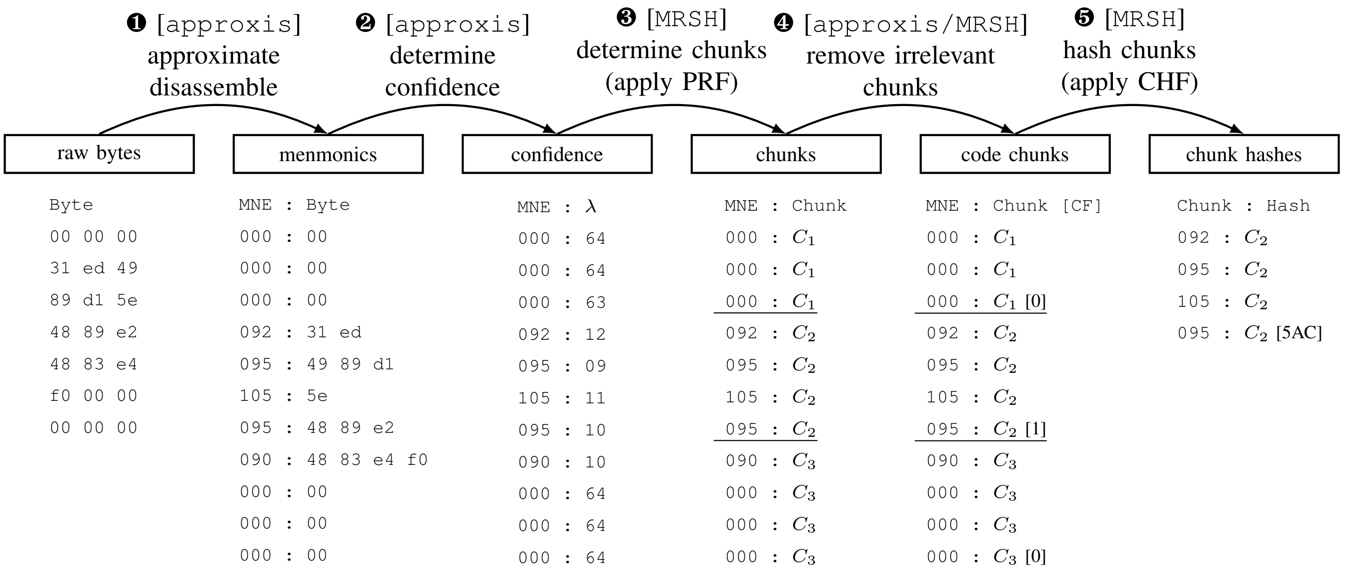

Figure 5. Overview of the data processing steps. The process outlines the interleaved characteristics of the overall approach. We highlighted integrated components of approxis and MRSH.

processing (i.e. buf_lo, buf_ro, buf_pe, buf_mn). We skip the details of the buffer swapping for simplicity, but recommend to consider the implementation as multiple circular buffers. For the prototype (and hence for the runtime performance evaluation) we expect that the input stream (i.e. buf_by) can be stored in memory completely. For a better understanding of the overall processing and the usage of the mentioned buffers, we explain the procedure based on a comprehensive example in the following section.

Figure 6 gives an example of how the different buffers are utilized. The example shows 10 steps of processing, where in each step a instruction is decoded from the byte buffer (buf_by, by) at the highlighted offset $\star$. The decoded instruction length and a corresponding mnemonic are saved into separate buffers after each offset in buf_ro, ro and buf_mn, mn, respectively. Thus, the current buffers steadily increase with each decoded offset. For instance, in row 1 the first seven bytes in the byte buffer are e8 00 00 00 00 83 2d. The disassembler decodes the first six bytes to the mnemonic 114, which represents a call instruction. The pointer moves to the next offset and repeats the decoding process similar to the first row. Thus, in row 2, the next byte sequence 83 2d 00 00 00 00 01 gets decoded to the mnemonic 91. Beside the mnemonic, we again store the corresponding length of the instruction 7 in the buffer ro. In the third row, we could see that the previous pointer was increased by 7 and the process repeats with the decoded instruction bytes 74 02.

Once the mnemonics are decoded, they are 'added' to our sliding window, e.g., in row one the mnemonic 114 is the first entry of the rolling hash. Given that the rolling hash has a length of seven, the last seven mnemonics serve as input for the PRF. As the amount of mnemonic representatives is larger than 2^8, the hash value is calculated over a sequence of integers, which stores the mnemonic represents decoded by approxis. As soon as the PRF determines a chunk boundary (see the output of the PRF in row 5 and row 8 of Figure 6), the decoded instructions (buf_mn) are hashed using the chunk hash function (CHF, FNV-1a algorithm [12]). The hash value is then stored into the Bloom filter (the hash value is separated into 5 sub-hashes where each sets a bit of the Bloom filter; the procedure is identical to MRSH). In Figure 6, we represent the hash values by a shortened representation in the last column denoted as BF. The buffers by and ro are cleared out after each chunk extraction (see row 6 and 9 in Figure 6), where the sliding window of the applied PRF keeps the buffer of the last 7 mnemonics for the next decoding pass.

To filter code related chunks and to reduce the overall amount of Bloom filter inserts, we utilizes the introduced value of confidence for consecutive instructions (see Section II-C). Therefore, chunks are inserted into a Bloom filter as soon as they fulfill two properties. First, the chunk size has to contain at least 10 consecutive instructions. Second, the overall amount of considerable meaningful mnemonic bigrams has to be at least 30 % for a current chunk.

Hashing the decoded byte sequences (buf_mn) will for example neglect all byte sequences which represent operand information within an instruction. More importantly, different opcodes on a byte level can been mapped to the same mnemonic representative. Besides hashing the decoded instructions, one could also hash the other buffers. In this

No.	buf_by buf_ro	buf_mn	PRF(buf_mn[-7:])	BF (Hash)
1.	by: … 00 00 00 00 ★ e8 00 00 00 00 83 2d 00 00 00 01 74 02 f3 c3 … ro: 5	mn: 114	PRF(114) → ✗	[]
2.	by: … 00 00 00 00 ★ 83 2d 00 00 00 00 01 74 02 f3 c3 e8 00 00 00 00 … ro: 5 7	mn: 114 91	PRF(114 91) → ✗	[]
3.	by: … 00 00 00 01 ★ 74 02 f3 c3 e8 00 00 00 00 e9 00 00 00 00 66 0f … ro: 5 7 2	mn: 114 91 44	PRF(114 91 44) → ✗	[]
4.	by: … 00 01 74 02 ★ f3 c3 e8 00 00 00 00 e9 00 00 00 00 66 0f 1f 44 … ro: 5 7 2 2	mn: 114 91 44 330	PRF(114 91 44 330) → ✗	[]
5.	by: … 74 02 f3 c3 ★ e8 00 00 00 00 e9 00 00 00 00 66 0f 1f 44 00 00 … ro: 5 7 2 2 5	mn: 114 91 44 330 114	PRF(114 91 44 330 114) → ✓	['c42']
6.	by: … 00 00 00 00 ★ e9 00 00 00 00 66 0f 1f 44 00 00 e8 00 00 00 00 … ro: 5	mn: 115	PRF(114 91 44 330 114 115) → ✗	['c42']
7.	by: … 00 00 00 00 ★ 66 0f 1f 44 00 00 e8 00 00 00 00 48 83 ec 70 48 … ro: 5 6	mn: 115 14	PRF(114 91 44 330 114 115 14) → ✗	['c42']
8.	by: … 1f 44 00 00 ★ e8 00 00 00 00 48 83 ec 70 48 89 e7 e8 00 00 00 … ro: 5 6 5	mn: 115 14 114	PRF(91 44 330 114 115 14 114) → ✓	['c42', '2b4']
9.	by: … 00 00 00 00 ★ 48 83 ec 70 48 89 e7 e8 00 00 00 00 48 8b 54 24 … ro: 4	mn: 91	PRF(44 330 114 115 14 114 91) → ✗	['c42', '2b4']
10.	by: … 48 83 ec 70 ★ 48 89 e7 e8 00 00 00 00 48 8b 54 24 20 48 2b 54 … ro: 4 3	mn: 91 95	PRF(330 114 115 14 114 91 95) → ✗	['c42', '2b4']

Figure 6. Example of the overall processing pass with different buffers of the raw buffer (by), the buffer of decoded offsets (ro) and the decoded mnemonics (mn). The current decoded offset is denoted with ★ and shifted by the amount ro after each step.

prototype, we actually propose the hashing of two buffers, the decoded buffer of representatives (buf_mn) and the original input buffer (buf_by). This empowers an investigator to detect similar instruction sequences and inspect possible deviations on a byte level.

We summarize two central adaptations to the original MRSH and approxis implementation. In contrary to previous bytewise approximate matching approaches, the chunk boundaries are defined with the help of the *decoded byte sequences*, not the byte sequences itself. Additionally, the code detection is not performed within a fixed-sized sliding window as proposed by Liebler and Baier [20], but rather on a chunk level.

D. Configurable Parameters

An overview of the important configurable parameters is given in the following subsection. We additionally give a short explanation and reasoning of the parameters and the selected default values. We will first describe the important parameters which define the overall chunk extraction process. An overview of the parameters could be seen in Table IV.

Selecting the feature (chunk) size (b): As already introduced, the PRF approximately defines the extracted chunk sizes. Considering the minimum respected page size of 4KiB and the presence of non-contiguous memory mappings, we depict a default value of $b = 64$.

Code confidence: The process of filtering chunks, which store code fragments, could be controlled by two parameters. The parameter CODE_THRESH describes the maximum value of λ which defines two consecutive instructions to be meaningful or not. The values of λ are stored within the buffer buf_lo during a decoding pass. If less than 10 consecutive instructions are detected within a chunk, the chunk will not be inserted into the Bloom filter. Additionally, considering large chunks, we measure the total amount of instructions within a chunk. We require at least 30 % of the decoded instructions inside a chunk, before the chunk will be inserted into the Bloom filter. We denote this threshold of code coverage as parameter CODE_COV.

Penalties: The original implementation of approxis considers large amount of repeating decoded mnemonics as less meaningful. An example could be misleadingly decoded sequences of non-allocated zero bytes or other padding instructions (e.g., NOP instructions). The running length counter of approxis counts subsequent similar decoded mnemonics. As soon as the running length counter

Parameter	Range	Default	Description
BLOCK_SIZE	[0,1]	64	Defined modulus and approximated size of a chunk (b).
CODE_THRESH	[0-100]	30	The value defines the threshold of code confidence. A sequence of instructions should be considered as code fragments, as soon as the value of confidence is lower than the defined CODE_THRESH.
CODE_COV	[0-1]	0.3	Defines the minimum required percentage of code coverage within a chunk, before it gets hashed and inserted into the Bloom filter.

Table IV
PARAMETERS OF THE MRSH−MEM IMPLEMENTATION.

instruction exceeds RLE_THRESH, a penalty is written into the buffer buf_pe. The saved penalty is added to the value of confidence stored in the buffer buf_lo afterwards. Beside the threshold we additionally configure a factorial, which decreases the current running length after a sequence of similar mnemonics was interrupted. We increase the RLE_DRAIN to respect the non-contiguous properties of physical memory dumps.

Determining the Bloom filter size (m): In the former section we described the idiosyncrasies and properties of memory management. In this paragraph we explain the parameter adaptations and the following impacts to the needed Bloom filter size. For further details to the following formulas we refer to Breitinger et al. [6]. We consider 8 GiB as reasonable RAM size and select the expected input size (s) to be $s = 8$ GiB.

With the expectation that a modulus b defines a trigger point and thus the probability of a hit is reciprocally proportional to the average chunk size, we estimate the number of extracted chunks n for a given input image with size s. The calculation could be seen in equation 2.

$$n = \frac{s \cdot 2^{20}}{b} = \frac{8 \cdot 1024 \cdot 2^{20}}{64} = 134,217,728 \qquad (2)$$

In Breitinger et al. [6] the authors mention that the choice of k is limited by the used FNV-1a hash function. Thus, the value of k is limited to $5 \leq k \leq 7$. Similar to MRSH−NET we choose the value of k to be $k = 6$. A single Bloom filter of size 32 MiB could be used to monitor approximately 2 GiB of data, whereas as Bloom filter of size 2 GiB could approximately monitor 100 GiB of data [4]. Obviously, the filter has to be stored in memory during examination. Similar to Breitinger and Baggili [4] we consider this size as still manageable even on casual or mobile systems. To determine the maximum needed size of the Bloom filter in dependency to the expected input size, we depict the corresponding formula from [6]. In addition, we set the parameter r, which defines the minimum amount of correctly to be identified consecutive features to $r = 6$. Considering equation 3 and the above mentioned parameters, we propose a Bloom filter size of $m \approx 7.0426 \cdot 10^8$ bits ≈ 84 MiB. An overview of the configurable parameters could be seen in Table VI.

$$m = -\frac{k \cdot n}{\ln(1 - \sqrt[k \cdot r]{p})}, \text{ where } p = (1 - e^{-kn/m})^k. \qquad (3)$$

IV. APPLICATION

The implementation of MRSH−NET uses a single, large Bloom filter which bares two notable disadvantages: memory consumption and the lack of file identification, i.e., the approach can only answer the question if a file is contained in a given Bloom filter, but we cannot say to which file a similarity exists. However, we decided to use the identical approach for realizing the prototype; we will evaluate possible strategies in future research to match chunks with a given file base.

The adaptation and integration of a single Bloom filter (BF) gives us a good computational performance for initial white- or blacklisting of extracted chunks. However, to perform the capabilities of better identification, we additionally create a database of extracted *chunk hash values* (CHV). The current *chunk hash database* (CHDB) consists of single large lookup tree, which stores all chunk hash values with a corresponding file name inside each leaf node. As we focus on computational speed and expect better solutions for a fast file identification, we do not consider the database in the case of runtime performance analysis or memory consumption (see Section VII for further discussions). Figure 7 provides an overview of the general application. First, an investigator has to acquire the dumps (memory and hard disk). Additionally, the acquisition of files from different repositories can be considered. All input files are processed with MRSH−MEM and stored in the Bloom filter as well as in a database of known code fragments (CHDB). Applying MRSH−MEM on the acquired memory dump will then answer the question if a particular memory fragment is found in the BF. The comparison against the database will allow to answer the question which file was matched.

Note: Different versions of the same executable can share the same code base. Thus, similar chunk hash values can occur, which will be inserted into the Bloom filter digest. Leaf nodes in our CHDB, which are occupied by chunks of

Parameter	Range	Default	Description
RLE_THRESH	[0-100]	10	Sets the threshold when the repeating sequences of instructions should be considered as not valid.
RLE_DRAIN	[0.1-1.0]	0.9	The value defines a factorial, which lowers the running length counter significantly faster. If the value is lower than the defined threshold, we switch to stepwise decrementing the running length counter.

Table V
PARAMETERS OF THE MRSH-MEM IMPLEMENTATION FOR CONTROLLING THE PROCESS OF DISASSEMBLING.

Parameter	Range	Default	Description
BF_SIZE_IN_BYTES	[0,X]	128 MiB	Size of the Bloom filter (b, must be a power of two).
SUBHASHES	[5,7]	6	Number of used subhashes (k).
MIN_RUN	[0,X]	6	Minimum amount of correctly to be identified consecutive features (r).

Table VI
PARAMETERS OF THE MRSH-MEM IMPLEMENTATION.

multiple versions of an executable (e.g., the same chunks have been extracted for multiple versions of a file) are denoted in the following plot as *multiple* hits. Chunk hash values, which only appeared for a single version, are marked as *single* hits.

Test environment: For testing purposes, we acquired memory and hard disk fragments from an existing Debian 8 installation, which was originally setup inside a virtualized environment for common network analysis tasks. In detail, we inspected a Debian 8 installation (Debian 3.16.7 x86_64 GNU/Linux) running with the help of Virtual Box (Version 5.2.6 r120293). The system contains several real world applications and was used for several weeks without a reboot. To acquire the memory we used LiME[8] (Linux Memory Extractor) which is a Loadable Kernel Module for memory acquisition. We inserted the module into the running Kernel and acquired 2 GiB of the memory in raw format.

A. *Identify present Linux Kernel Version*

In our first application we identify the presence of Kernel fragments and the Kernel version of the target system by analyzing the acquired raw memory dump with MRSH-MEM. This process can support further structured analysis and possibly enhance the task of profile determination. In our application we utilize a set of available Linux images of a public Debian repository[9]. The Kernel files (i.e., vmlinux/vmlinuz) have been obtained by the corresponding deb-Packages, the .text sections have been extracted and the images have been processed with MRSH-MEM. We additionally store the extracted Linux Kernels and its corresponding chunk hash values in our introduced CHDB. Subsequently, we query the CHDB with 12 different Kernel images (see Table VII for an overview of all inserted Linux Kernels).

While we expect that most of the Linux Kernels from the repository share a reasonable amount of similar code chunks, this can obviously vary for different versions. To determine the actual Kernel version of our target system, we analyzed the detected chunks in two ways. First, we determined the total amount of detected chunks for each processed Kernel version. Second, we examined those chunks, which are only mapped to a *single* Kernel version by the CHDB and do not share *multiple* of those chunks with other Kernel versions.

After performing the step of chunk identification with MRSH-MEM, we additionally identified the related Kernel version(s) for each chunk. The amount and distribution of detected chunks by its corresponding kernel version(s) can be seen in Figure 8. The statically linked Kernel images share a reasonable amount of similar code fragments (bar *multiple*). However, the actual Kernel version clearly occupies most of the extracted chunks and thus, we could distinguish the present Kernel from the other images (see column (9) in Figure 8). Next, we only considered chunks which are mapped to a single Linux Kernel and do not count shared code fragments between different versions, i.e., we filter out identified chunks which are related to multiple Kernels. The examination of distinct mapped chunks in Figure 8 (bar *single*) underline the presence of our expected Kernel version (vmlinuz-3.16.0-4-amd64).

Considerations: Discussing the examination of the Ker-

[8] https://github.com/504ensicsLabs/LiME (last accessed 2018-02-10).

[9] http://ftp.us.debian.org/debian/pool/main/l/linux/ (last accessed 2018-02-10).

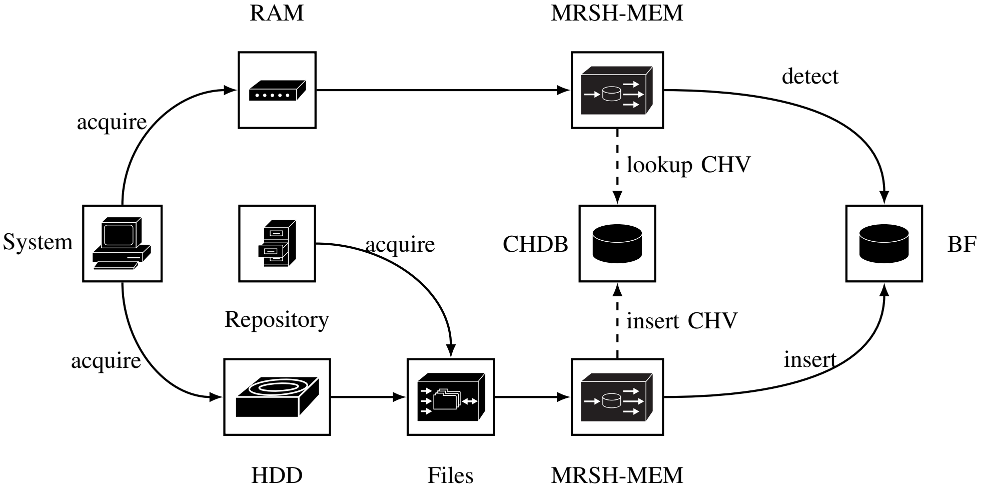

Figure 7. Overview of the application of `MRSH-MEM`.

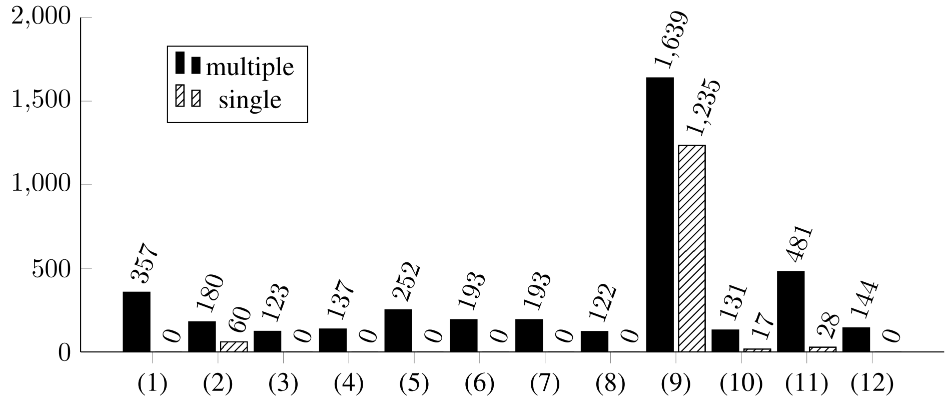

Figure 8. The detected chunk sequences and the overall counts for each Kernel version. As could be seen, the present Kernel version of our target system, i.e. vmlinux-3.2.0-4-amd64 (9), shows a significant amount of detected chunks.

nel `.text` section in memory leads to the question if `MRSH-MEM` can be used for detecting advanced Kernel infection techniques. Considering different hijacking techniques should lead to the presences of modifications in the memory located version of the original Kernel. However, the process of Kernel loading is quiet complex and the Linux Kernel binaries could additional contain modification instructions, i.e., alternative instructions, which patches the original code during loading (`.altinstructions`[10]). At this point we leave the question if `MRSH-MEM` is usable for advanced code integrity checks of Linux Kernels unanswered for further research.

B. *Identify Application in User Memory*

As already introduced in Section II, the Kernel memory mappings should be considered contiguous in most of the cases. To determine the capabilities of our approach in user space memory, we performed a task of process and application identification. We inspected the raw memory dump on the presences of application related code fragments. In detail, we acquired three different versions of the Wireshark Protocol Analyzer[11] from a Debian repository[12] (see Table VIII). The acquired ELFs were dynamically linked and stripped. We extracted the allocable `.text` sections of the different executables and processed them with `MRSH-MEM`, where each executable approximately contained 4130 chunks. Again, the chunks were also inserted into the CHDB for the evaluation of *single* and *multiple* hits.

We ensured that an instance of Wireshark 1.12.1 was running at the time of memory acquisition. Figure 9 illustrates

[10] https://lwn.net/Articles/531148/ (last accessed 2018-02-10).

[11] https://www.wireshark.org/ (last accessed 2018-02-10).

[12] http://ftp.us.debian.org/debian/pool/main/w/wireshark/ (last accessed 2018-02-10).

ID	Kernel	ID	Kernel	ID	Kernel
(1)	3.2.0-4-amd64	(2)	4.13.0-0.bpo.1-amd64	(3)	4.14.0-0.bpo.2-rt-amd64
(4)	4.14.0-0.bpo.3-amd64	(5)	3.2.0-4-rt-amd64	(6)	4.14.0-3-amd64
(7)	4.15.0-rc8-amd64	(8)	4.14.0-0.bpo.2-amd64	(9)	**3.16.0-4-amd64**
(10)	4.14.0-3-rt-amd64	(11)	3.16.0-0.bpo.4-amd64	(12)	4.14.0-0.bpo.3-rt-amd64

Table VII
EXTRACTED LINUX KERNEL IMAGES FROM THE DEBIAN REPOSITORY (MARKED WITH AN IDENTIFIER). THE ACTUAL PRESENT KERNEL IN THE EXTRACTED MEMORY IMAGE IS HIGHLIGHTED (9).

the capabilities of detecting and discriminating a running (or formerly running) application in memory, where the amount of single occupied chunks (1766) clearly identifies the actual running Wireshark version (1.12.1).

To investigate possible false positives and to examine the discrimination between a running and not running process we repeated the procedure after rebooting the system. Thus, we were not expecting to find presence of Wireshark. The results are shown in Figure 10 and indicate very low numbers / matches. Precisely, the bars show some hits in the case of multiple occupied chunks. To lower the values of false positives, we propose the adaptation and increase of the MIN_RUN parameter. We additional suggest a minimum required chunk size, as most of the false positives were smaller than 40 bytes.

C. Runtime performance

In the following paragraph we examine the runtime efficiency of MRSH-MEM. In detail, we measured the runtime for disassembling, chunk extraction, chunk hashing and Bloom filter handling. Note, we differentiate between Bloom filter creation and Bloom filter lookup. As mentioned in the original paper of approxis [20], the processed byte sequences can significantly influence the overall disassembling performance. Therefore, similar to Liebler and Baier [20] we study the runtime performance for three different images: a concatenated set of 64 bit ELF binaries, a raw memory dump acquired with LiME and a random sequence of bytes. Lastly, we removed all unnecessary functionalities (e.g., printout mechanisms) and compiled our binary with an optimization set to O2[13].

The efficiency test was performed on a Lenovo Thinkpad x250 with a Intel Core i5 2x 2,2 GHz and 8 GB RAM. The performance of the built in Solid State Drive was also determined, where the read performance was 508 MB/s and the write performance was 513 MB/s. The overall results are shown in Table IX. The column of chunks defines the amount of triggered chunk boundaries for each image and for one pass.

Considerations: The current implementation shows further potential for improving the overall runtime performance. So far, our current implementation does not consider any additional steps of previous data filtration steps (e.g. the usage of entropy analysis). In addition, it should be mentioned that the current processing does not consider any parallelization and the introduced approach empowers to concurrently process the input memory images.

V. RELATED WORK

The application of a YARA[14] rules for the examination of memory was recently discussed by Cohen [9]. The author described a context-aware scanning scheme on the physical address space using the Windows PFN database, which could be used to map each physical page to a corresponding process. By the examination of physical memory dumps, the approach still gains a reasonable performance, which is caused by an optimized IO throughput. In contrary to the application on hard disk, the authors discuss the applicability, expandability and the adaptations of pattern matching rules in the course of memory analysis.

White et al. [30] and Walters et al. [29] discussed identifying known code sequences by applying cryptographic hash functions. Therefore, memory pages containing code fragments are first normalized, before they are further processed. The offsets which have to be normalized have been saved into a database of hash templates; consisting of hash values and the corresponding offsets. After identifying a process in the Virtual Memory Space, the hash templates are selected by extracted process informations from the memory dump itself. White et al. [30] improved the costly lookup process, which was first introduced by Walters et al. [29], where the comparison between each template and each page in the Physical Address Space leads to a complexity of $O(n * m)$ for a comparison of n templates against m memory pages. A virtual PE Loader was created to perform the process of binary lifting and to extract the variable memory offsets in the executable. A public available Volatility plugin[15] provides a whitelisting similar to White et al. [30]. The approach performs a lookup on a page level of executables. Therefore, the memory is processed and sent to a hash server. In contrary to [30], the lifting of the code is performed on

[13] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html (last accessed 2018-02-10).

[14] https://github.com/VirusTotal/yara (last accessed 2018-02-10).

[15] https://github.com/K2/Scripting/blob/master/inVteroJitHash.py (last accessed 2018-02-10).

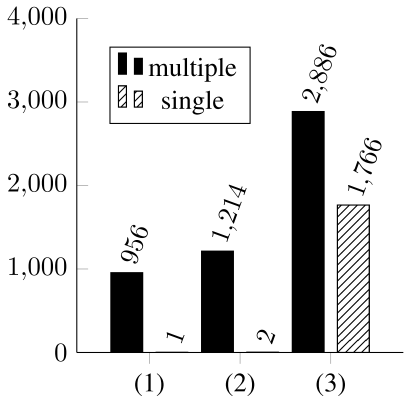

Figure 9. Examination of a memory dump of our target system meanwhile Wireshark was running (ELF executable amd64; version 1.12.1).

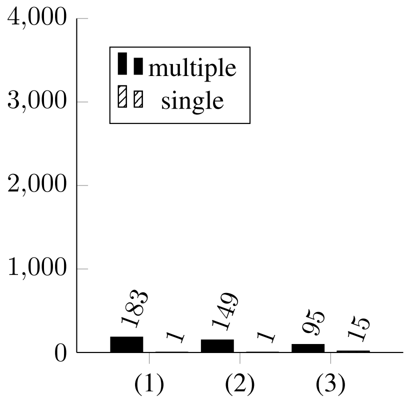

Figure 10. Memory dump of our target system after rebooting the virtual machine and thus, without a running Wireshark instance.

ID	Version
(1)	2.4.4-1_amd64
(2)	2.2.6+g32dac6a-2+deb9u2_amd64
(3)	**1.12.1+g01b65bf-4+deb8u13_amd64**

Table VIII
LIST OF EXTRACTED WIRESHARK VERSIONS. THE ACTUAL RUNNING VERSION IS HIGHLIGHTED (3).

Execution time		Chunks	Description
insert	lookup		
46.0s	48.0s	6,887,955	Concatenated set of 64bit binaries from `/usr/bin`
50.0s	50.0s	1,608,674	Raw memory dump acquired with `LiME`
197.0s	192.0s	10,537,710	Random sequences of bytes generated with `/dev/urandom`

Table IX
INSERT AND LOOKUP RUNTIME PERFORMANCE OF `MRSH-MEM` FOR DIFFERENT INPUT IMAGES.

the server, which creates integrity hashes with the help of the virtual address of the process on the client.

Ligh et al. [21] introduced a script (`ssdeep_procs`), which enumerates running processes on a system, dumps them to hard disk and compares the extracted executables with the help of `ssdeep` [19]. The Volatility plugin called `impfuzzy`[16] applies Fuzzy Hashing on the Import API of PE files to detect malicious changes. However, most of the mentioned approaches are context-aware and fall into the category of structured analysis where we have already mentioned the advantages, disadvantages and conceptional differences. In contrary to the introduced approaches, our approach aims to extend current techniques of unstructured analysis and the creation of new forms of data driven cross validation.

Different authors mentioned or questioned the application of approximate matching in the course of memory forensics [30, 22]. In the specific case of matching executables in memory to its counterpart loaded in memory, most of the authors doubt the usefulness of applying approximate matching on the raw sequences in memory. One major reason are legitimate changes to the code sequences caused by the loading process itself. Similar to White et al. [30] and Walters et al. [29] we propose an additional step of normalization by previously disassembling detected code sequences. This is accomplished with the utilization of an approximate disassembler [20].

The term approximate disassembling describes the rough disassembling of unknown code sequences and was, to the best of our knowledge, first mentioned by Shah [27]. The authors focused on a fast and rough disassembling of a code base as preceding step of machine learning based classification tasks. Therefore, they first analyzed the Most Frequent Occurred (MFO) instructions of executable files similar to Bilar [1]. By disassembling a large set of PE executables with the help of IDA Pro, a frequency analysis

[16] https://github.com/JPCERTCC/aa-tools/tree/master/impfuzzy (last accessed 2018-02-10).

of subsequent MFO instruction pairs was performed. The statistical examination was used for creating a solution tree, which could be used to select the most plausible next offsets inside a byte stream. The approach resolves multiple possible paths by examining the statistical probability of different byte offsets inside a byte stream. The approach was called to be 20 % faster than IDA Pro. However, no computational performance was made against a linear sweep disassembler, as those are known to be faster than a recursive traversal based disassembler like IDA Pro. Additionally, comparing the computational performance could be misleading, as IDA Pro outreaches the capabilities of approximate disassembling. The approach was only tested for small files with a size from one to five megabytes.

Considering the field of memory forensics and its introduced conditions, we inspected approaches that are related to our task of identifying fragmented code structures. Garfinkel and McCarrin [13] introduced an approach called *hash-based carving* which aims at identifying fragmented files, files that are incomplete, or files that have been partially modified. The approach is mainly considered in the field of sector-based volumes and the sliding window based extraction is sized to 4 KiB. The overall process is computational demanding but highly parallelizable. The authors make use of a previously introduced hashdb [31] and outline their real-world experience by the utilization of hash-based carving. A major contribution of the work is the discussion of classifying blocks and the negative impact of common blocks, which are shared between documents. Their work shows the problem of a high false identification rate caused by large amounts of shared blocks within the processed document classes. As our work focuses on the adaption of approximate matching into the field of memory forensics and as we propose a more context-related extraction of chunks by adopting fundamentals of an implementation called `MRSH-NET` [4, 5], we address the process of a better chunk identification and resolving a corresponding executables as a future task.

VI. Conclusion

Approximate matching techniques are known among the digital forensics community and have been utilized in different fields of application. Current implementations empower to whitelist or identify fragments of data in the field of classical disk or network forensics. The application of approximate matching on memory bares several pitfalls. Similar to other unstructured analysis techniques, our introduced approach has to consider several idiosyncrasies of inspecting the physical memory space. Therefore, we first discussed those considerations and limitations.

We introduced a new specimen of approximate matching (`MRSH-MEM`), by interfacing approximate matching (`MRSH-NET`) with an additional step of approximate disassembling (`approxis`). We described the implementation details of merging both techniques as well as the needed adaptations and changed parameter settings. The integration of an additional step of disassembling stabilizes the original bytewise application, as our approach works on disassembled and thus, normalized instruction sequences. To the best of our knowledge, this is the first implementation, which interfaces approximate matching with an additional step of approximate disassembling.

We showed the feasibility of our approach by comparing a memory dump against code fragments gained from different resources, i.e., code extracted from a hard disk as well as a repository. Our introduced approach detects allocable code fragments without the need of manual inspecting an executable or the manual definition of any matching rules. Our first prototype empowers to easily create a database of different applications and perform the examination of raw memory dumps. As former publications claimed approximate matching to be slow, we showed that our current prototype achieves a good computational performance, without the usage of any parallelization.

Our current implementation, which is written in C, could be shipped as a Python Extension for easy use and integration. We consider our implementation in different fields of application and see further potential of extending existing Memory Forensics Frameworks, which are also mostly written in Python.

VII. Future work

There are four major aspects we will address. First, the process of saving and identifying files will be improved. The current implementation of the chunk hash database (CHDB) and the usage of a single Bloom filter shows promising results. However, an advanced database should be considered, which empowers to actually identify and name a specific fragment found in memory. A general discussion of this task was already started by Garfinkel and McCarrin [13] and the authors showed major challenges caused by shared blocks across different samples of a specific file type, e.g., office documents. As we process executable sections of code, different results should be expected which requires the reassessment of a chunk based file identification in our specific field of application. A promising candidate for further developments could be the adaptation and integration of Hierarchical Bloom Filter Trees (HBFT) [23] or the utilization of already introduced hash databases [31].

Second, the process of chunk hashing should be further investigated. The extraction of chunks based on mnemonic representatives or the original byte sequences itself, could be further extended by other types of intermediate representations. This should allow to perform additional steps of investigation, as the current approximated state of disassembling is useful for the fast detection of similarities but gives less insights into the performed instructions and their meaning.

Third, the core of MRSH-MEM is written in C to gain the best runtime performance. However, an integration into one of the public available memory forensics frameworks is possible, as we propose an integration as Python Extension. Further applications and the creation of new extensions will be considered, e.g., the development of advanced integrity checks. Our approach may also be used to extend current frameworks and the process of structural analysis. A first application could be the integration of data driven cross validation functionalities, e.g., by enumerating an identified process by its original and disk-placed disassembled codebase. As already outlined, the task of code integrity checks is an interesting application, e.g., for advanced Kernel integrity checks. However, this application needs additional research and engineering effort.

Finally, as our current approach only focuses on code in memory, we will investigate the capabilities of identifying non-code related structures. As we perform a single pass over the complete image, this should not lead to a significant runtime overhead.

Acknowledgment

This work was supported by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP (www.crisp-da.de).

References

[1] D. Bilar, "Statistical structures: Fingerprinting malware for classification and analysis," *Proceedings of Black Hat Federal 2006*, 2006.

[2] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, pp. 422–426, 1970.

[3] F. Breitinger, "On the utility of bytewise approximate matching in computer science with a special focus on digital forensics investigations," Ph.D. dissertation, Technical University Darmstadt, 2014.

[4] F. Breitinger and I. Baggili, "File detection on network traffic using approximate matching," *The Journal of Digital Forensics, Security and Law: JDFSL*, vol. 9, no. 2, p. 23, 2014.

[5] F. Breitinger and H. Baier, "Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2," in *International Conference on Digital Forensics and Cyber Crime*. Springer, 2012, pp. 167–182.

[6] F. Breitinger, H. Baier, and D. White, "On the database lookup problem of approximate matching," *Digital Investigation*, vol. 11, pp. S1–S9, 2014.

[7] A. Case and G. G. Richard, "Memory forensics: The path forward," *Digital Investigation*, vol. 20, pp. 23–33, 2017.

[8] C. Cohen and J. S. Havrilla, "Function hashing for malicious code analysis," *CERT Research Annual Report*, pp. 26–29, 2009.

[9] M. Cohen, "Scanning memory with yara," *Digital Investigation*, vol. 20, pp. 34–43, 2017.

[10] M. R. Farhadi, B. C. Fung, P. Charland, and M. Debbabi, "Binclone: Detecting code clones in malware," in *Software Security and Reliability (SERE), 2014 Eighth International Conference on*. IEEE, 2014, pp. 78–87.

[11] M. R. Farhadi, B. C. Fung, Y. B. Fung, P. Charland, S. Preda, and M. Debbabi, "Scalable code clone search for malware analysis," *Digital Investigation*, vol. 15, pp. 46 – 60, 2015, special Issue: Big Data and Intelligent Data Analysis.

[12] G. Fowler, L. C. Noll, K.-P. Vo, D. Eastlake, and T. Hansen, "The fnv non-cryptographic hash algorithm," *IETF-draft*, 2011.

[13] S. L. Garfinkel and M. McCarrin, "Hash-based carving: Searching media for complete files and file fragments with sector hashing and hashdb," *Digital Investigation*, vol. 14, pp. S95–S105, 2015.

[14] V. S. Harichandran, F. Breitinger, and I. Baggili, "Bytewise approximate matching: The good, the bad, and the unknown," *The Journal of Digital Forensics, Security and Law: JDFSL*, vol. 11, no. 2, p. 59, 2016.

[15] J. Jang, D. Brumley, and S. Venkataraman, "Bitshred: Fast, scalable malware triage," *Cylab, Carnegie Mellon University, Pittsburgh, PA, Technical Report CMU-Cylab-10*, vol. 22, 2010.

[16] ——, "Bitshred: feature hashing malware for scalable triage and semantic analysis," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 309–320.

[17] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan, "Binary function clustering using semantic hashes," in *Proceedings of the 2012 11th International Conference on Machine Learning and Applications - Volume 01*, ser. ICMLA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 386–391.

[18] A. Karnik, S. Goswami, and R. Guha, "Detecting obfuscated viruses using cosine similarity analysis," in *Modelling & Simulation, 2007. AMS'07. First Asia International Conference on*. IEEE, 2007, pp. 165–170.

[19] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digit. Investig.*, vol. 3, pp. 91–97, September 2006.

[20] L. Liebler and H. Baier, "Approxis: A fast, robust, lightweight and approximate disassembler considered in the field of memory forensics," in *International Conference on Digital Forensics and Cyber Crime*. Springer, 2017, pp. 158–172.

[21] M. Ligh, S. Adair, B. Hartstein, and M. Richard,

Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code. Wiley Publishing, 2010.

[22] M. H. Ligh, A. Case, J. Levy, and A. Walters, *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons, 2014.

[23] D. Lillis, F. Breitinger, and M. Scanlon, "Expediting mrsh-v2 approximate matching with hierarchical bloom filter trees," in *International Conference on Digital Forensics and Cyber Crime*. Springer, 2017, pp. 144–157.

[24] E. Raff and C. Nicholas, "An alternative to ncd for large sequences, lempel-ziv jaccard distance," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17. New York, NY, USA: ACM, 2017, pp. 1007–1015. [Online]. Available: http://doi.acm.org/10.1145/3097983.3098111

[25] ——, "Lempel-ziv jaccard distance, an effective alternative to ssdeep and sdhash," *Digital Investigation*, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287617302566

[26] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queens School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[27] A. Shah, "Approximate disassembly using dynamic programming. masters report, department of computer science, san jose state university," 2010.

[28] A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia, "A.: Exploiting similarity between variants to defeat malware: vilo method for comparing and searching binary programs," in *In: Proceedings of BlackHat DC 2007. (2007)*, 2007. [Online]. Available: https://blackhat.com/presentations/bh-dc-07/Walenstein/Paper/bh-dc-07-walenstein-WP.pdf

[29] A. Walters, B. Matheny, and D. White, "Using hashing to improve volatile memory forensic analysis," in *American Acadaemy of forensic sciences annual meeting*, 2008.

[30] A. White, B. Schatz, and E. Foo, "Integrity verification of user space code," *Digit. Investig.*, vol. 10, pp. S59–S68, August 2013.

[31] J. Young, K. Foster, S. Garfinkel, and K. Fairbanks, "Distinct sector hashes for target file detection," *Computer*, vol. 45, no. 12, pp. 28–35, 2012.

[32] V. Zwanger, E. Gerhards-Padilla, and M. Meier, "Codescanner: Detecting (hidden) x86/x64 code in arbitrary files," in *Malicious and Unwanted Software: The Americas (MALWARE), 2014 9th International Conference on*. IEEE, 2014, pp. 118–127.