MRSH-MEM:
Approximate Matching on Raw Memory Dumps

Lorenz Liebler †, Frank Breitinger ‡

† University of Applied Sciences Darmstadt, Germany, da/sec
Biometrics and Internet-Security Research Group

‡ University of New Haven USA, UNHcFREG
Cyber Forensics Research and Education Group

IMF 2018 - Hamburg, 2018-05-08
## Introduction

### Memory Analysis

#### Interpretation of Structures

Framework interprets the complex system related structures, where Profiles interface images (Rekall/Volatility):

- formats of acquisition
- memory management
- underlying architecture
- OS meta structures
- different versions

#### Memory Carving

Unstructured analysis extract content information out of memory dumps:

- string extraction
- file carver
- signature matching (YARA)
## Memory Analysis

<table>
<thead>
<tr>
<th>Interpretation of Structures</th>
<th>Memory Carving</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ detailed examination of manifold information</td>
<td>+ straight forward application</td>
</tr>
<tr>
<td>+ cross validation tasks</td>
<td>+ not reliant on OS related structures</td>
</tr>
<tr>
<td>- needs domain knowledge for application</td>
<td>- less insights and not so powerful</td>
</tr>
<tr>
<td>- needs maintenance; understand and implement OS in framework</td>
<td>- carving approach for specific examination</td>
</tr>
</tbody>
</table>

- needs domain knowledge for application
- needs maintenance; understand and implement OS in framework
- less insights and not so powerful
- carving approach for specific examination
Motivation of Memory Carving

1. Extend analysis by data-driven **cross validation**
   (e.g. avoid OS-structure based analysis)

2. Open new possibilities to counter **anti-forensics**
   (e.g. Williams and Torres [8]: irrelevant and non-existing meta structures)

3. Need **fast data reduction** methods similar to disk forensics
   (e.g. for whitelisting known or blacklisting malicious code)

4. Methods for **first or last resort of interpretation**
   (e.g. no adequate / matching profiles; missing patches)
Memory Carving - Code

- special focus on examination of code-related structures
  - Whitelisting of benign code
  - Blacklisting of malicious code

- **Loading executables could lead to major manipulations:**
  ELF/PE loader, offset patching, base relocations, page alignment, alternative instructions, ...
Memory Management

Beside the adaptations during loading, we should consider:

1. virtually contiguous $\neq$ physically contiguous
2. page size and page alignment could vary
3. memory shared between processes
4. not able to resolve virtual address without context
5. memory could be swapped to disk
Related Approaches

**Code integrity in memory - White et al. [7]**

based on Walters et al. [6]

- Creates Hash-Templates of previously **normalized pages** (Hash-Templates are offsets + hash value)
- Imitates loading by a Virtual PE Loader
- Based on process identification (Filename)
Related Approaches

Practical realization similar to White et al. [7]

inVteroJitHash

- Forensics, Memory integrity and assurance tool
- Server-based PE integrity hash database
- Send loading address and hash to server
- **Lifting** of the binaries and hashing on server side
- BlackHat USA ’17
Related Approaches

Summarized

- Most of the previous approaches rely on structural examinations and are process-context aware:
  - Process enumeration / reconstruction
  - Process identification
  - Code normalization/lifting
  - Integrity check (data reduction)

- We want to **carve code** in memory dumps without recreating a process context.

- Could we utilize Approximate Matching for this task?
MRSH Family [2, 3, 4]

- Sliding window rolls through byte sequence
- PRF defines chunk boundaries
- CHF compress the chunk
- MRSH-NET saves chunk in a single large Bloom filter (Hamming distance)
Memory forensics - impracticability

- **Bytewise** Approximate Matching respects every change in the underlying byte structure versus *mutability of code* in memory

- Influences **Chunk Extraction** (PRF)
- Influences **Chunk Hashing** (CHF)
- Influences **Similarity Digest** itself

- We need an additional layer of *normalization* similar to Walters et al. [6] and White et al. [7]
**Motivation**

1. **Detect** sequences of code within raw bytes
2. **Normalize** detected code by disassembling

→ apply Approximate Matching on disassembled instructions

**Definition:** Approximate Disassembling should not provide a full decoding of the x86 complex instruction set. We decode for each instruction a representing mnemonic and length.

<table>
<thead>
<tr>
<th>Raw bytes</th>
<th>Mnemonic + Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>41 55</td>
<td>push 2</td>
</tr>
<tr>
<td>48 89 f3</td>
<td>mov 3</td>
</tr>
<tr>
<td>48 81 ec</td>
<td>sub 3</td>
</tr>
</tbody>
</table>
Approximate Disassembling

Classes of Disassemblers

- Disassembler for unknown x86/x64 instruction sequences
- Focuses on computational efficiency
- Discriminate code from data

<table>
<thead>
<tr>
<th>Decoding</th>
<th>Length Disas.</th>
<th>Approximate Disas.</th>
<th>Linear Sweep</th>
<th>Recursive Traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Length</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Linearity</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Code Detection</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Interpretation</td>
<td>Bit</td>
<td>Byte</td>
<td>Bit</td>
<td>Bit</td>
</tr>
</tbody>
</table>
approxis [5] - Disassembling

Example:
Simplified x64 instruction set!

- Build prefix-tree from a set of ground truth assemblies obtained by Andriesse et al. [1]
- Stay on a byte-level during disassembling; traverse tree
approxis [5] - Disassembling

Interpret the raw byte sequence with the generated prefix tree.

```
41 55 48 89 f3
48 81 ec 48 8d
64 48 8b
```

```
push 41 55
mov 48 89 f3
sub 48 81 ec
lea 48 8d
mov 64 48 8b
```
approxis [5] - Code Confidence

Mnemonic bigram frequencies as absolute logits: \[ \lambda = \left| \ln \frac{p}{1-p} \right| \]

- Interleaved 32 and 64 bit binaries into block of random data
- $\omega_x$ describes average confidence of current window at offset $x$
approxis [5] - Computational Performance

- Created three images with a size of 2 GiB
- Reduced diStorm: no output, large buffer, full decoding

<table>
<thead>
<tr>
<th></th>
<th>approxis</th>
<th></th>
<th></th>
<th>disassembler mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32</td>
<td>64</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Execution time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>29.084s</td>
<td>21.936s</td>
<td>1m20.770s</td>
<td>1m7.772s</td>
</tr>
<tr>
<td></td>
<td>27.859s</td>
<td>31.918s</td>
<td>1m43.999s</td>
<td>1m43.046s</td>
</tr>
<tr>
<td></td>
<td>1m15.521s</td>
<td>1m44.990s</td>
<td>1m58.278s</td>
<td>1m56.192s</td>
</tr>
<tr>
<td>Description</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64bit binaries from /usr/bin</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Raw memory dump (LiME)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Random sequences (/dev/urandom)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Approach: MRSH-MEM**

**Concept**

- MRSH-MEM: integration of approxis into MRSH-NET
- Focus on computational efficiency
- From **Bytewise** to **Mnemonic-wise** Approximate Matching

Diagram:

- Disk Sample to Disk Digest
- Memory Image to Memory Digest
- Data Processing
- Compare

**Memory**

**Data Processing**
Approach: MRSH-MEM

MRSH-MEM - Processing Pipeline

1. [approxis] approximate disassemble
2. [approxis] determine confidence
3. [MRSH] determine chunks (apply PRF)
4. [approxis/MRSH] remove irrelevant chunks
5. [MRSH] hash chunks (apply CHF)
Approach: MRSH–MEM

MRSH–MEM - Processing Pipeline

1. [approxisis] approximate disassemble
2. [approxisis] determine confidence
3. [MRSH] determine chunks (apply PRF)
4. [approxisis/MRSH] remove irrelevant chunks
5. [MRSH] hash chunks (apply CHF)

<table>
<thead>
<tr>
<th>raw bytes</th>
<th>menmonics</th>
<th>confidence</th>
<th>chunks</th>
<th>code chunks</th>
<th>chunk hashes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00</td>
<td>000 : 00</td>
<td>000 : 64</td>
<td>000 : 64</td>
<td>000 : C₁</td>
<td>009 : C₂</td>
</tr>
<tr>
<td>31 ed 49</td>
<td>000 : 00</td>
<td>000 : 64</td>
<td>000 : 64</td>
<td>000 : C₁</td>
<td>005 : C₂</td>
</tr>
<tr>
<td>89 d1 5e</td>
<td>000 : 00</td>
<td>000 : 63</td>
<td>000 : 63</td>
<td>000 : C₁</td>
<td>105 : C₂</td>
</tr>
<tr>
<td>48 89 e2</td>
<td>092 : 31 ed</td>
<td>092 : 12</td>
<td>092 : C₂</td>
<td>092 : C₂</td>
<td>095 : C₂</td>
</tr>
<tr>
<td>48 83 e4</td>
<td>095 : 49 89 d1</td>
<td>095 : 09</td>
<td>095 : C₂</td>
<td>095 : C₂</td>
<td>095 : C₂</td>
</tr>
<tr>
<td>f0 00 00</td>
<td>105 : 5e</td>
<td>105 : 11</td>
<td>105 : C₂</td>
<td>105 : C₂</td>
<td>095 : C₂ [5AC]</td>
</tr>
<tr>
<td>00 00 00</td>
<td>095 : 48 89 e2</td>
<td>095 : 10</td>
<td>095 : C₂</td>
<td>095 : C₂ [1]</td>
<td></td>
</tr>
<tr>
<td>090 : 48 83 e4 f0</td>
<td>090 : 10</td>
<td>090 : 10</td>
<td>090 : C₃</td>
<td>090 : C₃</td>
<td></td>
</tr>
<tr>
<td>000 : 00</td>
<td>000 : 00</td>
<td>000 : 64</td>
<td>000 : 64</td>
<td>000 : C₃</td>
<td>000 : C₃</td>
</tr>
<tr>
<td>000 : 00</td>
<td>000 : 00</td>
<td>000 : 64</td>
<td>000 : 64</td>
<td>000 : C₃</td>
<td>000 : C₃ [0]</td>
</tr>
</tbody>
</table>
MRSH-MEM - Technical Details

- Detailed example in the paper
- Strongly interleaved implementation
- Usage of **multiple buffers**, e.g.:
  1. Raw byte buffer
  2. Integerized mnemonic buffer
  3. Relative offset buffer
     ...

- Usage of **multiple parameters**, e.g.:
  1. Block size
  2. Code confidence threshold
  3. Code coverage per block
     ...

Concept

- **MRSH-MEM** uses a single, large Bloom filter → disadvantage: **Lack of file identification**: 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.

**temporal solution CHDB:**
- database of extracted chunk hash values (CHV)
- chunk hash database (CHDB) consists of single lookup tree
- each leaf node with corresponding file name(s)
Considerations:
Discussing the examination of the Kernel section in memory leads to the question if MRSH-MEM can be used for detecting advanced Kernel infection techniques. 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 (.altinstructions). Those instructions patch the original code during loading. 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 from a Debian repository (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...

9https://lwn.net/Articles/531148/(last accessed 2018-02-10).

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 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 the plot indicates 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 [21], the processed byte sequences can significantly influence the overall disassembling performance. Therefore, similar to Liebler and Baier [21] 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...
Target System

- Debian 8 installation (Debian 3.16.7 x86 64 GNU/Linux)
- Virtual Box (Version 5.2.6 r120293)
- Network analysis tasks
- Acquire dump with LiME 7 (Linux Memory Extractor)
Examination 1) Kernel Version

- Determine the running kernel version of an acquired dump
- Extracted 12 Linux Kernel images from the Debian repository
- Present Kernel: **3.16.0-4-amd64** (9)

<table>
<thead>
<tr>
<th>ID</th>
<th>Kernel</th>
<th>ID</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>3.2.0-4-amd64</td>
<td>(2)</td>
<td>4.13.0-0.bpo.1-amd64</td>
</tr>
<tr>
<td>(3)</td>
<td>4.14.0-0.bpo.2-rt-amd64</td>
<td>(4)</td>
<td>4.14.0-0.bpo.3-amd64</td>
</tr>
<tr>
<td>(5)</td>
<td>3.2.0-4-rt-amd64</td>
<td>(6)</td>
<td>4.14.0-3-amd64</td>
</tr>
<tr>
<td>(7)</td>
<td>4.15.0-rc8-amd64</td>
<td>(8)</td>
<td>4.14.0-0.bpo.2-amd64</td>
</tr>
<tr>
<td>(9)</td>
<td><strong>3.16.0-4-amd64</strong></td>
<td>(10)</td>
<td>4.14.0-3-rt-amd64</td>
</tr>
<tr>
<td>(11)</td>
<td>3.16.0-0.bpo.4-amd64</td>
<td>(12)</td>
<td>4.14.0-0.bpo.3-rt-amd64</td>
</tr>
</tbody>
</table>
Examination 1) Kernel Version

- single hits clearly identify correct running kernel version
Examination 2) Running Application

<table>
<thead>
<tr>
<th>ID</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>2.4.4-1_amd64</td>
</tr>
<tr>
<td>(2)</td>
<td>2.2.6*_amd64</td>
</tr>
<tr>
<td>(3)</td>
<td>1.12.1*_amd64</td>
</tr>
</tbody>
</table>

- Acquired two memory dumps of target system with running and without running Wireshark instance.
## Runtime Performance

<table>
<thead>
<tr>
<th>Execution time</th>
<th>Chunks</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert 46.0s</td>
<td>lookup 48.0s</td>
<td>6,887,955</td>
</tr>
<tr>
<td>50.0s</td>
<td>50.0s</td>
<td>1,608,674</td>
</tr>
<tr>
<td>197.0s</td>
<td>192.0s</td>
<td>10,537,710</td>
</tr>
</tbody>
</table>

- Intel(R) Core(TM) i5-3570K CPU @ 3.40GHZ, 16 GiB DDR3 RAM (1333 MHz) and 6 MiB L3 cache
- Prototype in C (-03)
- Created three images with a size of 2 GiB
- 64 bit case; Bloom filter only
Conclusion

- Discuss the considerations and limitations by applying Approximate Matching on code located in memory
- Introduced a new specimen of Approximate Matching: MRSH-MEM
- Demonstrated a first use case by comparing a memory dump with code fragments of different resources

- More details given in our paper
- Release prototype
  https://github.com/dasec/approximate-memory

lorenz.liebler@h-da.de
https://dasec.h-da.de/staff/lorenz-liebler/
Future Wok

1. Database Lookup Problem (CHDB replacement)
2. Better verification (Synthetic Carving Images)
3. Extend by Windows-based analysis (in 2018)
4. Integration into framework-based analysis (e.g. as plugin for Volatitliy, Rekall)
Bibliography I


