# `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

# Memory Analysis

| Interpretation of Structures | Memory Carving |
|---|---|
| Framework interprets the complex system related structures, where Profiles interface images (Rekall/Volatility):<br>▶ formats of acquisition<br>▶ memory management<br>▶ underlying architecture<br>▶ OS meta structures<br>▶ different versions | Unstructured analysis extract content information out of memory dumps:<br>▶ string extraction<br>▶ file carver<br>▶ signature matching (YARA) |

# Memory Analysis

| Interpretation of Structures | Memory Carving |
|---|---|
| + detailed examination of manifold information | + straight forward application |
| + cross validation tasks | + not reliant on OS related structures |
| - needs domain knowledge for application | - less insights and not so powerful |
| - needs maintance; understand and implement OS in framework | - 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

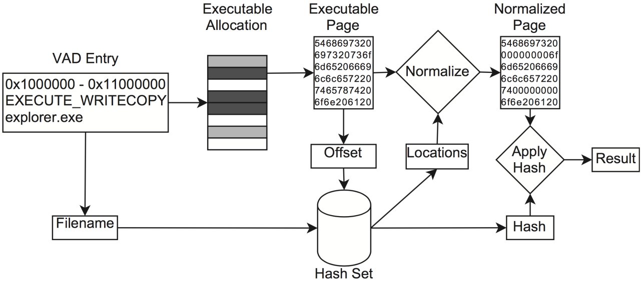
Beside the adaptations during loading, we should consider:

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

# 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)



# Practical realization similar to White et al. [7]

**inVteroJitHash**

`https://github.com/K2/Scripting/blob/master/inVteroJitHash.py`

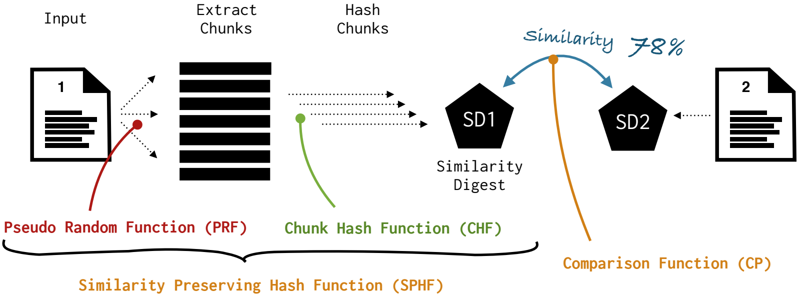
- Forensics, Memory integrity and assurance tool
- Server-based PE integrity hash database
- Send loading address and hash to server
- **Lifting** of the binaries and hashing on server side
- BlackHat USA ’17

# 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

ϟ Influences **Chunk Extraction** (PRF)
ϟ 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.

Raw bytes

```
41 55
48 89 f3
48 81 ec
```

→

Mnemonic + Length

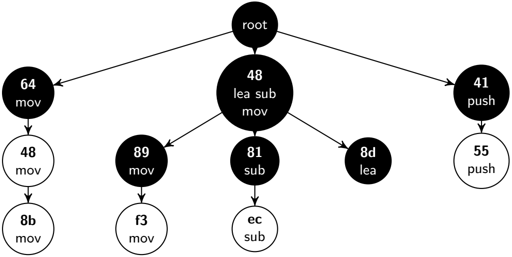
```
push  2
mov   3
sub   3
```

# Classes of Disassemblers

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

| Decoding | Length Disas. | Approximate Disas. | Linear Sweep | Recursive Traversal |
|---|---|---|---|---|
| Full | ✗ | ✗ | ✓ | ✓ |
| Mnemonic | ✗ | ✓ | ✓ | ✓ |
| Length | ✓ | ✓ | ✓ | ✓ |
| Linearity | ✓ | ✓ | ✓ | ✗ |
| Code Detection | - | ✓ | - | - |
| Interpretation | Bit | Byte | Bit | Bit |

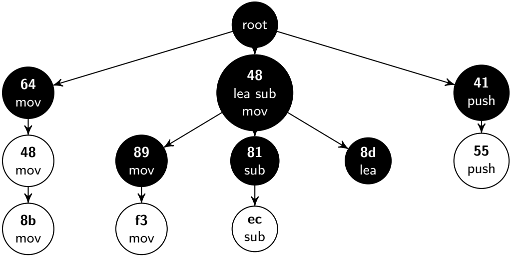
# `approxis` [5] - Disassembling



**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.*

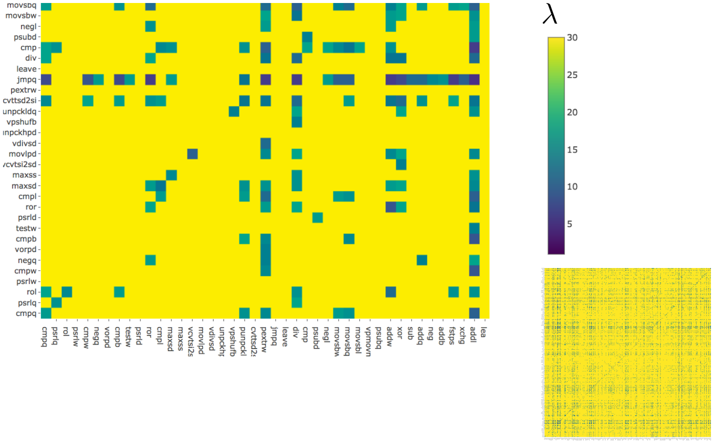
```
41 55 48 89 f3
48 81 ec 48 8d
64 48 8b
```

→

```
push  41 55
mov   48 89 f3
sub   48 81 ec
lea   48 8d
mov   64 48 8b
```

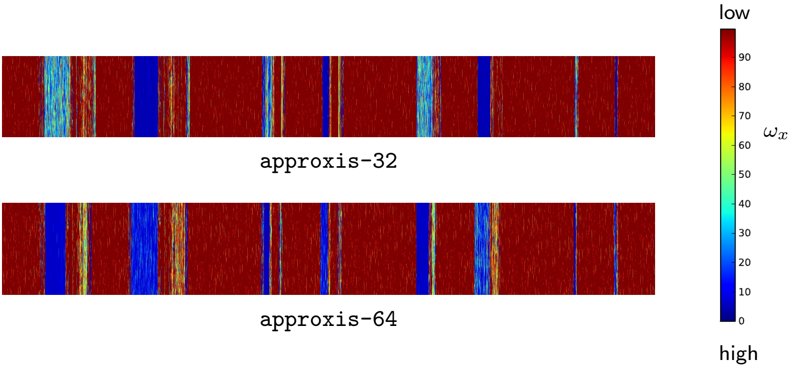
# `approxis` [5] - Code Confidence

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



# `approxis` [5] - Code Detection

- 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

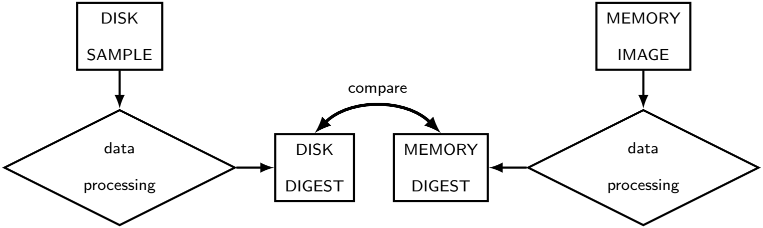
- Created three images with a size of 2 `GiB`
- Reduced diStorm: no output, large buffer, full decoding

| Execution time | | | | Description |
|---|---|---|---|---|
| approxis | | diStorm | | disassembler |
| 32 | 64 | 32 | 64 | mode |
| 29.084s | 21.936s | 1m20.770s | 1m7.772s | 64bit binaries from `/usr/bin` |
| 27.859s | 31.918s | 1m43.999s | 1m43.046s | Raw memory dump (`LiME`) |
| 1m15.521s | 1m44.990s | 1m58.278s | 1m56.192s | Random sequences (`/dev/urandom`) |

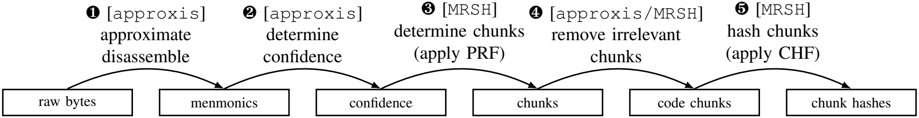
# Approach: `MRSH-MEM`

## Concept

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



# `MRSH-MEM` - Processing Pipeline



❶ [`approxis`] approximate disassemble

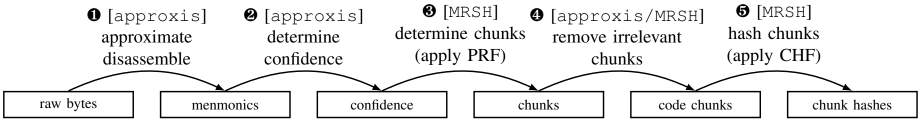
❷ [`approxis`] determine confidence

❸ [`MRSH`] determine chunks (apply PRF)

❹ [`approxis/MRSH`] remove irrelevant chunks

❺ [`MRSH`] hash chunks (apply CHF)

# MRSH-MEM - Processing Pipeline



❶ [approxis] approximate disassemble

❷ [approxis] determine confidence

❸ [MRSH] determine chunks (apply PRF)

❹ [approxis/MRSH] remove irrelevant chunks

❺ [MRSH] hash chunks (apply CHF)

| Byte |
|---|
| 00 00 00 |
| 31 ed 49 |
| 89 d1 5e |
| 48 89 e2 |
| 48 83 e4 |
| f0 00 00 |
| 00 00 00 |

| MNE | : | Byte |
|---|---|---|
| 000 | : | 00 |
| 000 | : | 00 |
| 000 | : | 00 |
| 092 | : | 31 ed |
| 095 | : | 49 89 d1 |
| 105 | : | 5e |
| 095 | : | 48 89 e2 |
| 090 | : | 48 83 e4 f0 |
| 000 | : | 00 |
| 000 | : | 00 |
| 000 | : | 00 |

| MNE | : | $\lambda$ |
|---|---|---|
| 000 | : | 64 |
| 000 | : | 64 |
| 000 | : | 63 |
| 092 | : | 12 |
| 095 | : | 09 |
| 105 | : | 11 |
| 095 | : | 10 |
| 090 | : | 10 |
| 000 | : | 64 |
| 000 | : | 64 |
| 000 | : | 64 |

| MNE | : | Chunk |
|---|---|---|
| 000 | : | $C_1$ |
| 000 | : | $C_1$ |
| 000 | : | $C_1$ |
| 092 | : | $C_2$ |
| 095 | : | $C_2$ |
| 105 | : | $C_2$ |
| 095 | : | $C_2$ |
| 090 | : | $C_3$ |
| 000 | : | $C_3$ |
| 000 | : | $C_3$ |
| 000 | : | $C_3$ |

| MNE | : | Chunk [CF] |
|---|---|---|
| 000 | : | $C_1$ |
| 000 | : | $C_1$ |
| 000 | : | $C_1$ [0] |
| 092 | : | $C_2$ |
| 095 | : | $C_2$ |
| 105 | : | $C_2$ |
| 095 | : | $C_2$ [1] |
| 090 | : | $C_3$ |
| 000 | : | $C_3$ |
| 000 | : | $C_3$ |
| 000 | : | $C_3$ [0] |

| Chunk | : | Hash |
|---|---|---|
| 092 | : | $C_2$ |
| 095 | : | $C_2$ |
| 105 | : | $C_2$ |
| 095 | : | $C_2$ [5AC] |

# `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

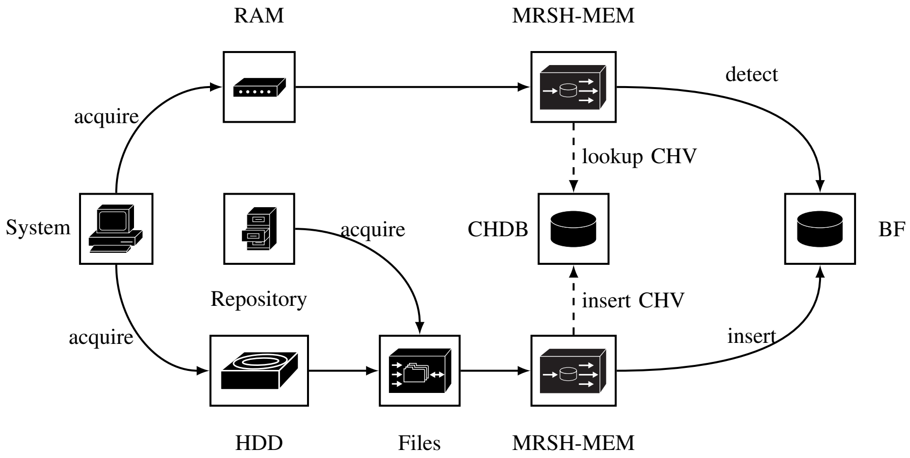
    ...

# 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)

## Concept Overview



# Target System

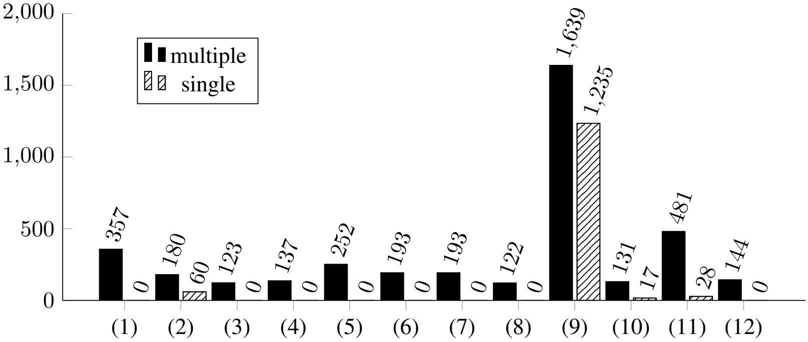
- Debian 8 installation (Debian 3.16.7 x86 64 GNU/Linux)
- Virtual Box (Version 5.2.6 r120293)
- Network analysis tasks
- Acquire dump with LiME 7 (Linux Memory Extractor)

# 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)

| 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 |

# Examination 1) Kernel Version

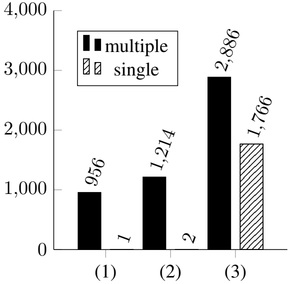


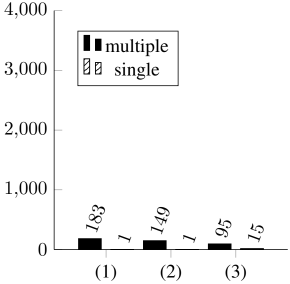
- single hits clearly identify correct running kernel version

# Examination 2) Running Application

| ID | Version | ID | Version | ID | Version |
|---|---|---|---|---|---|
| (1) | 2.4.4-1_amd64 | (2) | 2.2.6*_amd64 | (3) | **1.12.1*_amd64** |

- Acquired two memory dumps of target system **with** running and **without** running Wireshark instance





# Runtime Performance

| 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` |

- `Intel(R) Core(TM) i5-3570K CPU @ 3.40GHz`, `16 GiB DDR3 RAM (1333 MHz)` and `6 MiB L3` cache
- Prototype in `C` (`-O3`)
- 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

[1] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *USENIX Security Symposium*, pages 583–600, 2016.

[2] Frank Breitinger and Ibrahim Baggili. File detection on network traffic using approximate matching. *The Journal of Digital Forensics, Security and Law: JDFSL*, 9(2):23, 2014.

[3] Frank Breitinger and Harald Baier. Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2. In *International Conference on Digital Forensics and Cyber Crime*, pages 167–182. Springer, 2012.

[4] Vikram S Harichandran, Frank Breitinger, and Ibrahim Baggili. Bytewise approximate matching: The good, the bad, and the unknown. *The Journal of Digital Forensics, Security and Law: JDFSL*, 11(2):59, 2016.

[5] Lorenz Liebler and Harald 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*, pages 158–172. Springer, 2017.

[6] A Walters, Blake Matheny, and Doug White. Using hashing to improve volatile memory forensic analysis. In *American Acadaemy of forensic sciences annual meeting*, 2008.

[7] Andrew White, Bradley Schatz, and Ernest Foo. Integrity verification of user space code. *Digital Investigation*, 10:S59–S68, 2013.

[8] Jake Williams and Alissa Torres. Add-complicating memory forensics through memory disarray. *ShmooCon, Jan*, 2014.