- Approxis Memory Carving -

Approximate Disassembling and Matching on Memory Dumps
submitted to the *Volatility Plugin Contest 2018.*

Lorenz Liebler (✉`lorenz.liebler@h-da.de`, `@kn000x`)
Patrick Schmitt and Harald Baier

In the following document we roughly describe the details of our submission for the *Volatility Plugin Contest 2018.* We shortly describe some background, important notes for the reviewers and additionally give a short overview of the submission.

Contents

1 Motivation and Background

The application of fuzzy hashing algorithms is often mentioned in the context of memory forensics, but the applicability is not fully decided: "Although it **may be possible** to compare fuzzy hashes (e.g., the percentage similarity), cryptographic hashes will never match because of the following" [2].
First, we investigated the capabilities of *mrsh-v2* in the course of memory analysis. Mrsh-v2 is an improved variant of the original mrsh implementation. Both algorithms are a specimen of *Context-Triggered Piecewise-Hashing* and a subtype of approximate matching (a.k.a. fuzzy hashing or similarity hashing). We outlined several obvious and advanced pitfalls, similar to the subdomain of binary matching or analysis. An oversimplified overview of Context-Triggered Piecewise-Hashing is given in Figure 1.1.

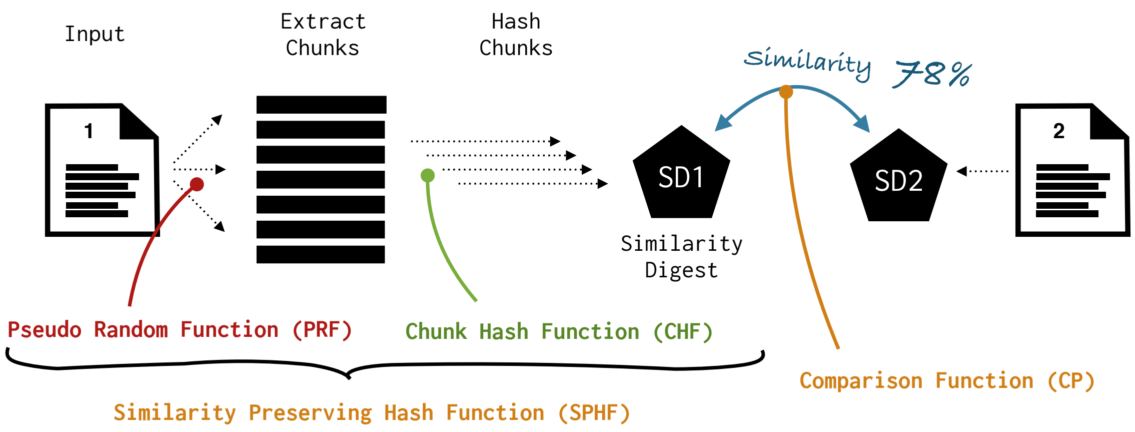

Input
Extract Chunks
Hash Chunks
Similarity 78%
1
SD1
SD2
2
Similarity Digest
Pseudo Random Function (PRF)
Chunk Hash Function (CHF)
Comparison Function (CP)
Similarity Preserving Hash Function (SPHF)

Figure 1.1 The basic functionality and its components of Context-Triggered Piecewise-Hashing.

In the last year we implemented several components to interface approximate matching with the task of memory analysis. Beside that, several approaches have pointed into the same direction. However, to the best of our knowledge, this the first implementation which summarizes such capabilities and discusses the application of fuzzy hashing for the task of carving memory fragments. We consider the approach for the task of carving code artefacts in raw or physical memory dumps. With the presented prototype implementation, we see further paths of data driven cross validation, blacklisting, whitelisting, integrity checks, user space verification, code injection detection or hollowing detection. Summarized, we see the potential that our implementation could enhance OS-reliant analysis.

Capabilties: Our current implementation is able to quickly process large amounts of binaries and match/identify similar/identical sequences in raw images. We interfaced approximate matching (a.k.a. fuzzy hashing or similarity hashing) with an additional layer of approximate disassembling. Our implementation is able to discriminate code from data and reduces variances in the code structures (caused during loading/linking).

2 Related Publications or Plugins

In the following we mention some publications and projects which influenced our own prototype of `approxis`.

Walters and White. Several approaches have been discussed for integrity verifcation of user space related memory regions [5, 6, 4, 3]. The original approach of Walters et al. [3] was constantly improved. The pitfalls of user space code verification have been outlined. Both approaches use additional steps of code normalization patterns to damp variances caused by the loader. White et al. [5] proposed the utilization of a virtual binary (PE) loader to generate a suitable database for code normalization. The simulation of the loading process empowered to formalize offsets which have to be normalized. A process identification helps to depict the suitable hash templates out of a large database of generated and normalized hash-patterns. The hashing and comparison is performed on the boundaries of a memory page. Both approaches require a golden image baseline.

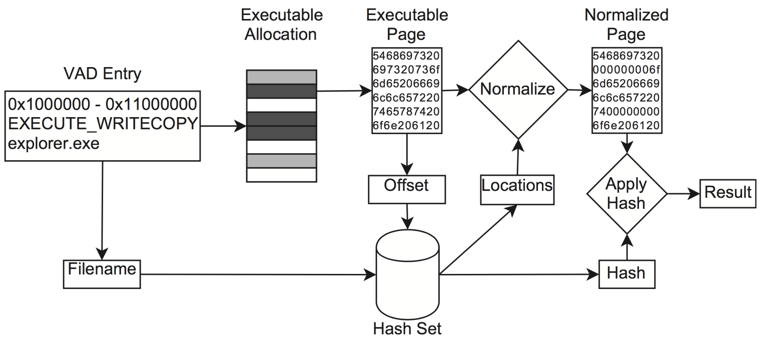


Figure 2.1 Matching of hash-templates with extended process identification. Source: White et al..

Online Hash Database. Comparable to Walters and White, K2 realized an implementation of memory/process verification[1]. Similar to White et al. [5] a process identification and PDB verification leads to a lookup of memory fragments on a page boundary. The utilization of PDB informations makes the technique independent of a golden image baseline. The implementation empowers to perform a binary diffing and to inspect variations for an acquired process context. The lookup matches the hash values against an appropriate or supposed candidate and highlights discrepancies.

The already mentioned approaches by White/Walters and K2 are strongly focusing on in memory code/process verification. This requires the correct identification of a target process itself. In contrary, the page-boundary could suffer from simple offsets.

YARA Carving. Cohen [1] proposed in his paper *Sanning Memory with YARA* the application of YARA rules for the task of *unstructured*-memory analysis. Cohen mentions the application in the case of malware signature scanning. Of course, the application in the case of memory analysis yields new pitfalls: YARA-Rules have to be adapted to be applicable in the context of physical memory images. With the utilization of the Windows PFN database, the overall approach was process-context aware by processing a physical image.

Summary. Most of the mentioned techniques share similar needs and demands. With the introduction of our approach, we see several additions to existing tools and research. Most of the memory verification approaches are reliant on the extraction of adequate and usable process information. Thus, the verification itself depends on the correct process identification. Similar statements could be applied on related tools which are applied on specific artefacts (e.g., import address tables[2]). In addition, the mentioned approaches mostly rely on page boundaries and thus, suffer from fixed block/chunk sizes during examination. In contrary to this *top down* approaches (extraction of structural properties and verification afterwards), we try to enrich current structured analysis with the help of *bottom up* examinations (carving and matching artifacts).

3 Approxis

The following section will give an overview of the application and implementation of approxis.

[1]`https://github.com/K2/Scripting`
[2]`https://github.com/JPCERTCC/impfuzzy/tree/master/impfuzzy_for_Volatility`

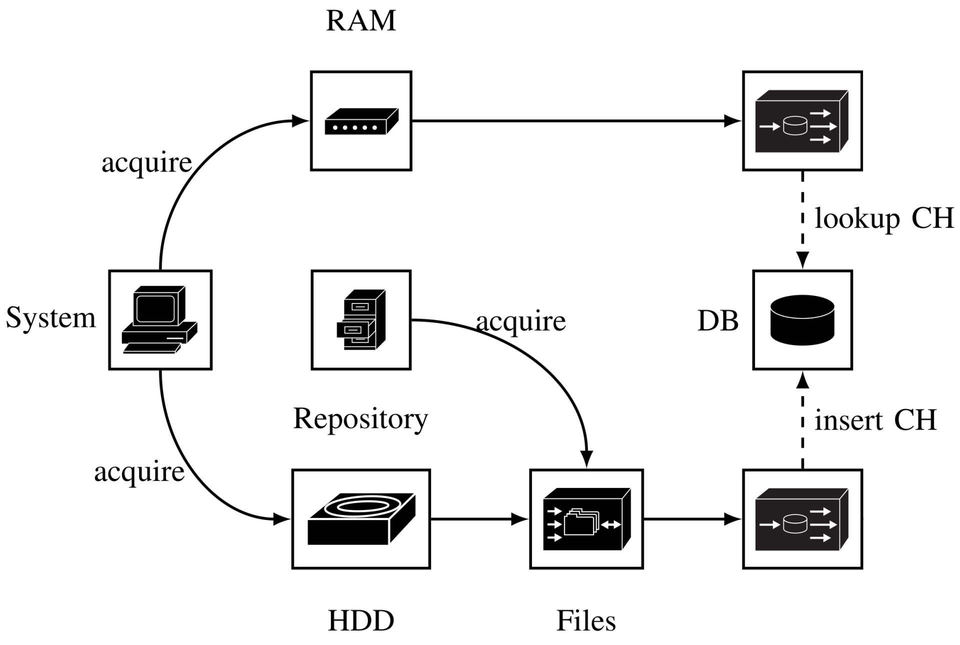


Figure 2.2 Simplified use case scenario: comparing/carving memory with offline or online acquired data.

3.1 Use Case

In the course of this submission we describe a very basic application of our current prototype implementation. Therefore, we first acquire a large set of offline data. For example from an appropriate hard disk or online repository. The data is processed with the multi-layered processing of approxis and saved into a large hash database. As could be seen in Figure 2.2, the acquisition paths could be differed into source (hard disk, repository) and target medium (RAM). Fragments are first inserted into a database and the query could be performed against it.

3.2 Procssing steps and Features

We will give some details and insights of the current implementation and its features. However, we will skip most of the advanced explanations and refer to our contact or the mentioned publications in Section 5.

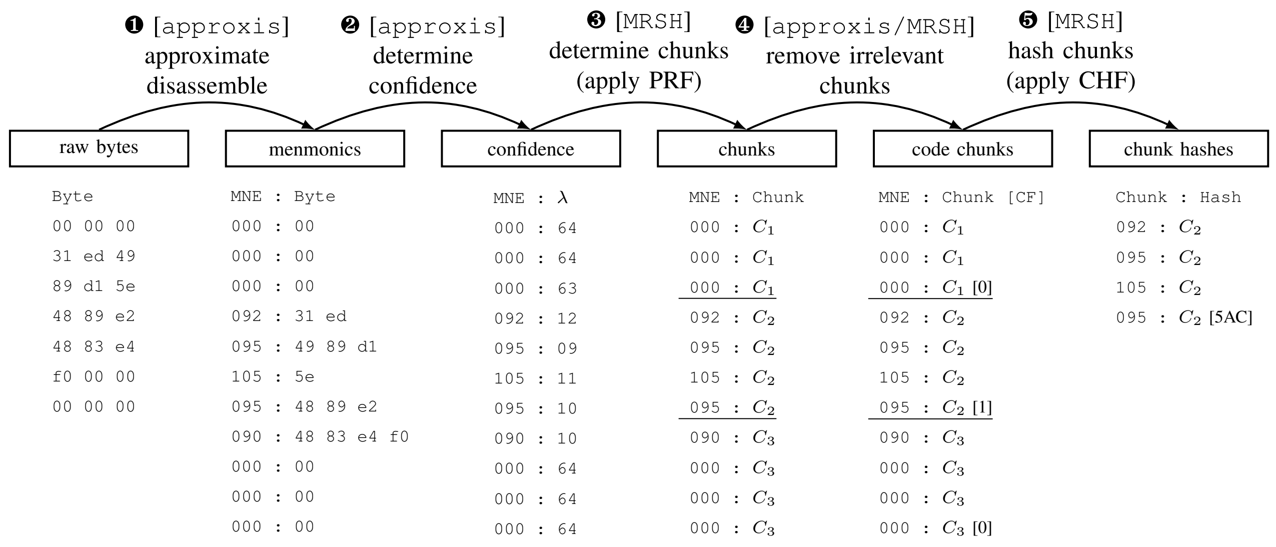
3.2.1 Pipeline

The single processing steps of `approxis` could be overviewed in Figure 3.1.

❶ The raw bytes from the memory are disassembled using `approxis` 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 By processing a large set of ground truth binaries (ELFs) we first learned ”common” sequences of instructions. Using this 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.

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). A pseudo-random function and a fixed modulus defines a current chunk boundaries (Context-triggered Piecewise-Extraction).


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)
raw bytes
menmonics
confidence
chunks
code chunks
chunk hashes

raw bytes	menmonics	confidence	chunks	code chunks	chunk hashes
Byte	MNE : Byte	MNE : λ	MNE : Chunk	MNE : Chunk [CF]	Chunk : Hash
00 00 00	000 : 00	000 : 64	000 : C_1	000 : C_1	092 : C_2
31 ed 49	000 : 00	000 : 64	000 : C_1	000 : C_1	095 : C_2
89 d1 5e	000 : 00	000 : 63	000 : C_1	000 : C_1 [0]	105 : C_2
48 89 e2	092 : 31 ed	092 : 12	092 : C_2	092 : C_2	095 : C_2 [5AC]
48 83 e4	095 : 49 89 d1	095 : 09	095 : C_2	095 : C_2	
f0 00 00	105 : 5e	105 : 11	105 : C_2	105 : C_2	
00 00 00	095 : 48 89 e2	095 : 10	095 : C_2	095 : C_2 [1]	
	090 : 48 83 e4 f0	090 : 10	090 : C_3	090 : C_3	
	000 : 00	000 : 64	000 : C_3	000 : C_3	
	000 : 00	000 : 64	000 : C_3	000 : C_3	
	000 : 00	000 : 64	000 : C_3	000 : C_3 [0]	

Figure 3.1 The simplified processing pipeline.

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 bad confidence score (64, 64, 63; higher is worse), therefore we consider this chunk as not relevant (indicated by [0]). We will mention this later during the possible parametrization.

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.

3.2.2 Features and Parameters

In the following we will mention some specific parameters which are actually implemented in the prototype and partially available.
Chunk Size. With the parametrization of the block size, someone could influence the extraction process of chunks. In detail, if we focus on smaller interleaved chunks only or on large matching sequences nearly to a page-size. We are not fixed to an overall block size.
Code/Data Extraction. As already mentioned, with the help of the approximate disassembling we are able to discriminate code from data. However, this is still a field of ongoing research. The carving capabilities strongly depends on the capabilities of the current disassembler engine. We see a lot of potential for further improvements and hopefully could contribute in further releases. We will outline the parameters later. For a visual demonstration please see Figure 3.2.
Minimum Run Size. The *run size* normally depicts the minimum amount of subsequent chunks before a hit/match will be count.

3.3 Prototype Installation and Usage

The current implementation was mainly realized in C. After the replacement of simple Bloom filter lookups with more advanced database features, we had to lift the code to a mixed up C/C++ prototype version. We also integrated Python Extensions to interface our current implementation with Volatility.

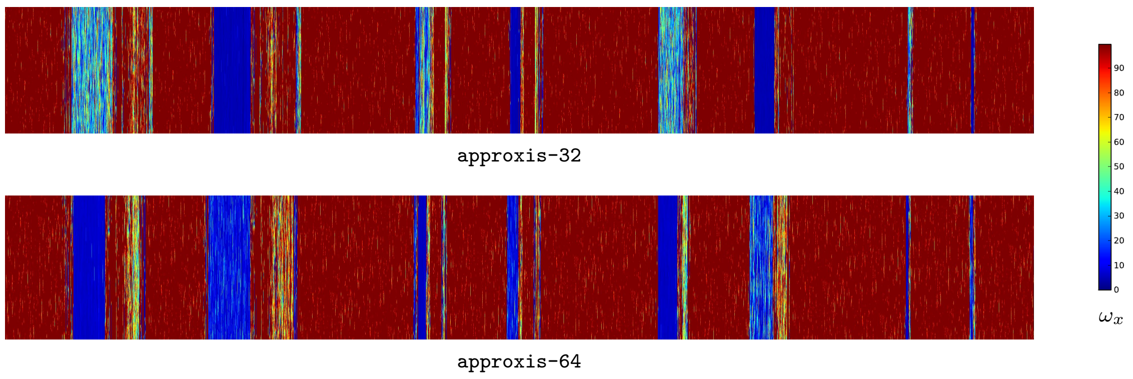


Figure 3.2 Approximate Disassembling features the detection of interleaved code sequences. The both bars show random patterns with interleaved x86 and x64 binaries. Note: the plot emphasizes that the engines is able to discriminate the architecture. (ω_x is the windowed confidence value)

3.3.1 Installation

You could install `approxis` as standalone tool or as a python extension. We currently support Python2 and Python3. However, the interface is pretty undeveloped. For the installation cmake is required. Please note: the installation process could take several minutes.

Standalone: The current prototype `approxis` has to be build with optimization set to -O3. Please do not change the current cmake configuration.

```
# Native Installation
tar -xzf approxis.tar.gz
cd approxs
cmake CMakeLists.txt
make
```

Python-Extension: For the installation as Python extension we recommend the creation of an virtual environment. The current version should support Python 2.7 and Python 3.*. For the installation type the following commands:

```
# Python-Extension Installation (You could Replace 2.7 with 3)
tar -xzf approxis.tar.gz
cd approxis
virtualenv -p /usr/bin/python2.7 env27
source env27/bin/activate
pip install .
```

3.3.2 Usage

It should be mentioned that the current interface is not yet well defined. In the following we describe the basic usage without mentioning obvious and possible improvements.

Standalone: The current command structure first expects the creation of a lookup database. You could specific single files, larger images or folder structures. Folder structures are parsed recursively without following/processing symlinks. The offset could be depicted to select regions of input data. Parameters like the virtual offsets could be helpful for dissecting binaries in combination with advanced disassemblers. The disassembling output is currently deactivated.

```
# Database Creation
#./approxis [architecture] [filename] [file_offset] [virtual_offset] [section_size] [db]
./approxis 64 /path/to/libraries 0 0 0 database_name
```

Afterwards, someone could perform the lookup against the database. It should be mentioned that during the comparison-phase, the actual `db` specification has no influence and no database will be created.

```
# Database Lookup
#./approxis [architecture] [filename] [file_offset] [virtual_offset] [section_size] [db] <db_comp>

./approxis 64 /path/to/image.lime 0 0 0 no_matter database_name
```

Python-Extension: You normally should simply import the module and use its interface methods and global parameters in a similar fashion as the standalone version. In the course, of this submission we have no actual return value. The matching sequences are stored into a separate file (`log.txt`).

```
# Database Creation as Extension
import approxis

ret = approxis.run(64, "/path/to/libraries", 0, 0, 0, "database_name")
ret = approxis.run(64, "/path/to/image", 0, 0, 0, "temp", "database_name")
```

3.3.3 Execution Output

We minimized the output to the essential matching of unique hits (chunks which could be matched to a single input file). We additionally output the multihits and save the matching sequences into a temporal database. The database format is a simple format in the shape of
`Chunk start - Chunk end | File/Multihit`.
The current prototype writes the output to a file `log.txt`, which could be parsed by a Volatility plugin afterwards. Of course, this processing is far away from perfect (fastest way to realize a demonstrator ☺).

```
import approxis
approxis.run(64, "/path/to/image", 0, 0, 0, "ignored", "database_name")
Start-End|Filename
****************************************************************
 --> Add file: image
...
2146694447-2146694679|libgcrypt.so.20.0.3
2146695686-2146696081|libgcrypt.so.20.0.3
2146696081-2146696475|libgcrypt.so.20.0.3
...
2146700479-2146700840|libgcrypt.so.20.0.3
2146705391-2146705985|libgcrypt.so.20.0.3
2146708766-2146709186|libwireshark.so.5.0.1
2146709186-2146709541|libwireshark.so.5.0.1
2146709541-2146710793|libwireshark.so.5.0.1
2146710793-2146711174|libwireshark.so.5.0.1
2146711174-2146711610|libwireshark.so.5.0.1
...
```

Long-term goal is a good runtime performance. Therefore, the approximate disassembling engine performs the decoding on a byte-level. The current implementation processes a 2 GiB image in approx 1 minute. The execution was performed on an descent business laptop (Machine with Intel Core i5 2x2.2 GHz Processor and 8 GiB RAM).

3.4 Parametrization

There are different parameters available. The current parametrization allows the influence of the chunk extraction process: with three different parameters the user has the possibility to control the focus of extraction. In detail, the user could extract only chunks which contain code with a high probability (min=70,max=100), chunks which contain data with a high probability (min=0,max=30), or chunks which contain contain both (min=0,max=100; i.e., all chunks).

```
# Parametrization of global approxis parameters
import approxis

```

```
"""
 CODE_THRESH
 Threshold which defines if two decoded instructions are
 meaningful instruction pairs or not. A low value would thus
 only consider very common sequences. A high value would
 consider even sequences of data as code offsets.
"""
approxis.CODE_THRESH = ctre

"""
 CODE_COV_MIN / CODE_COV_MAX
 Code offsets defined by the threshold of CODE_THRESH are
 counted in each chunk. The value of CODE_COV_MIN defines
 the minimum threshold of present code offsets within a chunk.
 Thus, replacing cmin with 0 would consider every chunk. A
 value of cmin=50 would only process chunks where at least
 50\% of the offsets are considered as code.
 The global parameters of CODE_COV_MAX defines the upper
 bound of possible code offsets. For example, depicting a
 value of cmax=0 would lead to the extraction of chunks
 which do not contain any considerable code offset.
"""

approxis.CODE_COV_MIN = cmin
approxis.CODE_COV_MAX = cmax
```

4 Volatility Plugin - `apx_maps`

We created a simple plugin called `linux_apx_maps`, which is mainly based on the original `linux_proc_maps`. The plugin utilizes the carving output of our approach and empowers to match sequences against specific mappings. The current implementation is far from perfect and we see further improvements in case of integration and capabilities. Please see Listing 4.1 for further details.

```

> python vol.py --file=/path/to/image/image.lime --profile=LinuxDebianx64 linux_apx_maps -p 3095 -L /path/to/log.txt

 [...]Start       End         File Path   [...]                                                PAllocs     POffsets
[...]----------- ----------- ------------[...]-------------------------------------------------- ----------- --------
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]xbuf-2.0/2.10.0/loaders/libpixbufloader-png.so    5/5         [('libpixbufloader-png.so', 9419)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]xbuf-2.0/2.10.0/loaders/libpixbufloader-png.so    1/1         []
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]xbuf-2.0/2.10.0/loaders/libpixbufloader-png.so    1/1         []
 [...]0x7f...000  0x7f...000             [...]                                                  2/2048      []
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]xbuf-2.0/2.10.0/loaders/libpixbufloader-xpm.so    6/6         [('libcairo.so.2.11400.0', 14295), ('dumpcap', 621)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]xbuf-2.0/2.10.0/loaders/libpixbufloader-xpm.so    1/1         []
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]xbuf-2.0/2.10.0/loaders/libpixbufloader-xpm.so    1/1         []
 [...]0x7f...000  0x7f...000             [...]                                                  1/2558      []
 [...]0x7f...000  0x7f...000             [...]                                                  1/2558      []
 [...]0x7f...000  0x7f...000             [...]                                                  1280/1283   [('libwireshark.so.5.0.1', 27182), ... ('libgtk-x11-2.0.so.0.2400.25', 49)]
 [...]0x7f...000  0x7f...000  /usr/share/[...]/DejaVuSansMono.ttf                               36/82       [('libicudata.so.52.1', 41)]
 [...]0x7f...000  0x7f...000             [...]                                                  267/292     [('libwireshark.so.5.0.1', 411513), ('wireshark', 4376)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/m2m.so                         4/4         [('m2m.so', 20154)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/m2m.so                         1/1         []
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/m2m.so                         1/1         []
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/gryphon.so                     17/17       [('gryphon.so', 73604), ('profinet.so', 1709)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/gryphon.so                     2/2         []
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/gryphon.so                     4/4         [('m2m.so', 9816)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/profinet.so                    83/90       [('profinet.so', 594154), ('libwireshark.so.5.0.1', 9921), ... ('wireshark', 4234)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/profinet.so                    8/8         [('profinet.so', 45907)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/profinet.so                    18/18       [('gryphon.so', 196)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/mate.so                        22/22       [('mate.so', 141783), ('wimaxasncp.so', 249), ('libwireshark.so.5.0.1', 165)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/mate.so                        1/1         []
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/mate.so                        1/1         []
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/wimaxasncp.so                  15/15       [('wimaxasncp.so', 23551), ('libcontroller-linux-input.so', 43)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/wimaxasncp.so                  2/2         []
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/wimaxasncp.so                  1/1         []
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/stats_tree.so                  2/2         [('stats_tree.so', 3301)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/stats_tree.so                  1/1         []
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/stats_tree.so                  1/1         []
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/opcua.so                       41/44       [('opcua.so', 195417), ('docsis.so', 9718), ('libwireshark.so.5.0.1', 56)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/opcua.so                       4/4         [('opcua.so', 1508)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/opcua.so                       11/11       [('libwireshark.so.5.0.1', 3244)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/docsis.so                      46/47       [('docsis.so', 569927), ('libQtWebKit.so.4.10.4', 8236)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/docsis.so                      2/2         [('docsis.so', 4176)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/docsis.so                      14/14       [('docsis.so', 3069)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/wimaxmacphy.so                 15/15       [('wimaxmacphy.so', 39709), ('ethercat.so', 26846), ('docsis.so', 116)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/wimaxmacphy.so                 2/2         []
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/wimaxmacphy.so                 5/5         []
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/ethercat.so                    24/24       [('ethercat.so', 266597)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/ethercat.so                    2/2         []
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/ethercat.so                    9/9         [('libwireshark.so.5.0.1', 88)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/irda.so                        10/10       [('irda.so', 102909)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/irda.so                        1/1         []
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/irda.so                        2/2         [('irda.so', 18210)]
 [...]0x7f...000  0x7f...000             [...]                                                  1/1         []
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/wimax.so                       72/120      [('wimax.so', 598003), ('unistim.so', 4017)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/wimax.so                       6/6         [('libwireshark.so.5.0.1', 6203), ('wimax.so', 2980)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/wimax.so                       34/34       [('libgdk-3.so.0.1400.5', 2320)]
 [...]0x7f...000  0x7f...000  /usr/lib/x8[...]ark/plugins/1.12.1/unistim.so                     24/26       [('unistim.so', 82657)]

...
```

Figure 4.1 Application of `apx_map` on a running process. Last column identifies matches in specific area (`POffsets`) and the amount of present allocations usable for matching (`PAllocs`)
.

5 Conclusion and Literature

We see a lot of potential and a lot of challenges to be solved:

- The current implementation and code quality should be improved and refactored. Even if we focussed on runtime performance during the development of our prototype, we see a lot of potential for further performance optimizations. The current code is really sloppy and needs several rounds of rewriting.
- Parameter settings have to be adopted for different tasks. In the case of library identification someone would prefer the extraction of code-related chunks. In the case of matching two binaries (e.g., with different optimization levels) someone could prefer the extraction of constant data fragments (e.g., strings).
- The parametrization must be described in an easy and understandable way. In addition, several parameters still need to be defined before application (e.g., target architecture x86/x64).

Summarized, we hope to push the current academic approach to a more usable and discussable application. Therefore, we rely on the feedback of real-world participators and their parametrization. This is the reason why we push forward and decided to publish the current prototype implementation in the course of this contest.

Further information could be found in the following publications.

```
Lorenz Liebler and Frank Breitinger, "mrsh-mem: Approximate Matching on Raw Memory Dumps",
in Proceedings of 11th International Conference on IT Security Incident Management & IT
Forensics (IMF'18), Hamburg (Germany), May 2018.

Lorenz Liebler and Harald Baier, "Approxis: a fast, robust, lightweight and approximate
Disassembler considered in the field of memory forensics", in Proceedings of the 9th EAI
International Conference on Digital Forensics & Cyber Crime (ICDF2C), Prague (Czech
Republic). October 2017.
```

Bibliography

[1] Michael Cohen. Scanning memory with yara. *Digital Investigation*, 20:34–43, 2017.

[2] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory.* John Wiley & Sons, 2014.

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

[4] Andrew White. *Identifying the unknown in user space memory.* PhD thesis, Queensland University of Technology, 2013.

[5] Andrew White, Bradley Schatz, and Ernest Foo. Integrity verification of user space code. *Digit. Investig.*, 10:S59–S68, August 2013.

[6] Andrew White, Bradley Schatz, and Ernest Foo. Integrity verification of user space code. Presented at The Digital Forensic Research Conference, Monterey, CA, 2013. URL `https://www.dfrws.org/sites/default/files/session-files/pres-integrity_verification_of_user_space_code.pdf`.