Approxis: A Fast, Robust, Lightweight and Approximate Disassembler Considered in the Field of Memory Forensics

Lorenz Liebler(✉) and Harald Baier

da/sec - Biometrics and Internet Security Research Group,
University of Applied Sciences, Darmstadt, Germany
{lorenz.liebler,harald.baier}@h-da.de

Abstract. The discipline of detecting known and unknown code structures in large sets of data is a challenging task. An example could be the examination of memory dumps of an infected system. Memory forensic frameworks rely on system relevant information and the examination of structures which are located within a dump itself. With the constant increasing size of used memory, the creation of additional methods of data reduction (similar to those in disk forensics) are eligible. In the field of disk forensics, approximate matching algorithms are well known. However, in the field of memory forensics, the application of those algorithms is impractical. In this paper we introduce `approxis`: an approximate disassembler. In contrary to other disassemblers our approach does not rely on an internal disassembler engine, as the system is based on a compressed set of ground truth x86 and x86-64 assemblies. Our first prototype shows a good computational performance and is able to detect code in large sets of raw data. Additionally, our current implementation is able to differentiate between architectures while disassembling. Summarized, `approxis` is the first attempt to interface approximate matching with the field of memory forensics.

Keywords: Approximate disassembly · Approximate matching
Disassembly · Binary analysis · Memory forensics

1 Introduction

Detecting known malicious code in memory is a challenging task. This is mainly due to two reasons: first, malware authors tend to obfuscate their code by tampering it for each instance. Second, code in memory differs from persistent code because of changes performed by the memory loader (e.g., the security feature *Address Space Layout Randomization (ASLR)* makes it impossible to predict the final state of an executable right before run time). Hence an approach to identify malicious code within a memory forensics investigation by comparing code fragments in its untampered shape (e.g., as an image on disk) to its memory loaded representation (e.g., a module with variable code) is a non-trivial task.

P. Matoušek and M. Schmiedecker (Eds.): ICDF2C 2017, LNICST 216, pp. 158–172, 2018.
https://doi.org/10.1007/978-3-319-73697-6_12

Memory forensic tools like volatility use system related structures to extract loaded executables and to list executed processes on a system. The classical approach to identify loaded malware is performed with the help of signatures, static byte sequences or by the examination of access protections. White et al. [10] formulate requirements of investigating a memory image and postulate that methods of data reduction (similar to those in disk forensics) are eligible. In the field of disk forensics approximate matching algorithms (a.k.a. similarity hashing or fuzzy hashing) represent a robust and fast instrument to differentiate between known and unknown data fragments [3,6]. However, White et al. [10] claim that approximate matching algorithms are not suitable in the course of memory forensics, as code in memory always differs to on disk.

In this paper we argue that the concept of approximate matching may be transferred from post-mortem or network forensics to the field of memory forensics. We differentiate between two stages of research to succeed. First, a technical component is needed, which acquires portions of code in different domains and extracts these fragments out of vast amounts of unknown data. Second, the acquired code fragments must be comparable. As the existing approaches in the field of memory forensics try to solve both issues at once by creating a stack of dependencies and accepting limitations of applicability [8,10], our approach focuses on the technical component first and introduces an interface to transfer the overall problem of code detection into the domain of approximate matching.

Our main contribution of this paper is the technical acquisition component `approxis`: a lightweight, robust, fast and approximate disassembler as a prerequisite for memory-based approximate matching. The goal of `approxis` is to build a technical component for the usage in digital forensics, however, `approxis` may be used in different fields like real-time systems, too. Its functionality is comparable to a basic length-disassembler approach with additional features.

Our approach is unaware of the full instruction encoding scheme of x86 or x86-64 platforms: by the usage of 4.2 GiB precompiled ELF (Executable and Linking Format) files and its corresponding ground truth assembly structure obtained by [1], we build up a decision tree of byte instructions. Each path of the tree represents the decoding process of a byte sequence to its corresponding instruction length. We use the opcode and mnemonic frequencies to assist the disassembling process and to differentiate between code and non-code byte sequences. The overall goal of `approxis` is not to reach the accuracy of professional disassemblers, but to outreach the capabilities of a simple length disassembler.

We evaluate our approach in different fields of application. First, we show the promising disassembling accuracy of `approxis` compared to `objdump`, a widely distributed and often used linear disassembler. Second, our approach is able to distinguish between code and data. Third, we demonstrate the capabilities to identify interleaved segments of code within large sets of raw binary data. Our current implementation introduces the possibility to determine the architecture of code during the process of disassembling. Finally, we demonstrate the computational performance of `approxis` by the application on a raw memory image.

It is important to outline the conditions and the operational field of `approxis`, as our approach should not be considered in the well known domains of binary analysis. Thus, even if the final evaluation of `approxis` could seem to be incomplete to the reader, we argue that the extensive introduction of our approach in the field of memory forensics is important to understand the following design decisions. Additionally, it is somewhat negligible and deceptive to compare our approach to other disassemblers. However, our current implementation of `approxis` is designed for processing large portions of raw memory dumps, so a straight comparison with other disassemblers is not always valid.

The remainder of this paper is organized as follows: In Sect. 2 we give an overview of related work. We introduce key features of existing research and describe instances of different disassemblers. In Sect. 3 we define central requirements which should be fulfilled by `approxis`. In Sect. 4 we briefly introduce the x86 decoding scheme and the challenges of disassembling. We also introduce the results of analyzing our ground truth assemblies obtained by [1], which build the foundation for our code detection and approximate disassembling approach. In Sect. 5 we introduce `approxis` and its functionality. In Sect. 6 we present our assessment and experimental results. Finally, Sect. 7 concludes this paper.

2 Related Work

Researches discussed different approaches for the application of cryptographic hash functions on memory fragments. Existing work addresses the problem of identifying known code by hashing normalized portions of code in memory. A short survey of existing approaches was given by [10]. In [8] offsets of variable code fragments were used to normalize and hash executables on a page level. A database of hash templates was created which consists of hash values and its corresponding offsets. These hash templates are applied on the physical address space. The comparison between each template and each page lead to a complexity of $O(n * m)$ for a comparison of n templates against m memory pages.

The authors of [10] extended the approach and tried to improve the naive all-against-all comparison introduced by [8]. Therefore, they applied the hashes on virtual memory pages and used structures in memory to identify a process. By identifying a process, the lookup of a corresponding hash template could be performed efficiently. Before creating the hash values, the introduced approaches convert a present executable from disk to its state in memory and normalize it. The conversion of disk stored image files to a virtual loaded module was accomplished with the help of a virtual Windows PE Loader [10]. The identification of variable offsets by imitating the loading process of an executable seems legit. A normalization based on previously disassembling a present sequence of bytes in memory was not mentioned by the authors.

Recent research of linear disassemblers has shown the significant underestimation of linear disassembly and the dualism in the stance on disassembly in literature [1]. A more exotic form are the so called length-disassemblers,

which could be understood as a limited subset of linear disassemblers. A length-disassembler only extracts the lengths of an instruction. Beside the classical linear and recursive disassemblers, the authors of [7] introduced an experimental approach of fast and approximate disassembly. The approach is based on the statistically examination of the most frequent occurred mnemonics. A set of extracted sequences of mnemonics have been used to create a lookup table of predominant bigrams. With the help of this table, a fuzzy 32 bit decoding scheme was proposed, which showed decent results.

As already introduced, approximate matching algorithms can be used to detect similarities among objects, but also to detect embedded objects or fragment of objects [3,6]. Investigators can use it to differ between non-relevant and relevant fragments in large sets of suspicious data. In the course of memory forensics this approach would obviously struggle with volatile instruction operands and updated byte-sequences. Current approximate matching techniques constantly evolve, e.g. by the integration of better lookup strategies like Cuckoo Filters [5].

The problem of identifying code structures in large sets of binary data could be misleadingly compared with the problem of identifying interleaved data within code sections of a single executable [9]. The major goals of our approach are the fast identification and the approximate disassembly of code fragments.

3 Requirements of Approximate Disassembling

In this section we introduce and explain four essential requirements for our research: *lightweight*, *robustness*, *speed* and *versatility*. These requirements should be understood as superior and long term goals in the context of applying approximate matching to the field of memory forensics. They have to be respected in this research and beyond this work. To be able to better describe the fundamental requirements, we first introduce the central goals of this publication. As the application of approximate matching algorithms to portions of memory seems unfeasible due to a unpredictable representation of code in memory, we suggest a process of normalization after approximate disassembling portions of code in large sets of raw and mixed data. As this work addresses the step of identifying and disassembling code in data, we define four major goals:

1. Detect sequences of code in a vast amount of different shaped raw data.
2. Extract sequences of instruction-related bytes with little overhead.
3. Make a statement about the confidence of the code detection process.
4. Determine additional information, like the architecture of the code.

These practical goals describe the motivation of this work, where the following requirements describe the bounding conditions to achieve those goals. The defined requirements are discussed by recalling some central properties of the introduced competing approaches and by considering the mentioned goals.

The first requirement *lightweight* aims to reduce the stack of dependencies of the target system with a focus on the instruction set and the loader traces. In

contrast to existing approaches, we propose a normalization based on previously disassembling code in different states of an executable. We consider this approach significant more lightweight than imitating loader traces with the help of a self-constructed virtual loader. A disassembler is therefore less interleaved to record the changes of a memory loader to an image file.

Previous work to detect known fragments of code (e.g., the approach introduced by [10]) relies on the correct identification of a running process. This offers new degrees of bypassing and obfuscation to the malware author, e.g., by unlinking Virtual Address Descriptor (VAD) nodes using Direct Kernel Object Manipulation [4]. Our second requirement *robustness* means to identify a code fragment without process structures and being thus more robust against obfuscation compared to competing approaches.

Our third requirement is *speed*, which is a central requirement adopted from the field of approximate matching. In our current stage of research the detection and extraction of code from a vast amount of data has to be done with good computational performance. As we are interested in an approximate disassembler, we trade computational performance more important than accuracy of the disassembled code. However, the degree of disassembling should enable further normalization or the reduction of code representation.

Most of the introduced systems in Sect. 2 are limited to x86 systems. A more *versatile* approach is desirable, which is not dependent on an a-priori knowledge of the architecture of the target system. The requirement *versatility* means that the disassembler works reliably for different target architectures.

4 Background and Fundamentals

In this section we introduce the basic fundamentals of our approach for the introduction of `approxis`. We briefly introduce the target x86 system. Afterwards, we introduce the set of ground truth assembly files in a detailed way.

4.1 Disassembling

We first give a short introduction to the x86 encoding scheme and the fundamentals of disassembling. Disassemblers are used to transform machine code into a human readable representation. In the field of binary analysis and reverse engineering the demands and requirements of a disassembler engine are clearly identified. With the x86 instruction set these tools have to deal with variable-length and unaligned instruction encodings. Additional, executables sections could be interleaved by code and data sequences. As the authors of [9] already described, this system design trades simplicity for brevity and speed. Summarized, the process of disassembly in general is undecidable [1,9]. As could be seen in Fig. 1 the x86 instructions are defined by sequences of mandatory and non-mandatory bytes. The `Reg` field of the `ModR/M` byte is sometimes used as an additional opcode extension field. Prefix bytes could additionally change the overall instruction length. For further details we refer to the Intel Instruction manual[1].

[1] https://software.intel.com/en-us/articles/intel-sdm.

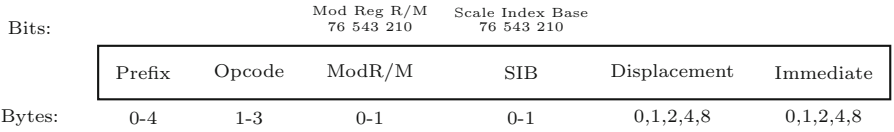

Fig. 1. x86 machine instruction format

The core of this research is to approximate disassemble a vast amount of unknown data. This desire clearly stays in conflict with the goal of classical disassembler engines, where computational performance is often understood as a secondary goal. We ignore recursive traversal, as this would implicate an impractical layer of computational overhead. The development and the maintenance process of disassemblers is somewhat cumbersome and tedious. Even the lookup tables of a simple length-disassembler have to be maintained.

4.2 Mnemonic Frequency Analysis

We analyzed the opcode and mnemonic distribution of a set of ELF binaries, namely a dataset containing 521 different binaries obtained by [1]. As we focus on the acquisition of byte sequences which rely to code only, we extracted the `.text` section of each binary file. It should be mentioned that the following distribution analysis is nothing new [2,7]. However, existing distribution analysis of mnemonics often rely on malware, which could be biased. We used the ground truth of assemblies to determine the distribution of mnemonics and extracted the bigrams of mnemonics (see Table 1). We splitted the set of assemblies by its architecture and determined the *total* amount of unigrams and bigrams. The column of *distinct* values describes the set of all occurring mnemonics. The columns *max*, *mean* and *min* describe the assignment of the *total* amount of instructions to each *distinct* unigram or bigram. For example, the most frequently occurred mnemonic in the case of 32 bit binaries represents 33.25% of all instructions.

Table 1. Overview of unigram and bigram mnemonic counts.

	32 bit (200 files)				64 bit (321 files)			
	Total	Distinct	Max	Mean	Total	Distinct	Max	Mean
Unigrams	35.232k	322	11.714k	1531	61.441k	436	21.627k	1859
Bigrams	35.232k	11632	5.889k	17	61.441k	16059	10.360k	28

The frequency of occurrence of all bigrams are extracted, the probability p of each bigram is saved as logarithmic odds (logit). We further denote the absolute values of logits as λ (see Eq. 1). Similar to [7] we want to avoid computational underflow by multiplication of probabilities.

$$\lambda = \left| \ln \frac{p}{1-p} \right| \tag{1}$$

4.3 Byte Tree Analysis

The former subsection revisits the frequencies of most frequently occurred mnemonics. In a next step we analyze the byte frequencies on a instruction base. We have to deal with a vast amount of overlapping byte sequences and non-relevant operand information. To refine our demands, the overall goal of `approxis` is not do establish a high-accuracy disassembler, but to identify instruction offsets and a predominant mnemonic. We extract all bytes of an instruction and insert them in a database structured as tree. Each node of the tree represents a byte and stores a reference to all its corresponding children, the subsequent instruction bytes (see Fig. 2).

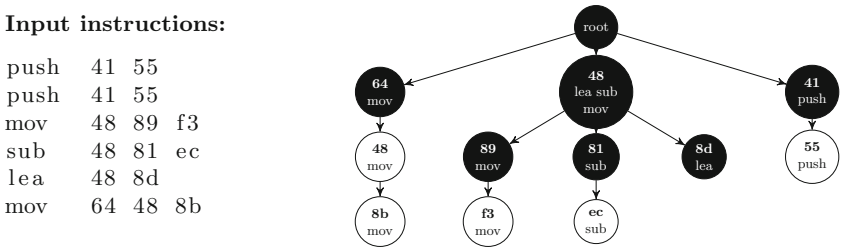

Fig. 2. Oversimplified bytetree example after inserting several instructions.

As an example we inspect the byte sequence `488d` and its subsequent bytes after inserting our ground truth into the tree. In listing 1 we can see the complete output of a single node. We should mention that the amount of the child nodes was shortened for a better representation ❷. We also save auxiliary information like the amount of counted bytes for a current node ❶, the counts of all corresponding mnemonics ❸❺ and the counts of different occurring instruction lengths ❸❹. Each node maintains different formats and could possibly lead to redundant information. This structure represents an intermediate state needed for the following steps of data analysis, post processing and tree reduction.

Listing 1: Inspecting a node of `lea` (`48 8d`) instruction

```
1 Current node: ['48,8d']; Count: 1334022 ❶
2 Child nodes:  [83,aa,04,87,2d,8b,0c,8f,93, ... ,69,7d,6d,71,75,48] ❷
3 { 3:669k, 2:11k, 4:273k, 7:207k, 6:172k}
4 { 2:{lea:11k}, 3:{lea:669k}, 4:{lea:273k}, 6:{lea:172k}, 7:{lea:207k}} ❸
5 [[3, 669k], [4, 273k], [7, 207k], [6, 172k], [2, 11k]] ❹
6 (lea', 1334k) ❺
```

After inserting the whole ground truth into the tree we perform an additional step of reduction. Every node which represents a single length and a single mnemonic was transformed to a leaf node. So we cropped all subsequent child nodes of the current node, which doesn't affect the instruction mnemonic. The reduced shape of the tree is highlighted black in Fig. 2. The impact of reduction could be seen in Table 2.

Table 2. Comparison of original and reduced bytetree.

Platform	Input bytes	Original tree			Reduced tree		
		Nodes	Height	Size	Nodes	Height	Size
64 bit	253.535.572	12.773.078	15	445M	87.224	10	7.5M
32 bit	123.221.439	5.871.232	15	206M	35.211	9	3.0M

5 Approach

The observation of the preceding section lead the deduction of our approach, which is based on the introduced bytetree and mnemonic frequency analysis.

5.1 Disassembling

We argue that length-disassemblers could be assumed to be very fast and lightweight. Though, even a simple length-disassembler needs to respect a lot of basic operations and needs to be maintained for different target architectures. The disassembler library `distorm`[2] is based on a trie structure and conceptional similar to our approach. It outperforms other disassemblers with its instruction lookup complexity of $O(1)$. However, the engine still respects instruction sets on a bit granularity and performs a detailed decoding. As we trade computational speed more important than accuracy, `approxis` will stay on a byte granularity level. We consult the previously gained learnings of the mnemonic analysis to improve our process of length disassembling. It should be clear and fair to mention that existing disassemblers aren't designed for our field of application. Processing a large amount of raw data is out of the scope of classical disassemblers. As existing length-disassembler engines reduce the amount of needed decoding mechanisms to a minimum, we introduce an approach to resolve a corresponding mnemonic without respecting any provided opcode maps. Hence, comparing the computational speed of `approxis` with other disassemblers seems less meaningful.

Bytetree Disassembling. To address the introduced requirement *lightweight* (see Sect. 3), `approxis` does not depend on the integration of a specific disassembler engine. The process of disassembling is mainly realized with the already introduced bytetree. We implemented our first prototype of `approxis` in the language C and used a reduced bytetree to generate cascades of switch statements. These statements are used to sequentially process the input instructions and to perform the translation into a corresponding length and mnemonic. The information of the bytetree nodes have been reduced to a minimum core. We only store the amount of counted visited bytes per node and the lengths. Nodes with more than one mnemonic are reduced to a single representative, which is the predominant and most counted mnemonic of the specific node.

[2] https://github.com/gdabah/distorm.

The performance of the bytetree was evaluated by a set of 1318 64 bit binaries. The disassemblies obtained by the bytetree have been compared to the disassemblies obtained by `objdump`. Determining the correct offsets is important to build a solid foundation for further normalization. Thus, it is important to measure the amount of correctly disassembled instruction offsets compared to the set of true instruction offsets. We disassembled all binaries of an *Ubuntu LTS 16.04 x86_64* and extracted the `.text` sections. The determined instruction offsets by `objdump` build our ground truth of *relevant offsets* θ_{rl}. We measured the performance of our bytetree disassembler by verifying all *retrieved offsets* θ_{rt} against our set of *relevant offsets*. An overview of fairly good performance is shown in Table 3 (row `bt-dis`). We denote the performance in values of precision and recall, where

$$\text{precision} = \frac{|\{\theta_{rl}\} \cap \{\theta_{rt}\}|}{|\{\theta_{rt}\}|} \quad \text{and} \quad \text{recall} = \frac{|\{\theta_{rl}\} \cap \{\theta_{rt}\}|}{|\{\theta_{rl}\}|}.$$

Table 3. Precision and recall of `approxis`.

Approach	Precision				Recall			
	Max	Min	Mean (geo./ari.)		Max	Min	Mean (geo./ari.)	
`bt-dis`	100%	84.40%	99.50%	99.51%	100%	92.40%	99.80%	99.80%
`bta-dis`	100%	91.49%	99.76%	99.76%	100%	93.62%	99.84%	99.84%

We examined the binary with the lowest precision (i.e., 84.40%), namely `xvminitoppm`, which converts a `XV` thumbnail picture to `PPM`. Extracting a bunch of false positives underlines our assumption: even with a reliable vast amount of ground truth files, the integration of all instructions is impossible. In case of `xvminitoppm` a lot of overlong Multi Media Extension (MMX) instructions are implemented, which are not present in the bytetree.

Assisted Length Disassembling. Disassembling a unknown binary, with unknown instruction bytes could lead to ambiguous decision paths within the bytetree. Namely, an unknown sequence of input bytes would lead to an exit of the tree structure at a non-leaf node, with multiple remaining lengths and mnemonics. An example in Fig. 3 could not be clearly disassembled with the tree from Fig. 2. To detect those outliers and to extend `approxis` with other features, we integrate our results from Sect. 4. In detail, we use the logarithmic odds of mnemonic bigrams to assist the process of disassembling and to identify reasonable instruction lengths, which could not be resolved by the bytetree itself. As the authors of [7] proposed a disassembler based on a set of logarithmic odds only, we argue that the descent performance of this approach is not sufficient.

As the process of bytetree based disassembling is straightforward, the integration of the absolute logit value λ has not yet been described. We consider

λ as a *value of confidence* if two disassembled and subsequent instructions are plausible or not. So it is more likely that a sequence of instructions is in fact meaningful as long as λ remains small. In contrast, a high value of λ illustrates two subsequent instructions, which are not common at all. We limit the range of the absolute logit λ, where $0 \leq \lambda \leq 100$. This *value of confidence* could be used differently to cope with the goals and requirements in Sect. 3. We first focus on assisting our process of disassembling by resolving plausible instruction lengths. Summarized, we use λ to determine the most plausible offset of a byte sequence, which is not known by our bytetree. The following steps describe the process of assisted disassembling in detail:

1. We use a table of **confidence values** λ_i to evaluate the transition between two instruction sequences denoted by its mnemonic. If a lookup of a subsequent mnemonic pair fails, the action gets penalized with an exorbitant high value. Every retrieved λ_i has to be under a selected threshold τ. We repeat the disassembling with all stored length values of a current node until an offset fulfills the threshold. If none of the length values returns a λ_i under the threshold τ, we select the most common length of the current node.
2. All byte sequences with an unknown byte at **offset zero**, i.e. a byte which is not present in the first level of the bytetree, are penalized by the system. As bytes, which are not present on the first level of the bytetree after processing a fairly large amount of ground truth files, are expected to be not common.
3. A simple **running length counter** keeps track of subsequently repeating confidence values, as these indicate a significant lack of variance, often occurring in large fragments of zero byte sequences or random padding sequences. These non-relevant byte sequences are additionally penalized.

Figure 3 illustrates the process of offset determination. We repeated the process of disassembling the set of 1318 64 bit ELF binaries with assisted length-disassembling. The obtained results in Table 3 (row `bta-dis`) show a significant improvement in the case of precision.

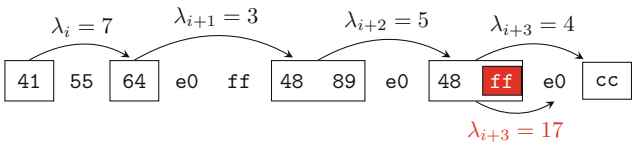

Fig. 3. Selecting certain offsets with a predefined threshold $\tau = 16$.

5.2 Code and Architecture Detection

Beside supporting the process of determining unknown instruction offsets during disassembling, we use the value of confidence to realize two goals: detect code sequences in data and discriminate the architecture of code.

Code Detection. The current implementation of `approxis` could differ between code and non-code fragments in unknown sequences of bytes. As shown in the previous subsection, the value of confidence λ_i is determined for two subsequent instructions to enhance the disassembling process. We use a sliding window approach to consider those values over sequences of subsequent instructions. More formally, we define a *windowed confidence value* ω_x in Eq. 2 as the average of all λ_i within a sliding window, with a predefined size n at offset x. Penalized values overwrite a local value λ_i and thus influence ω_x. The value of ω should be interpreted as a value of confidence over time. A rising value ω underlines the presence of large data fragments. A short rising peak of λ indicates the presence of short and interleaved data. A mid-ranged value of ω indicates the loose presence of instructions or the presence of non-common instructions.

$$\omega_x = \frac{\sum_{i=x}^{n+x} \lambda_i}{n} \tag{2}$$

Architecture Detection. We created a bytetree and a lookup table of λ_i for each architecture of our ground truth. Thus, switching the mode of operation could be realized by simply changing the references of the used bytetree and lookup-table. Mid-ranged values of ω could indicate uncommon sequences of instructions, which we will show later. Large sections of mid-range ω values could also indicate the presence of alternative architectures. We will demonstrate that these variances are significant for different architectures. Sections of code are normally within a range from 1 (high confidence) to 17 (low confidence).

6 Assessment and Experimental Results

In this section we evaluate `approxis` in different fields of application. These assessments focus on the detection of code in different areas of application.

Code Detection: The following evaluation addresses our defined requirement of *robustness* (see Sect. 3). To evaluate the code detection performance in the field of binary analysis, we first examined a randomly selected ELF binary. The result in Fig. 4 illustrates the capabilities of `approxis` to differentiate code from data. Figure 4a shows the initial reduction of confidence by the header. Figure 4b shows that the `.text` section is clearly distinguishable and introduced by the `.plt` section, which is not filled with common sequences of instructions.

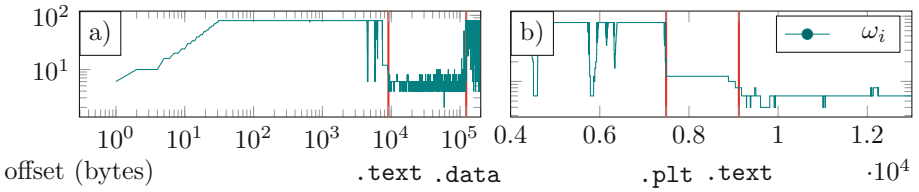

Fig. 4. `approxis` applied on `zip` (64 bit); value of ω_i with cutoff set to 100;

We extracted from a set of 792 ELF binaries the file offsets of different sections with the help of `objdump`. The offsets θ of the sections `.plt`, `.text` and `.data` define points of transition between code and data in each file. To evaluate the code detection performance we inspected the average local value of confidence λ_i for κ preceding and κ subsequent instructions at an offset θ. A transition τ_d from code to data or τ_c from data to code at offset θ is recognized by `approxis`, if the average local confidence differs by a threshold δ (see formula 3). In the case of transitions between `.plt` and `.text` we lowered the threshold from $\delta = 30$ to $\delta = 5$. The ratio of all correctly registered transitions is shown in Table 4.

$$\tau_c = \tau_d = \begin{cases} 1, & \text{if } \left| \frac{\sum_{i=\theta-\kappa}^{\theta} \lambda_i}{n} - \frac{\sum_{i=\theta}^{\theta+\kappa} \lambda_i}{n} \right| > \delta \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

Table 4. Ratio of correctly detected transitions.

Arch	# files	# transition	Detected
x86-64	400	1200	99 %
x86	392	1176	92 %

Architecture Detection. The following evaluation addresses our defined requirement of *versatility* (see Sect. 3). To illustrate the detection process of `approxis` for code fragments of different types, an image with random bytes was generated. Within the random byte sequences we inserted several non-overlapping binaries at predefined offsets. In detail, we inserted a 32 bit (i.e., ELF 64-bit LSB, dynam. linked, stripped) and a 64 bit (i.e., ELF 32-bit LSB, dynam. linked, stripped) version of four different binaries: `wget`, `curl`, `info` and `cut`. As introduced in Sect. 5, `approxis` currently relies on two different bytetrees and mnemonic lookup-tables. By applying both versions on our pathological image, we visualize the changing values of confidence (see Fig. 5a, b).

Similar to the analysis of data and code transitions, we examined the architecture discrimination with the help of 400 randomly selected ELF binaries for each architecture. We extracted the `.text` section of each binary and disassembled them with `approxis` in 64 bit and 32 bit mode. We determined the average of all ω_x for the whole `.text` section of each binary, denoted as $\bar{\omega}$. The distribution of $\bar{\omega}$ for each binary is illustrated in Fig. 6 and outlines the capabilities of `approxis` to discriminate a present architecture.

Computational Performance. The following evaluation addresses our defined requirement of *speed* (see Sect. 3). The execution time of `approxis` was tested on a machine with an `Intel(R) Core(TM) i5-3570K CPU @ 3.40 GHz` with `16 GiB DDR3 RAM (1333 MHz)` and `6 MiB L3` cache. The implementation was done in `C` and compiled with optimization set to `-O3`. As we focus on a possible

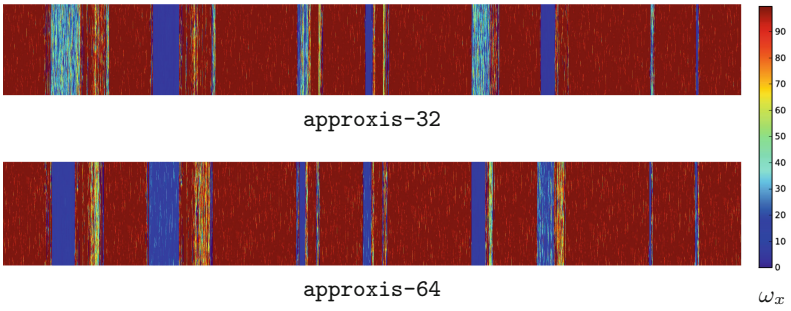

Fig. 5. Comparison of code detection for x86 and x86-64 binaries.

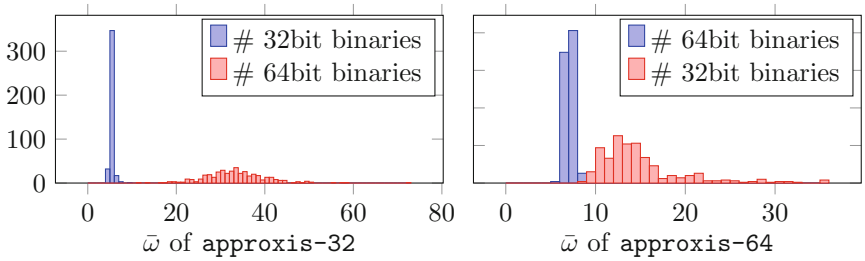

Fig. 6. Architecture detection of `approxis` with a selected bin size of one.

integration in existing approximate matching techniques, we only measured the computation time of the disassembling process and ignored the loading process to memory. It should be mentioned that the current prototype doesn't focus on performance optimization or parallelization. We created three images with a size of `2 GiB` each to evaluate the runtime performance. As we already mentioned in Sect. 5.1, the comparison of `approxis` with other disassemblers is somewhat misleading. As `approxis` outreaches the capabilities of length-disassemblers, but is not able to completely decode x86 instructions, the comparison of those disassemblers should not be understood as a comparison of competing approaches. We applied each disassembler in different modes and optimized our implementation of the distorm engine by removing unnecessary printouts and buffers. Table 5 outlines that the execution time of `approxis` relies on the processed input.

Table 5. Execution time of `approxis` and distorm with different input data.

Execution time				Description
Approxis		Distorm		Disassembler
32	64	32	64	Mode
29.084 s	21.936 s	1 m 20.770 s	1 m 7.772 s	Concatenated set of 64 bit binaries from `/usr/bin`
27.859 s	31.918 s	1 m 43.999 s	1 m 43.046 s	Raw memory dump acquired with `LiME`[a]
1 m 15.521 s	1 m 44.990 s	1 m 58.278 s	1 m 56.192 s	Random sequences of bytes generated with `/dev/urandom`

[a] https://github.com/504ensicslabs/lime

7 Conclusion

In this paper, we demonstrated a first approach to detect, discriminate and approximate disassemble code fragments within vast amount of data. In contrast to previous work, `approxis` revisits the analysis of raw memory with less prerequisites and dependencies. Our approach is a first step to fill the gap between state of the art high level memory examination (e.g., by the usage of volatility) and methods of data reduction similar to those in disk forensics. Our results show the capabilities of `approxis` to differentiate between code and data during the process of disassembling. By maintaining a value of confidence throughout the process of disassembling, we can reliably distinguish between different architectures and switch the used bytetree to obtain a better degree of accuracy. The current implementation shows also a good computational speed.

A next step should be the extraction of features, which are used in a context of approximate matching. Possible methods of subversion (e.g. anti-disassembling) should be considered. A process of exact and inexact matching of code is eligible to consider metamorphic structures and to damp variances in detected code. The approach could be transferred to other domains (e.g. embedded systems).

Acknowledgement. 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 (crisp-da.de).

References

1. Andriesse, D., Chen, X., van der Veen, V., Slowinska, A., Bos, H.: An in-depth analysis of disassembly on full-scale x86/x64 binaries. In: USENIX Security Symposium (2016)
2. Bilar, D.: Statistical structures: fingerprinting malware for classification and analysis. In: Proceedings of Black Hat Federal 2006 (2006)
3. Breitinger, F., Baier, H.: Similarity preserving hashing: eligible properties and a new algorithm MRSH-v2. In: Rogers, M., Seigfried-Spellar, K.C. (eds.) ICDF2C 2012. LNICST, vol. 114, pp. 167–182. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39891-9_11
4. Dolan-Gavitt, B.: The VAD tree: a process-eye view of physical memory. Digit. Invest. **4**, 62–64 (2007)
5. Gupta, V., Breitinger, F.: How cuckoo filter can improve existing approximate matching techniques. In: James, J.I., Breitinger, F. (eds.) ICDF2C 2015. LNICST, vol. 157, pp. 39–52. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25512-5_4
6. Roussev, V., Richard, G.G., Marziale, L.: Multi-resolution similarity hashing. Digit. Invest. **4**, 105–113 (2007)
7. Radhakrishnan, D.: Approximate disassembly. Master's Projects. 155 (2010). http://scholarworks.sjsu.edu/etd_projects/155/
8. Walters, A., Matheny, B., White, D.: Using hashing to improve volatile memory forensic analysis. In: American Acadaemy of Forensic Sciences Annual Meeting (2008)

9. Wartell, R., Zhou, Y., Hamlen, K.W., Kantarcioglu, M., Thuraisingham, B.: Differentiating code from data in x86 binaries. In: Gunopulos, D., Hofmann, T., Malerba, D., Vazirgiannis, M. (eds.) ECML PKDD 2011. LNCS (LNAI), vol. 6913, pp. 522–536. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23808-6_34
10. White, A., Schatz, B., Foo, E.: Integrity verification of user space code. Digit. Invest. **10**, S59–S68 (2013)