Digital Investigation 28 (2019) S12–S21

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

Towards exact and inexact approximate matching of executable binaries

Lorenz Liebler*, Harald Baier

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

ARTICLE INFO

Article history:

Keywords:
Binary analysis
Approximate matching
Malware analysis
Approximate disassembling

ABSTRACT

The application of approximate matching (a.k.a. fuzzy hashing or similarity hashing) is often considered in the field of malware or binary analysis. Recent research showed major weaknesses of predominant fuzzy hashing techniques in the case of measuring the similarity of executables (Pagani et al., 2018). Summarized, well known Context-Triggered Piecewise-Hashing approaches are not very reliable for the task of binary comparisons, as even benign changes heavily impact the underlying byte representation of an original binary. Modifications could be caused by benign or malicious source code changes, different compilers, and changed compiler settings. Approaches based on the extraction of statistically improbable features (Roussev, 2010) or n-gram histograms (Oliver et al., 2013) showed a better detection performance in case of inexactly matching binaries with varying build settings or source code modifications. However, the *inexact* matching of binaries lacks the ability to give more *exact* inferences, i.e., the ability to highlight offsets changed on a byte-level or slight variations within a modified binary. In this work we present apx-bin: an approximate matching implementation for the task of binary analysis and binary matching. Our approach unites *exact* and *inexact* matching capabilities. A first comparison of our approach against four different fuzzy hashing techniques showed major advantages in nearly all of the mentioned scenarios. Previous research underlines the volatile nature of schemes in different scenarios. In contrast, apx-bin is more robust and shows stable results across all considered scenarios. Our scheme, based on a code- and data-related feature extraction, can be further utilized as independent digest or integrated into existing schemes.

Introduction

Different approaches have been introduced for identifying an unknown binary, classifying malicious code or identifying reused code. Among those approaches approximate matching techniques have been suggested. Those algorithms play an important role in security and in the field of digital forensics. In the last years new schemes and variations have been proposed. However, the application of those algorithms on executable binaries often showed divergent results. Even if predominant implementations like ssdeep (Kornblum, 2006) are widely used, the applicability is often discussed and extended evaluations were necessary. Recent research (Pagani et al., 2018) underlines the reasons for this inconsistency in reputation and perceived usability. Originally developed for the task of data reduction, fragment detection, or similarity analysis of digital corpora, many fuzzy hashing schemes are not able to deal with the variety of a binary lifecycle. Newer schemes consider a possible randomization of the underlying sequences and damp occurring variances on a byte-level. Especially tlsh (Trend Micro's Locality Sensitive Hashing) has major advantages in the case of robustness, as the underlying scheme compensates variances by the extraction of n-gram features and by a frequency-based similarity matching.

The conceptual lifecycle of code introduces a variety of optional or mandatory modifications. Those could have different impacts on the underlying byte structure and thus, mainly influences the capabilities of fuzzy hashing schemes. As shown in Fig. 1, modifications could occur at different stages of programming, building, linking, and loading. Examined from a benign perspective the possibilities of modification are manifold and their occurrence are very likely. The given overview casually outlines the obvious fact that a broad variety of modifications do not have to be malicious at all. Even considerable small changes in the compiler configuration heavily influence the underlying byte representation (e.g. by the

* Corresponding author.
E-mail addresses: lorenz.liebler@h-da.de (L. Liebler), harald.baier@h-da.de (H. Baier).

https://doi.org/10.1016/j.diin.2019.01.027

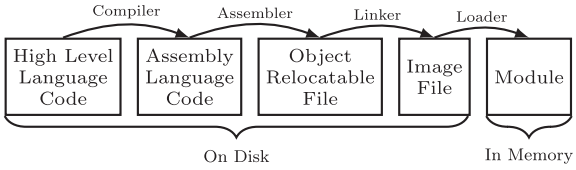

Fig. 1. Compilation of a binary until it is loaded into memory.

adaptation of optimization flags). The introduced pitfalls only belong to the bare benign cases, where malicious motivated obfuscation adds a new dimension of challenges a scheme should deal with.

Contributions. In this work we introduce apx-bin, an approximate matching scheme for the task of binary similarity matching. Different schemes have been constantly proposed and promoted in the course of digital forensics. Some schemes have been utilized for the task of binary analysis. We will give a short overview of four different schemes and summarize related research in Section 2.

We focus on schemes recently evaluated by other research, where authors gave valuable insights into the backgrounds of ambiguous matching results. Therefore, three different scenarios are suggested, reflecting several phases of a compilation pass (Pagani et al., 2018). Our approach is based on mrsh-mem, an algorithm introduced in the field of memory forensics and a derivative of mrsh-v2 (Liebler and Breitinger, 2018). We will review the utilized scenarios, their key findings, and mrsh-mem in Section 3. We also outline our choice of sticking to a specimen of previously doomed Context-Triggered Piecewise-Hashing (CTPH).

In Section 4 the discussed key findings are used to adapt mrsh-mem, to add new functionalities of feature selection, and to integrate a multi-layered extraction. We are able to prove several statements of Pagani et al. (2018) by the definition of a provisional score-model in Section 4.1. Moreover, our evaluation underlines the difficulty to withstand all introduced challenges. In Section 4.2 we define our proposed scheme and the beneficial distinction between code- and data-related features. The evaluation of our prototype will be discussed in Section 5. Our proposed candidate outperforms four competitive fuzzy hashing implementations in most of the considered tests. However, the overall strength of the approach is the stable inference and matching capabilities across all scenarios. Finally, we conclude our work and postulate future improvements in Section 6.

Binary matching

We give a short overview of the considered similarity schemes in Section 2.1. Evaluations which consider those schemes in the field of binary analysis are introduced in Section 2.2. For a more detailed explanation of the depicted schemes we refer to the original publications.

Similarity digests

In the case of cryptographic hash functions (e.g. SHA-256) small changes on an input drastically changes the final output digest. This is also denoted as the avalanche effect. To detect similar files, even after small changes have been applied on an exact copy, called similarity preserving hash functions, have been developed (a.k.a., fuzzy hashing or approximate matching). The National Institute of Standards and Technology (NIST) defined four possible use cases approximate matching techniques could deal with: similarity detection, cross sharing detection, embedded object detection and fragmentation detection (Breitinger et al., 2014). The schemes for creating similarity preserving digests could vary in a broad sense and are difficult to formalize in a uniform way. An approach could be described by its underlying steps of processing: the *selection of features* and the *creation of the digest* (Ren, 2015).

In the case of ssdeep (Kornblum, 2006) and mrsh-v2 (Breitinger and Baier, 2012) the concept of CTPH mainly relies on the extraction of chunks by the utilization of context-based patterns (e.g., the utilization of a pseudo-random function). Those *chunks* are afterwards hashed as a *sequence* and stored into a similarity digest. A digest could be represented as a concatenation of the hash values (ssdeep) or as Bloom filter (mrsh-v2). In the case of ssdeep, the Levenshtein distance is used to compare two 80 byte digests with each other. In the case of mrsh-v2 the Hamming distance is used to compare two or multiple Bloom filters. We will explain the details of mrsh-v2 and its extension mrsh-mem in Section 3.2.

In contrary, sdhash (Roussev, 2010) is based on the extraction of statistically improbable features with a length of 64 bytes. The identified *bag* of features are hashed into a Bloom filter. The distance between two digests is also determined with the Hamming distance. Oliver et al. (2013) proposed the *Trend Micro's Locality Sensitive Hashing* (tlsh). a scheme also considered in the field of binary analysis. The scheme extracts six triplets of 5-byte windows in a sliding window fashion. Thus, a large amount of relatively small features are extracted. The *bag* of features is afterwards hashed into a frequency based representation. Quantile-based analysis help to damp variances beside the n-gram based extraction. A distance score is calculated by additionally considering identical files and the file size.

An overview of the different concepts could be seen in Fig. 2. In the case of ssdeep and mrsh-v2 *chunks* are extracted as fixed *sequences* and further processed. In the case of tlsh and sdhash *bags* of extracted *triplets* (tlsh) or *bags* of extracted *blocks* (sdhash) are processed respectively. It should be clear that *bags* do not consider the order of elements. In contrast, *sequences* consider the order of elements inside of a chunk. The schemes could utilize different score ranges and measurements. Where tlsh utilizes a distance range from 0 (identical files) to $1000+$. The remaining schemes (i.e., ssdeep, mrsh-v2 and sdhash) utilize a score between 0 and 100 (where higher is more similar). The score of tlsh could be normalized to range between 0 and 100. The thresholds could vary and should be adopted to a specific use case.

Summary. As concepts of schemes differ heavily, the inference of the similarity measurements has to be discussed. The short insight into some schemes emphasizes the fact that obtained scores need to be considered differently. For example, a high value of similarity in the case of mrsh-v2 states a high similarity in the case of completely identical byte *sequences*. An user has to consider that such a score gives more *exact* inferences, albeit, is error-prone in the case of even small changes. It could be questioned if reducing the capabilities of different schemes to the task of matching two binaries (e.g., binary x equal to binary z) is always sufficient or if we should strive for a more diversified attestation (e.g., which offsets have been adapted).

Binary analysis

Research discussed the usability of the schemes in the context of binary or malware analysis. A predominant representative in this field is still ssdeep[1,2] In the past, evaluations of the different approaches lead to unclear results with missing insights. Most often

[1] https://www.virustotal.com/(last access 2019-01-20).

[2] https://www.nist.gov/software-quality-group/approximate-matching(last access 2019-01-20).

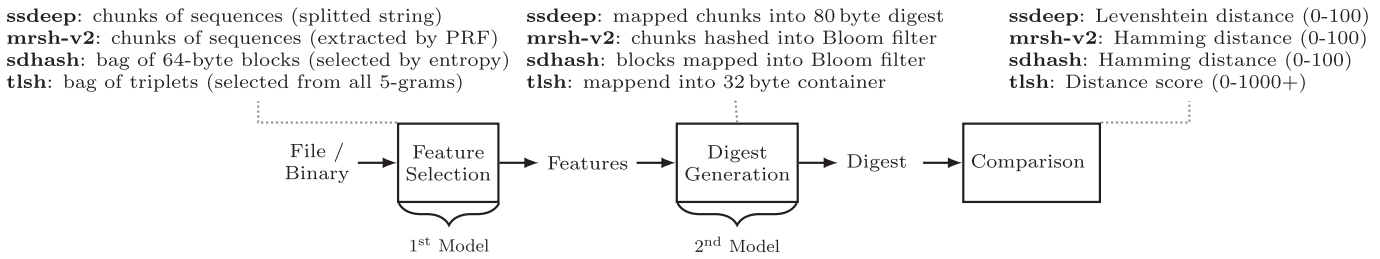

Fig. 2. Simplified steps of similarity digesting schemes; similar to Ren (2015).

matching results have been published without giving any reasoning why specific schemes perform differently for specific scenarios. In addition, test scenarios could vary in a broad shape, depending on the utilized datasets and test methodology. An extensive overview of different research was given by Pagani et al. (2018). In short, French and Casey (2012) claimed that ssdeep and sdhash have varying capabilities to match malicious binaries. Oliver et al. (2013) and Azab et al. (2014) emphasized the utilization of tlsh over all competing approaches. The evaluation of Upchurch and Zhou (2015) disproved those results. The application of the techniques on selected sections (i.e., on code sections only) also showed a significant impact (Li et al., 2015). For a more detailed overview of the different publications, their evaluations, and results we refer to Pagani et al. (2018).

Pagani et al. (2018) enlightened previous observations with more reproducible results and inferences by the utilization of controlled scenarios. The authors inspected the capabilities of four different fuzzy hashing techniques in the context of binary analysis: ssdeep, sdhash, mrsh-v2 and tlsh. Therefore, the authors introduced three different scenarios for measuring the performance of the four selected candidates. Afterwards, the results have been analysed and insights were given why specific schemes tend to fail in specific scenarios. We will give a comprehensive explanation of the different scenarios and the results in Section 3.1. Summarized, tlsh and sdhash outperformed ssdeep (i.e., CTPH-based techniques) in many of the proposed evaluation scenarios. Source code modifications or assembly manipulations have small influences on tlsh and its utilized n-gram frequencies. Varying compiler versions or options have barely any influence on sdhash. Thus, the recommended approaches could vary for different use cases.

Those results match with previous research discussed for non-binary and binary samples by Oliver et al. (2014). The work evaluated the capabilities of three schemes (i.e., sdhash, ssdeep and tlsh) applied on various types of randomly manipulated data. The authors focused "on situations where the file is deliberately modified by an adversary using randomization as the key component". Noteworthy scenarios are (benign) source code modifications and malicious obfuscations (metamorphism). The application of locality sensitive hashing (e.g., tlsh) showed a better performance and was considered as more difficult to exploit.

Summary. Recent research enlightens the reasons of failure in the case of CTPH-based binary analysis. The authors gave reasons for the fluctuating evaluation results of previous research. Considering the functionality of CTPH (i.e., ssdeep or mrsh-v2) and a reasonable amount of occurring changes those results are plausible. As we will outline later, the byte-wise extraction of fixed *sequences* is error-prone. We could conclude that CTPH was doomed in the field of extended binary matching, as the overall target domain implies manifold possibilities of variation. Especially in the field of malware analysis, where code authors have a motivation for obfuscation and modification to evade signature-based analysis, those concepts will struggle.

Background

In Section 3.1 we will give an overview of the different use case scenarios utilized to design and evaluate our current prototype implementation of apx-bin. Afterwards, we give a short insight into CTPH in Section 3.2 by the introduction of mrsh-mem and outline some details why CTPH performs worse in the context of binary analysis.

Evaluation scenarios

Pagani et al. (2018) examined the different schemes by the creation of three different scenarios. An overview of the different scenarios could be seen in Fig. 3. The first scenario **Library Identification (I)** focuses on the task of detecting embedded object files inside a binary. Five small example executables were statically-linked against five popular Linux libraries. All executables are compared against each object file. Denoted as *Object-to-Program Comparison* (I.1), the test was performed for the whole executable (.o) and the text segment only. The *Impact of Relocation* (I.2) considers relocations performed by the linker or dynamic loader. The.text segments after relocation of library object files are analysed. The original object file, the relocated object file, and the final executable file are compared against each other.

The results of matching object files to statically-linked executables showed advantages for sdhash and for mrsh-v2. Due to many small files tlsh struggles. In the case of ssdeep no single match was found. Again, sdhash showed a stable performance in the case of matching relocated code sections. In two of three comparisons sdhash outperformed tlsh and mrsh-v2. The case of comparing relocated object files with the original object showed advantages for tlsh. Ssdeep again always generated zero similarities. The

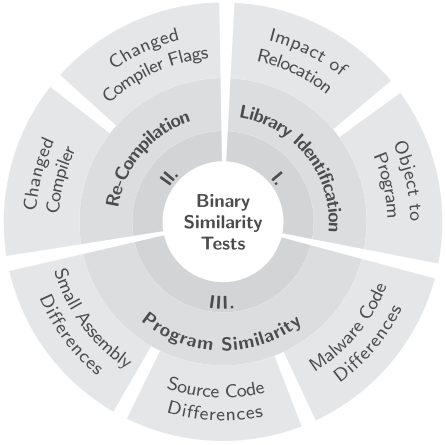

Fig. 3. Overview of test scenarios proposed by Pagani et al. (2018).

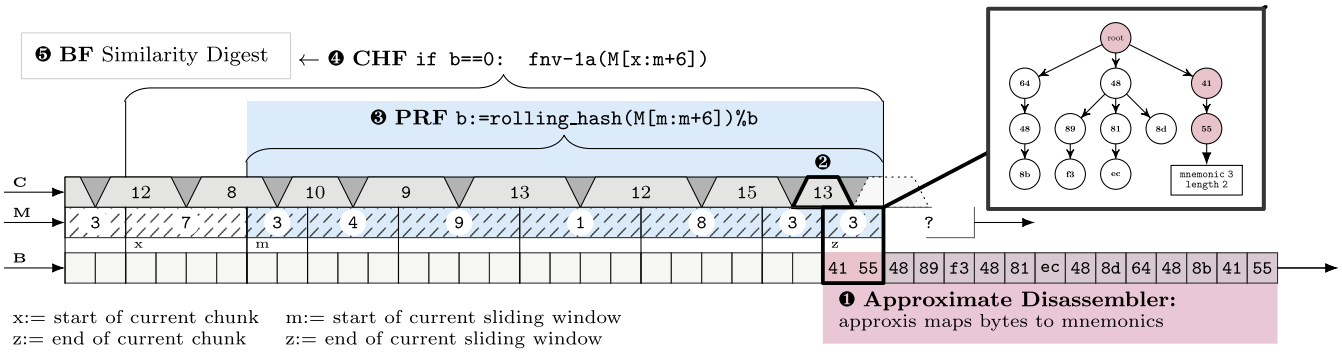

Fig. 4. Over-simplified model of the multi-layered processing steps of mrsh-mem (Liebler and Breitinger, 2018). The processing consists of multiple buffers of the raw input byte sequences (**B**), the mapped mnemonics (**M**) and the buffer which stores the value of code confidence for two subsequent instructions (**C**). A trained prefix tree maps the byte sequences to a integerized mnemonic (Liebler and Baier, 2017).

authors note that the relocation changes 10% of the object bytes in total and thus, obviously does break most of the features. They also note that splitting the libraries, linking only a small subset to the executable, discrepancies in the file size and often occurring changes make most of the schemes fail at least once.

The second scenario covers the detection of the same program after **Re-Compilation (II)**. First, the *Effect of Compiler Flags* (II.1) was inspected, i.e., the program was compiled with different flags and the same compiler. A process which heavily influences the final structure of the executable on several levels (i.e., O0, O1, O2, O3, Os). Second, the impact of compiling the same program with *Different Compilers* (II.2) was inspected.

The compilation process makes CTPH approaches again to struggle. In the case of differing compiler flags ssdeep performs worst again. In contrary, sdhash yields similarity scores across all optimization levels. The authors outline that tlsh showed promising results by accepting a slightly increased false positive rate. For larger files, all the mentioned approaches are outperformed by sdhash. An important observation by the authors is the fact that most of the matches are solely dependent on constant fragments contained in data sections (e.g.,.rodata). Only in some cases, where code remains constant across different settings, schemes are able to match code-related sections.

The third Scenario covers the evaluation of **Program Similarity (III)**. The scenario consists of three tests which consider adaptations to the underlying code. The test of *Small Assembly Differences* (III.1) randomly inserts an increasing amount of NOP instructions into a binary (i.e., ssh-client). Moreover, an increasing amount of instructions are swapped. In the case of *Minor Source Code Modifications* (III.2) the authors suggest the adaptation of the ssh-client application in three different ways: *Different Comparison Operator*, *New Condition* and *Change a constant value*. In the course of *Source Code Modifications on Malware* (III.3) two real-world malware samples are evaluated. The source of Mirai (Linux) and Grum (Windows) are adapted in three different ways: *C&C Domain Adaptation*, *Evasion* and *New Functionality*.

The insertion or swapping of just ten instructions showed a major impact on the score values on ssdeep, mrsh-v2, and sdhash. Even two simple NOP instructions could cause ssdeep scores to drop to zero. The authors discussed three major factors: compiler-optimized paddings between functions, linker-based paddings at the end of sections and the positioning of the instructions. Increasing paddings are additionally causing section shifts and thus, all global references are updated (including jump tables). In contrary, tlsh kept promising high similarity rates, because of its underlying frequency-based nature of n-grams.

Summary. We could summarize the main findings of Pagani et al. (2018). First, the distinction between data and code is of crucial importance and has an overall impact on the final inferences. Pagani et al. (2018) underlined this fact by the inspection of malware samples originally evaluated by Upchurch and Zhou (2015). Second, even small changes on the (source) code of samples or additional insertions could influence the overall binary and code structure in a broad way. This especially has a great impact on CTPH-based approaches. Summarized, sdhash and tlsh clearly outperformed CTPH-based schemes. Each of both have their strengths and weaknesses in different disciplines.

Approximate matching (mrsh-mem)

The original mrsh-v2 scheme is a member of Context-Triggered Piecewise-Hashing (Breitinger and Baier, 2012) and operates on the byte sequence of a given input. A given sequence is divided into chunks by the definition of a fixed modulus b (common values are $64 \leq b \leq 320$ bytes). Therefore, the algorithm iterates over the byte stream in a sliding window fashion, rolls through the sequence byte-by-byte, and processes 7 consecutive bytes at a time. A current window is hashed with a *Pseudo Random Function* (PRF) which returns a value between 0 and b. This value defines the trigger point (e.g., the value is equal to zero) and thus, defines the boundary of a current chunk. If the PRF behaves pseudo random, the probability of a hit is reciprocally proportional to b and each chunk is approximately the size of b bytes. In the case of mrsh-v2, the authors utilized a fast rolling hash function. A defined chunk is afterwards hashed with a *Chunk Hash Function* (CHF) and the hashes are stored in a Bloom filter (Bloom, 1970). The overall concept bares two major weaknesses: First, even small adaptations to the underlying byte structure could cause the definition of completely different chunk boundaries by the utilized PRF. Second, even if the chunk boundaries are not directly influenced by a differing byte, a single byte modifies a current sequence and thus, the extracted chunk hash value generated by the CHF.

For a better overview of the following explanations we refer to Fig. 4. In the case of mrsh-mem the technique does not process the raw byte sequences (**B**), but rather a mapped (mnemonic) stream (**M**). To map a varying sequence of bytes to a single representative ❶ the authors utilize a trained prefix tree of x86 and x64 ground truth assemblies (Liebler and Baier, 2017). The trained model returns a mnemonic and a corresponding instruction length. The decoding is based on an interpretation of bytes and performed without any interpretations on a bit-level.

The integration of an approximate disassembler empowers the mrsh-mem tool not only to map sequences of bytes to integerized mnemonics, but also gives the opportunity to differ between code

and data. The authors proposed the additional extraction of bigram frequencies to support the decoding process. The concept of code detection relies on those extracted frequencies. Beside the mapping of a sequence of bytes, a currently translated mnemonic and its predecessor are looked up. A probability for a current pair is determined ❷ and this determined *value of code confidence* is stored into a separate buffer (**C**).

The sliding window iterates over mapped bytes instead of the raw bytes ❸. Considering relocations and small adaptations, this damps occurring changes on a byte-level and stabilizes the extraction of chunks by a utilized PRF. As soon as a chunk boundary was defined, mrsh-mem hashes the sequence of mapped bytes with a CHF ❹. Common CHFs are MD5, SHA or FNV-1a, where mrsh-v2 and mrsh-mem utilizes FNV-1a. Lastly, all chunk hashes are translated into a digest. In the case of mrsh-mem, a single large Bloom filter is used to store the hash values ❺.

The current implementation supports several parameters which should be considered. The approximated chunk size could be controlled by the fixed modulus b. Liebler and Breitinger (2018) depicted a default value of $b = 64$ to lower the impact of smaller fragmentations. As already mentioned, within the buffer **C** the values of confidence are stored for two subsequent mappings (bigrams). The probabilities are saved as absolute logarithmic odds (logit). Offsets with a value of confidence below 30 are considered as possible code offsets within **M**. Inspecting our example in Fig. 4, all elements are considered as code offsets, as the values range between 8 and 15. In addition, the original paper recommends the filtration of chunks with at least 30% of the mappings considered to be code offsets, defined by a threshold τ_{min}. The ratio of considered code offsets to the overall amount of offsets is also denoted as the *code coverage c*.

Approximate binary matching

In Section 4.1 we prove previously discussed findings of Section 3.1 and inspect the influence of binary-related feature extraction and interpretation. The insights have been used to design a first score function in Section 4.2 which was finally integrated into apx-bin.

Pre-evaluation of scenarios

To further prove and inspect the impact of data- or code-related feature selection, we adapt the original mrsh-mem implementation in two ways:

Selection of chunks. First, to control the feature selection process we extend the original and code-based extraction by an extended parametrization. We do not solely focus on the extraction of code fragments and instead consider chunks of both types. In detail, we adapted the process of chunk extraction by the possible distinction of its code coverage c. By extending the minimum value of code coverage τ_{min} with an additional maximum value of code coverage τ_{max} we could control the feature selection process in terms of code- and data-selection. Defining a parameterized range of coverages (i.e., $\tau_{min} \leq c_i \leq \tau_{max}$) empowers us to inspect and reassess the previously mentioned impact of feature selection across all scenarios during our pre-evaluation. A parametrization of $\tau_{max} = 100$ and $\tau_{min} = 0$ would lead to the extraction of all chunks.

Multi-layered extraction. Second, we do not only extract the mapped buffer of mnemonics and also further process a chunk in its byte-representation. As the original mrsh-mem approach utilizes the extraction of the mnemonic buffer M only, we additionally extract the original byte buffer B of a selected chunk. Both chunks are hashed by the CHF and stored into a Bloom filter. In the case of comparing a file against a digest, we additionally weight chunk hits on a byte-level by a factor of 1.5. Scores on a mnemonic-level are kept unchanged and treated as less meaningful. This should damp collisions caused by the mapping of the byte sequences into an intermediate and simplified representation.

Score of the Pre-Evaluation. We propose the multi-layered extraction of chunks with additional steps of feature selection. We define a provisional score-model over two final sequences of extracted chunks. The preliminary similarity score sim_{pre} is shown in Equation (1). The total amount of extracted chunks is denoted as z. Hits are already represented by their previously determined values of coverage: A sequence of hits extracted from the mnemonic buffer (i.e., $\langle m_0, m_1, \ldots, m_{y-1} \rangle$) and a sequence of hits (i.e., $\langle b_0, b_1, \ldots, b_{y-1} \rangle$) extracted from the byte buffer. We weight scores on the byte-level by a factor of 1.5. The overall score tends to overshoot. We define an additional cut-off function which limits the score to a maximum value of 100. By varying the values of τ_{max} and τ_{min}, we could inspect the dependencies of feature selection.

$$sim_{pre} = min\left(\frac{\left(\sum_{i=1}^{y} f_c(b_i)\right) \cdot 1.5 + \sum_{i=1}^{y} f_c(m_i)}{z},\ 100\right),$$
$$\text{where } f_c(x) = \begin{cases} 1, & \text{if } \tau_{min} \leq x \leq \tau_{max} \\ 0, & \text{else} \end{cases} \text{ and} \tag{1}$$
$$\forall x, y \in \mathbb{R}\left(\min(x, y) = \frac{x + y - |x - y|}{2}\right).$$

Library identification

We inspect the influence of feature selection for the *Object-to-Program Comparison* of Scenario 1. In Fig. 5 the results of matching a whole object file against a library are presented. As could be seen, the plots visualize solid similarities by the extraction of code fragments (i.e., $\tau_{min} \geq 40$). Shifting the focus to non-code related fragments decreases the similarity scores significantly. In Fig. 6 the results for matching only the text sections to a library file are shown. Considering the processing of a text section (i.e., a section which mainly consists of code) the plots visualize that fewer features could be extracted. This is obvious, as by shifting the feature extraction to data-related features (i.e., $\tau_{max} \leq 80$) will be barely successful. In Figs. 5 and 6 similarity scores of the non-matching libraries (negative) are always near to zero. However, a remarkable amount positive cases constantly yielded values near to zero. By inspecting all positive cases and all chunks (i.e., $\tau_{max} = 100$, $\tau_{min} = 0$) with a similarity score below 10 %, we could see that most of those files are very small. Such files are hardly attributable even by a manual inspection. Histograms visualize the dependency between similarity scores and a corresponding file size.

Re-compilation

In the course of the second scenario we focus on the *Effect of Compiler Flags* and the *Effect of Compiler Changes*. Our evaluation underlines the discussed findings in Section 3.1, where the extraction of constant data fragments out of the.rotdata section shows better results in the case of optimizations. As could be seen in Fig. 7, the extraction of code fragments (i.e., $\tau_{max} = 100$, $\tau_{min} \geq 80$) showed ambiguous scores for positive and negative classes. In contrary, the extraction of data fragments (i.e., $\tau_{max} \leq 80$, $\tau_{min} = 0$) showed very good capabilities for the task of matching two binaries compiled with different flags.

In the case of switching compilers for the same applications, the picture slightly changes again. As could be seen in Fig. 8, all parametrizations show a good performance with low similarities for

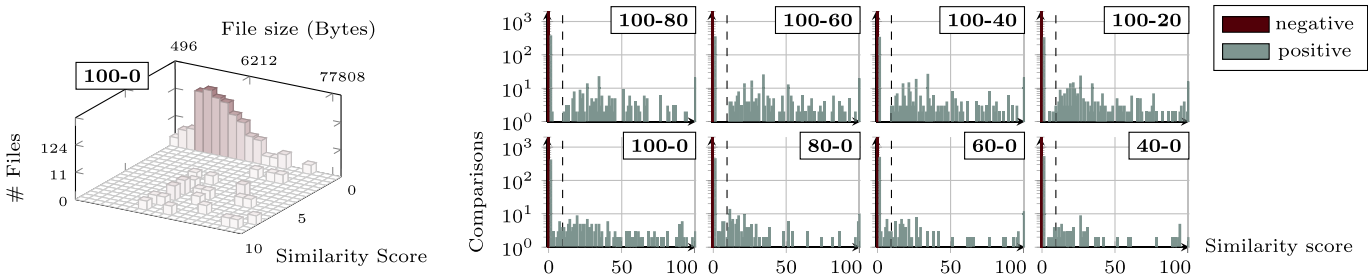

Fig. 5. Scenario I. Library Identification 1. Object-to-Program - Whole Object File. (Labels specify depicted values of τ_{max} - τ_{min}).

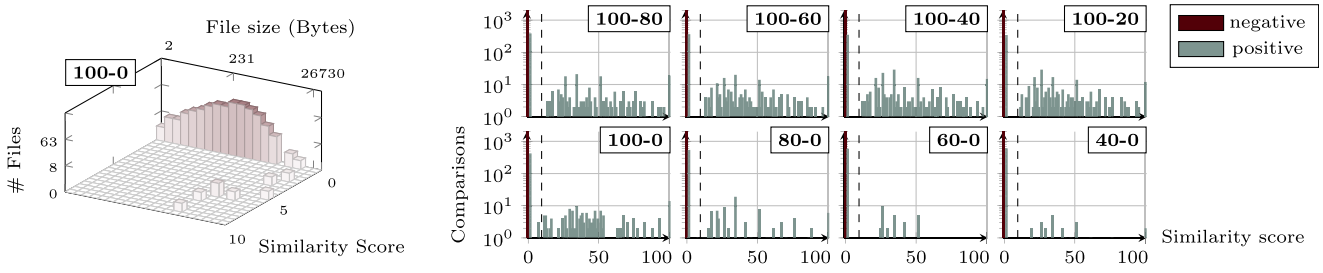

Fig. 6. Scenario I. Library Identification 1. Object-to-Program - Text Section. (Labels specify depicted values of τ_{max} - τ_{min}).

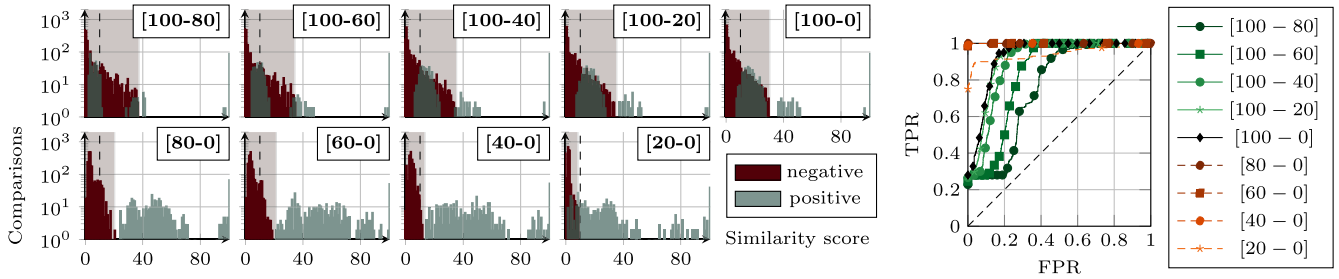

Fig. 7. Scenario II. Re-Compilation 1. Optimization flags. Overview of score distributions for apx-bin with varying feature selection. (Labels specify depicted values of τ_{max} and τ_{min}).

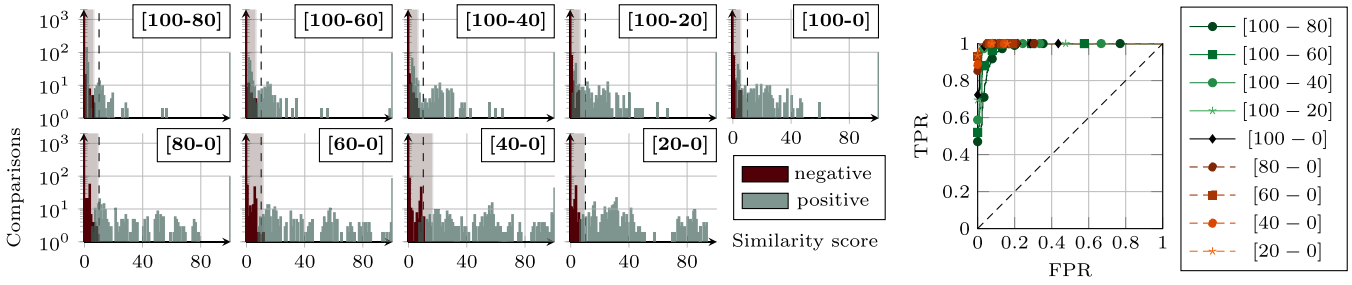

Fig. 8. Scenario II. Re-Compilation 2. Compiler Variation. Overview of score distributions for apx-bin with varying feature selection. (Labels specify depicted values of τ_{max} and τ_{min}).

non-matching cases (negative). But again, the examination of data-related chunks improves the bimodal distribution. As a trade off the matching rates for false matches slightly increase but most often stay under 10%.

Summary. The pre-evaluation underlines the ambivalence of the different use cases and the conclusions of previous research. The task of detecting used libraries primarily relies on the matching of similar code structures. In the case of matching binaries across different compilers and different configurations, the schemes rely on the extraction of constant data fragments. We will address this challenge in the following subsection.

Details of apx-bin

Our current implementation utilizes the overall CTPH-based characteristics of mrsh-v2 and mrsh-mem. We keep several of

the already mentioned parameters and will outline concepts which have been transferred, parametrized or partially modified.

CTPH Parameters and Concepts. The parameter b directly influences the expected block size during a running pass. A small value of b respects small interleaved sequences, whereas a big value of b considers large portions of matching sequences. As discussed in Section 3, even small changes significantly influence the overall byte structure. We consider a larger value of b as more error prone and suggest to lower the value to $b = 32$. This damps variances occurring and respects even small and interleaved sequences. We do not use a minimum number of required consecutive chunks before a hit would be considered (i.e., $min_{run} = 1$). In contrast, we define a minimum chunk size of 16 bytes which skips considerable small sequences. We keep the approach of chunk extraction similar to mrsh-mem. The implementation processes the mnemonic stream in a sliding window of seven consecutive instructions. Considering relocations and small adaptations, this should encounter changes occurring on a byte-level. For our PRF we utilize the rolling hash implementation of mrsh-v2 and we keep FNV-1a as CHF.

Score of apx-bin. Before formulating our depicted score-model, we summarize the most important key findings of previous research and our pre-evaluation. First, the correct matching of binaries in all of the mentioned scenarios requires the consideration of code and data in an appropriate ratio. The small proportion of constant data fragments should be rated in balance to the predominant code fragments. Second, a match on the byte level is considered to be more meaningful than a matched sequence of mapped bytes. We should be aware of actually non-matching but colliding chunks due to the mapping of bytes to integerized mnemonics. Third, in the case of sdhash and tlsh the extraction of bags showed major advantages. In the course of this work we stick to the extraction of CTPH-like sequences to further inspect the boundaries of this type. We could imagine the extension of the multi-layered extraction by an additional Counting Bloom filter.

We again consider a sequence of all extracted chunks represented by their specific values of *code coverage* (i.e., $\langle c_0, c_1, \ldots, c_{n-1} \rangle$). All occurring hits (i.e., contained in target file) differ by their mnemonic and byte representation. We define those hits by their values of code coverage for matching byte chunks (i.e., $\langle b_0, b_1, \ldots, b_{y-1} \rangle$) and for matching mnemonic chunks (i.e., $\langle m_0, m_1, \ldots, m_{z-1} \rangle$). A chunk could match either as a sequence of mnemonics and bytes, or as a sequence of mnemonics only ($y \leq z$). Chunks are considered as meaningful by the definition of two filters f_c and f_d (see Equation (2)). Thus, we focus on the extraction of chunks with high or low values of code coverage. Depicting $\tau_{max} = 50$ and $\tau_{min} = 50$ would obviously lead to the extraction of all chunks.

$$f_d(x) = \begin{cases} 1, & \text{if } 0 \leq x \leq \tau_{min} \\ 0, & \text{else} \end{cases}$$
$$f_c(x) = \begin{cases} 1, & \text{if } 100 \geq x \geq \tau_{max} \\ 0, & \text{else} \end{cases} \tag{2}$$

We formulate the final similarity score by the definition of two scores for data (γ_d) and code chunks (γ_c). Each of the scores represents the ratio of the total amount of filtered chunks to the overall amount of filtered hits. We define two additional weights to respect the outlined target domain. Hits on a byte-level are weighted with a factor of 2 and hits on data chunks are weighted by a factor of 1.5. As the current score can overshoot, we additionally define a cut-off function which limits the scores to 0.99. The Final score is the average of both scores γ_d and γ_c. It should be clear that our scheme inverts the thresholds of the pre-evaluation (i.e., dropping chunks with $\tau_{min} \leq c_i \leq \tau_{max}$).

$$sim_{bm} = \frac{\gamma_d + \gamma_c}{2}, \text{ where}$$
$$\gamma_d = min\left(\frac{\left(\sum_{i=0}^{y-1} f_d(b_i) \cdot 2 + \sum_{i=0}^{z-1} f_d(m_i) \right) \cdot 1.5}{2 \cdot \sum_{i=0}^{n} f_d(c_i)},\ 0.99 \right), \tag{3}$$
$$\gamma_c = min\left(\frac{\sum_{i=0}^{y-1} f_c(b_i) \cdot 2 + \sum_{i=0}^{z-1} f_c(m_i)}{2 \cdot \sum_{i=0}^{n-1} f_c(c_i)},\ 0.99 \right).$$

Competitive evaluation

We evaluate apx-bin in the course of the three introduced scenarios and by comparing them to the results from Pagani et al. (2018). In Fig. 9 the bimodal distribution of scores for several scenarios are visualized. As could be seen the current model is optimized for passing most of the mentioned tests. In the case of different compiler flags the large amount of shared code fragments still causes a remarkable amount of falsely matched scores. We discuss the performance of apx-bin compared to the other schemes in the remainder of this section.

Library identification

Our prototype outperforms all of the other schemes in the course of identifying libraries in different object files or text sections. As could be seen in Table 1, we depicted a threshold which causes zero false positives by keeping a relatively high true positive rate (i.e., the ratio of true positives to the positive class size). Similar to the original mrsh-v2 implementation all files could be processed by apx-bin (i.e., causing no errors).

In the evaluation of the impact of relocation apx-bin shows constant scores in all of the three comparisons. It should be considered that the step of approximate disassembling should damp most of the occurring relocations. However, due to the currently proposed score model, the matches on mapped sequences are considered as less meaningful. As could be seen in Fig. 10 apx-bin still performs better than most of the competing schemes on average.

Re-compilation

In Fig. 11 and Fig. 12 apx-bin is compared in the course of matching binaries with differing compiler flags or differing compilers respectively. Each column represents the variation between different combinations, not displaying the comparison of the same target class (e.g., O1–O1 or clang–clang). The red areas inside the plot visualize the maximum score of an occurred false match. The highest false matches most often occurred during the comparison of binaries from the same target class. Thus, those scores are not directly represented within the plot, but considered during the determination of all scores. The true positive rates are displayed as percentage values inside the plots. The thresholds are chosen to produce zero false positives.

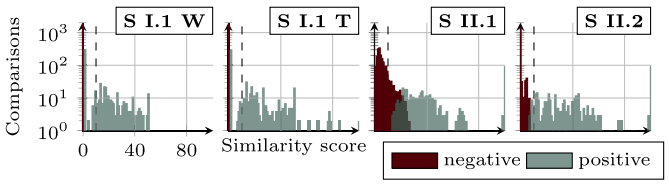

Fig. 9. Scenario I and II: Overview of score distributions for apx-bin. (S: scenario; W: whole library; T:.text section).

Table 1
Scenario I. Object-to-Program Comparison 1. True and False Positive Rate compared to results Pagani et al. (2018).

Alg	.o		.text		
	TPR %	FPR %	TPR %	FPR %	Err %
ssdeep	0	0	0	0	0
mrsh-v2	11.7	0.5	7.7	0.2	0
sdhash	12.8	0	24.4	0.1	53.9
tlsh	0.4	0.1	0.2	0.1	41.7
apx-bin	**48.9**	**0.0**	**44.1**	**0.0**	**0**

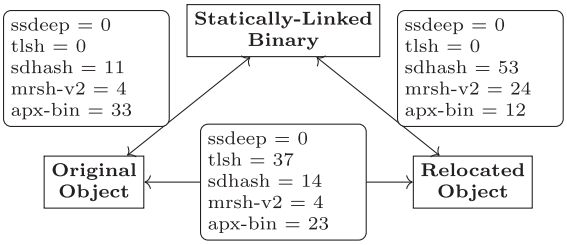

Fig. 10. Scenario 1. Impact of Relocation 2. Average values of similarity for apx-bin compared to results Pagani et al. (2018).

As could be seen in Fig. 11, apx-bin shows stable scores across all differing compiler flags. On average the score model outperforms all the other approaches with an average true positive rate of 61.6%. In the case of varying compilers (Fig. 12) apx can compete with the leading scores of mrsh-v2. Summarized, the current model of apx-bin stabilizes the score values across different settings.

Program similarity

The analysis of program similarity consists of small assembly differences and randomly swapping of instructions. The original paper showed that after the insertion of 100 randomly placed NOP instructions, most of the schemes dropped scores below 40% (i.e., ssdeep, mrsh-v2 and sdhash). In the case of randomly swapping instruction, the scores dropped after 10 swaps below 40% (i.e., ssdeep, mrsh-v2 and sdhash). The score of tlsh stays almost close to

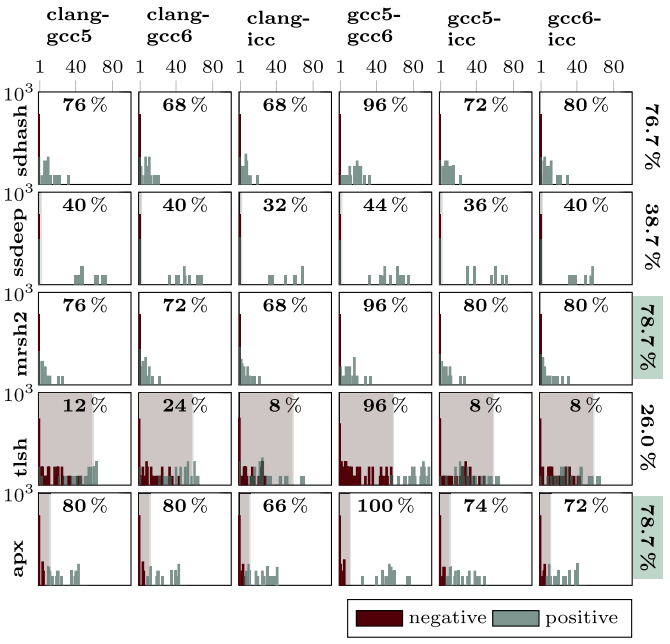

Fig. 12. Scenario II. Re-Compilation 2. Different Compilers: competitive evaluation of apx-bin. (y axis denotes the number of matches from 0 to 1000; x axis denotes the similarity score).

100% and clearly outperforms competing schemes.

In Fig. 13 we visualize the performance of apx-bin by splitting the overall score value (apx-bin) into its two score components γ_d (apx-data) and γ_c (apx-code). The plot clearly visualizes the strength of the proposed score model by additionally weighting the extracted chunks of data. The influence of instruction swapping or the insertion of NOP instructions impact the overall rate after 1000 instructions. However, even after 10000 performed changes, apx-bin shows score values above 40%.

In the course of minor source code modifications we inspect the impact of differing the comparison operator, defining new conditions and changing a constant value in the source code of the ssh-client. Inspecting the results in Table 2, apx-bin again shows stable

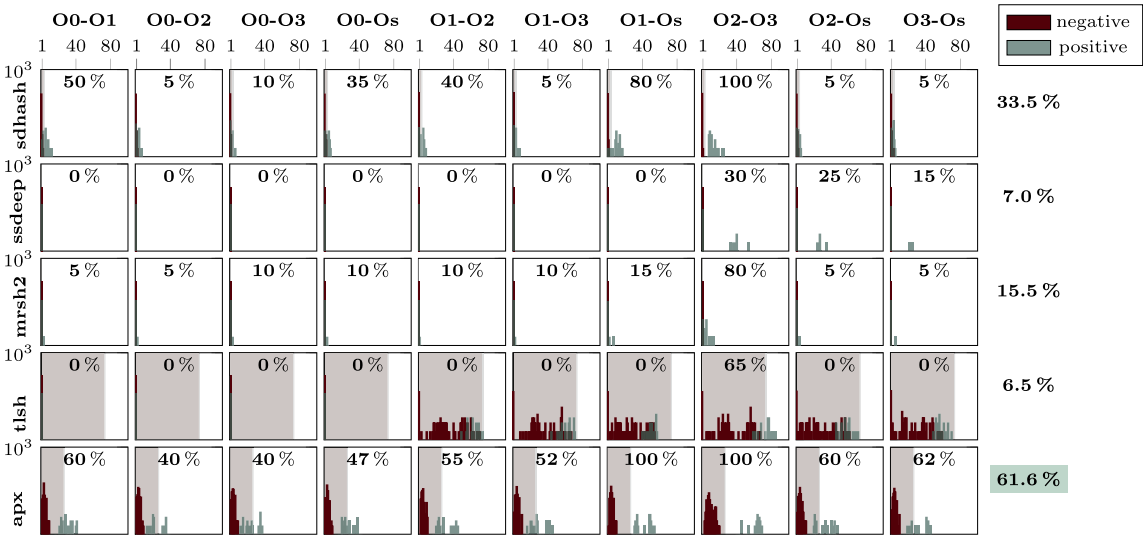

Fig. 11. Scenario II. Re-Compilation 1. Optimization flags: competitive evaluation of apx-bin. Plots visualize the similarity between different optimization levels and for different schemes. Comparisons with the same configuration (i.e., same flags) are not shown. The value of percentage denotes the rate of true positives with zero false positives. The coloured area describes the highest similarity score of a false match (y axis denotes the number of matches from 0 to 1000; x axis denotes the similarity score).

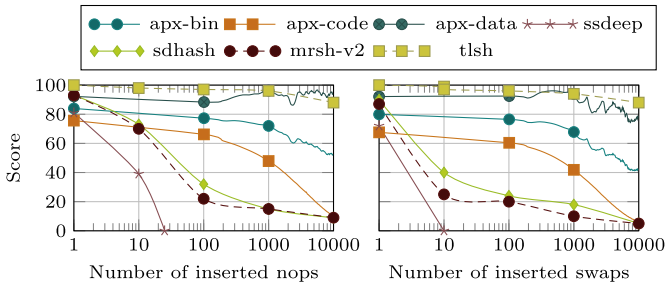

Fig. 13. Scenario III. Small Assembly Differences 1. Evaluation after the insertion of random NOPs and the swapping of instructions: apx-bin compared to results form Pagani et al. (2018).

Table 2
Scenario III. Minor Source Code Modifications 2. Overview of ranging similarities.

Change	ssdeep	mrsh-v2	tlsh	sdhash	apx
Operator	0–100	21–100	99–100	22–100	76–99
Condition	0–100	22–99	96–99	37–100	83–99
Constant	0–97	28–99	97–99	35–100	81–99

Table 3
Scenario III. Source Code Modifications on Malware 3. Modifications applied on the Mirai (M) and Grum (G) family. Overview of similarity scores.

Change	ssdeep		mrsh-v2		tlsh		sdhash		apx	
	M	G	M	G	M	G	M	G	M	G
C2 domain (r)	0	0	97	10	99	88	98	24	78	99
C2 domain (l)	0	0	44	13	94	84	72	22	76	86
Evasion	0	0	17	0	93	87	16	34	49	99
Functionality	0	0	9	0	88	84	22	7	34	79

results for all three cases. As mentioned by (Pagani et al., 2018), tlsh clearly is not affected by such small variances. In the case of modifying the source code of two malware samples, tlsh outperforms apx-bin in the case of the Mirai sample. However, in the case of Grum apx-bin shows competing results and outperforms tlsh in the case of Command and Control (C2) adaptations and evasion (see Table 3).

Conclusion

In this work we introduced and evaluated apx-bin, a new approximate matching derivative for the task of binary-related similarity matching. Contrary to recent approaches, we based our prototype on primitives of Context-Triggered Piecewise-Hashing. By the extension of mrsh-mem, an derivative of mrsh-v2, we could outperform several of the existing approaches in different scenarios. Our adaptations mainly rely on mapping a present sequences of bytes into an integerized representation, utilizing a trained prefix-tree. The integrated feature of discriminating code from data enabled us to propose a sufficient score model by weighting the specific rates. We reassessed previous research and proved that the discrimination of code- and data-related features empowers us to create more robust digests, a fact which is implicitly represented by the scores of some other schemes. Where tlsh and sdhash have advantages and disadvantages in different scenarios, apx-bin demonstrates more stable inferences across all considered evaluations (see Fig. 14).

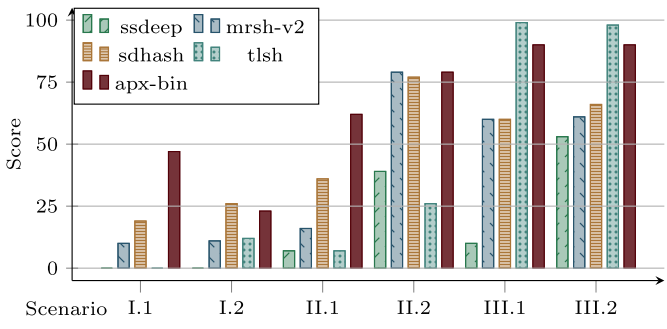

Fig. 14. Overview of capabilities of apx-bin compared to existing techniques in the different scenarios. Scenarios with multiple scores have been averaged.

Future work

Beside the extraction of the raw byte-chunks or mapped representatives, we could imagine the integration of n-gram based histograms as a third layer. The utilized approximate disassembler needs to be extended and the overall implementation should be formulated within a technical report. The current scheme should yield a compact digest, e.g., by the utilization of multiple Bloom filters.

The comprehensive set of scenarios proposed by Pagani et al. (2018) are a valuable contribution and should be further extended and formalized. An uniform evaluation platform with widespread scenarios within an adversarial environment is still required (Oliver et al., 2014). The utilized datasets should be additionally formalized as we had issues with very small or duplicate files within the set.

A more general discussion would be required to interpret the received values of similarity by different schemes, as the possible interpretation strongly differs and should match the needs of an investigator. It is discussable if the insertion of more than 1000 NOP instructions should not be represented by a utilized scheme. In other terms, we clearly need to distinguish between a *semantic* similarity and *bytewise* approximate matching.

Acknowledgment

This work was supported by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry of Higher Education, Research and the Arts (HMWK) within CRISP (www.crisp-da.de). We would like to thank Fabio Pagani for his useful comments and for providing the full evaluation set introduced in Pagani et al. (2018). Furthermore, we would like to thank the reviewers for their valuable feedback.

References

Azab, A., Layton, R., Alazab, M., Oliver, J., 2014. Mining malware to detect variants. In: Cybercrime and Trustworthy Computing Conference (CTC), 2014 Fifth. IEEE, pp. 44–53.
Bloom, B.H., 1970. Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13, 422–426.
Breitinger, F., Baier, H., 2012. Similarity preserving hashing: eligible properties and a new algorithm mrsh-v2. In: International Conference on Digital Forensics and Cyber Crime. Springer, pp. 167–182.
Breitinger, F., Guttman, B., McCarrin, M., Roussev, V., White, D., 2014. Approximate Matching: Definition and Terminology, vol. 800. NIST Special Publication, p. 168.
French, D., Casey, W., 2012. 2 fuzzy hashing techniques in applied malware analysis. In: Results of SEI Line-Funded Exploratory New Starts Projects, p. 2.
Kornblum, J., 2006. Identifying almost identical files using context triggered piecewise hashing. Digit. Invest. 3S, S91–S97.
Li, Y., Sundaramurthy, S.C., Bardas, A.G., Ou, X., Caragea, D., Hu, X., Jang, J., 2015. Experimental study of fuzzy hashing in malware clustering analysis. In: 8th Workshop on Cyber Security Experimentation and Test (Cset 15). USENIX Association, Washington, DC, p. 52.
Liebler, L., Baier, H., 2017. Approxis: a fast, robust, lightweight and approximate disassembler considered in the field of memory forensics. In: International Conference on Digital Forensics and Cyber Crime.

Liebler, L., Breitinger, F., 2018. mrsh-mem: approximate matching on raw memory dumps. In: International Conference on IT Security Incident Management and IT Forensics. IEEE, pp. 47–64.

Oliver, J., Cheng, C., Chen, Y., 2013. Tlsh–a locality sensitive hash. In: Cybercrime and Trustworthy Computing Workshop (CTC), 2013 Fourth. IEEE, pp. 7–13.

Oliver, J., Forman, S., Cheng, C., 2014. Using randomization to attack similarity digests. In: International Conference on Applications and Techniques in Information Security. Springer, pp. 199–210.

Pagani, F., Dell'Amico, M., Balzarotti, D., 2018. Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis. In: Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy. ACM, pp. 354–365.

Ren, L., 2015. A theoretic framework for evaluating similarity digesting tools. https://www.dfrws.org/file/127/download?token=5YOUdHpY_visited_on_2019-01-20.

Roussev, V., 2010. Data fingerprinting with similarity digests. In: IFIP International Conference on Digital Forensics. Springer, pp. 207–226.

Upchurch, J., Zhou, X., 2015. Variant: a malware similarity testing framework. In: Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on. IEEE, pp. 31–39.