Towards Carving-Based Post-Mortem Memory Forensics and the Applicability of Approximate Matching

Lorenz Liebler

Vollständiger Abdruck der an der Fakultät für Informatik der Universität der Bundeswehr München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Gutachter/Gutachterin:

- Prof. Dr. Harald Baier, Universität der Bundeswehr München
- Prof. Dr.-Ing. Felix Freiling, Universität Erlangen-Nürnberg

Die Dissertation wurde am 10.9.2021 bei der Universität der Bundeswehr München eingereicht und durch die Fakultät für Informatik am 2.12.2021 angenommen. Die mündliche Prüfung fand am 7.12.2021 statt.

Lorenz Liebler: Towards Carving-Based Post-Mortem Memory Forensics and the Applicability of Approximate Matching

SUPERVISORS:
Prof. Dr. Harald Baier, Universität der Bundeswehr München
Prof. Dr.-Ing. Felix Christoph Freiling, Universität Erlangen-Nürnberg

Declaration of Authorship

I, Lorenz Liebler, MSc., declare that this thesis entitled, “Towards Carving-Based Post-Mortem Memory Forensics and the Applicability of Approximate Matching” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ____

Date: ____

Abstract

Memory forensics is an important branch of digital forensics. Different concepts empower practitioners to perform deep analysis of potentially compromised systems by dissecting the acquired volatile memory of a target. The field mainly evolved in recent years and relies on the ambitious development of interfaces to extract and interpret structural information. In contrary, memory carving, also denoted as unstructured analysis, encompasses the extraction of artefacts or objects based on signatures or patterns. With the introduction of frameworks responsible for the complex structural interpretation of an acquired dump, researchers began to further expand the field, understandably shifting focus towards structured methodologies.

Even if structured analysis undoubtedly creates the foundation for deep insights into an acquired system, the overall concept bares some pitfalls and major implementation efforts. The multilayered and complex interpretation reveals much care and constant maintenance. This need has been further increased by shorter operating system release cycles. In addition, the analysis could suffer from various inconsistencies (memory smearing) of structural information, caused by the non-atomic acquisition of a running system. The degree of atomicity usually depends on the depicted method of acquisition, e.g., the acquisition via software tools, hardware appliances or virtualization features. Another eligible argument in favour of carving-based extraction is the potential compromise of structural information by an adversary, which seems to be a viable argument considering recent research in anti-memory-forensics. In a nutshell, mentioning just a small portion of a large variety of different obstacles, it should be desirable to back structured analysis by additional concepts of unstructured analysis and to introduce different concepts of data reduction similar to those in disk forensics. Therefore, this research investigates the transferability of Approximate Matching concepts to the field of memory forensics. Those functions are usually used to determine the similarity between two input files.

After giving a broad systematization of the field in terms of different criteria of research, we will discuss the transferability of existing schemes by the introduction of several contributions: First, we inspect the possibility of fast differentiating between code and data with the help of a new approach called Approximate Disassembling. In other terms, we introduce a concept for x86-64 code fragment carving. The approach will be the entry point for further discussions and extensions to existing Approximate Matching implementations. Second, we discuss the possible integration of our previously introduced dispatcher into an existing Approximate Matching technique. The required adaptations to the implementation and parametrization will be outlined. The approach will be discussed in the course of inspecting a raw memory image, i.e., for the task of identifying the running kernel version and different applications. Third, we introduce the possibility of extending an existing Approximate Matching technique with contextual extraction capabilities. In detail, techniques to carve the function start offset via common prologue sequence have been reassessed for our specific domain. Fourth, we inspect different and fairly new concepts of storing and handling extracted artefacts. The evaluation and assessment aspects of the different approaches is often denoted as Database Lookup Problem. In detail, we discuss the different implementations under aspects like common block filtration and deduplication. Last, we inspect the capabilities of our considered approaches in the course of binary matching. In detail, we adapt previously introduced techniques and discuss the performance in contrast to predominant Approximate Matching approaches.

Zusammenfassung

Das Feld der Speicherforensik ist ein wichtiger Zweig der digitalen Forensik. Verschiedene Konzepte ermöglichen es Praktikern, detaillierte Analysen von potenziell kompromittierten Systemen durchzuführen, indem sie den flüchtigen Speicher eines Ziels auswerten. Das Feld hat sich in den letzten Jahren stark weiterentwickelt und stützt sich auf die ambitionierte Entwicklung von Schnittstellen zur Extraktion und Interpretation von Strukturinformationen. Im Gegensatz dazu umfasst Memory Carving, auch als unstrukturierte Analyse bezeichnet, die Extraktion von Artefakten oder Objekten basierend auf Signaturen oder Mustern. Mit der Einführung von Frameworks, die für die komplexe strukturelle Interpretation eines erzeugten Dumps verantwortlich sind, begannen Forscher, das Feld weiter auszubauen, wobei sich der Fokus verständlicherweise auf strukturierte Methoden verlagerte.

Auch wenn die strukturierte Analyse zweifelsohne die Grundlage für tiefe Einblicke in ein erfasstes System schafft, birgt das Gesamtkonzept einige Fallstricke und großen Implementierungsaufwand. Darüber hinaus kann die Analyse unter verschiedenen Inkonsistenzen (Memory Smearing) der Strukturinformationen leiden, die beispielsweise durch die nicht-atomare Erfassung eines laufenden Systems verursacht werden. Der Grad der Atomarität hängt in der Regel von der dargestellten Erfassungsmethode ab, zum Beispiel der Erfassung per Software-Tools, Hardware-Appliances oder Virtualisierungsfunktionen. Ein weiteres zulässiges Argument für die Carving-basierte Extraktion ist die potenzielle Kompromittierung von Strukturinformationen durch einen Angreifer, was angesichts der jüngsten Forschungen im Bereich der Anti-Memory-Forensik ein tragfähiges Argument zu sein scheint. Zusammenfassend lässt sich sagen, dass es, um nur einen kleinen Ausschnitt aus einer Vielzahl verschiedener Hindernisse zu nennen, wünschenswert wäre, die strukturierte Analyse durch zusätzliche Konzepte der unstrukturierten Analyse zu unterstützen und verschiedene Konzepte der Datenreduktion einzuführen. Daher wird in dieser Arbeit die Übertragbarkeit von Approximate Matching Funktionen auf den Bereich der Speicherforensik untersucht. Diese Funktionen werden normalerweise verwendet, um die Ähnlichkeit zwischen zwei Eingabedateien zu bestimmen.

Zunächst untersuchen wir die Möglichkeit der schnellen Unterscheidung zwischen Code und Daten mit Hilfe eines neuen Ansatzes namens Approximate Disassembling. Mit anderen Worten, wir stellen ein Konzept für das x86-64 Codefragment Carving vor. Der Ansatz wird der Einstiegspunkt für weitere Diskussionen und Erweiterungen bestehender Approximate Matching-Implementierungen sein. Zweitens diskutieren wir die mögliche Integration unseres vorgestellten Dispatchers in ein bestehendes Approximate Matching-Verfahren. Die erforderlichen Anpassungen an der Implementierung und Parametrisierung werden skizziert. Der Ansatz wird im Zuge der Inspektion eines Rohspeicherabbilds diskutiert, d. h. für die Aufgabe, die laufende Kernelversion und verschiedene Anwendungen zu identifizieren. Drittens stellen wir die Möglichkeit vor, ein bestehendes Approximate Matching-Verfahren um kontextuelle Extraktionsmöglichkeiten zu erweitern. Im Detail wurden Techniken zum Carven des Funktionsstart-Offsets per Prologsequenzen für unsere spezifische Domäne neu bewertet. Viertens untersuchen wir verschiedene und junge Konzepte zur Speicherung und Handhabung von extrahierten Artefakten. Die Auswertungs- und Bewertungsaspekte der verschiedenen Ansätze werden oft als Database Lookup Problem bezeichnet. Im Detail diskutieren wir die verschiedenen Implementierungen unter Aspekten wie gemeinsamer Blockfilterung und Deduplizierung. Zuletzt untersuchen wir die Fähigkeiten der von uns betrachteten Ansätze im Zuge des Abgleichs von Programmdateien. Im Detail adaptieren wir zuvor eingeführte Techniken und diskutieren die Performanz im Gegensatz zu vorherrschenden Approximate Matching Ansätzen.

Acknowledgements

This work would not have been possible without the help of many others. First of all I would like to thank my supervisor *Prof. Dr. Harald Baier* for his scientific guidance and support during the last years. Giving me the opportunity to join his digital forensics research group was the starting point of this exciting journey. My special thanks also go to *Prof. Dr. Felix Freiling* for providing valuable feedback and for being a part of my doctoral committee. I would also like to thank *Prof. Dr. Frank Breitinger* for several successful projects and for hosting my stay in New Haven.

I thank my colleagues of da/sec and several members of the University of Applied Sciences Darmstadt. In particular I want to thank *Daniel*, who was always very supportive and silently took care of many critical administration tasks in the background. I also would like to thank my research assistants and interns, who supported me during my graduation, but also played an important role in fulfilling my project-related duties: *Marcel*, *Patrick*, *Sascha*, *Daniel S.*, *Hendrik*, and *Frank*.

There are some fellow students and colleagues who became very good friends over time. Thanks *Julien* and *Johannes* for all the discussions, nerd snipes, and supportive words in doubtful times. Your sincere interest in my work and personal well-being has been a major driving force for me. Additionally, I would like to thank my colleagues *Uli* and *Thomas* for taking part in helpful discussions and having productive times in the lab. I will certainly miss your creativity in terms of culinary and healthy lunch breaks.

It was a long way until I finally reached this goal. This would have never been possible without the constant support of my family. Most importantly I have to thank my parents *Eva* and *Otto*, and my older brother *Ralf*: You supported me in all phases of my life and created the environment for this success. I thank my grandparents *Maria* and *Ewald* for their endless support and love. I also would like to thank *Sascha*, who majorly influenced my decision to study computer sciences.

Kate - I am deeply grateful that I met you. Without you, this work would undoubtedly not exist. Thank you for your understanding, but especially for your advice, balance and love.

Contents

List of Figures

List of Tables

Acronyms

ACC Accuracy.

AM Approximate Matching.

ASLR Address Space Layout Randomization.

BF Bloom Filter.

CDP Code Detection Problem.

CF Cuckoo Filter.

CHDB Chunk Hash Database.

CHF Chunk Hash Function.

CHV Chunk Hash Values.

CTPH Context-Triggered Piecewise-Hash.

DFIR Digital Forensics and Incident Response.

DLP Database Lookup Problem.

FDP Function Detection Problem.

FP False Positive.

HBFT Hierarchical Bloom Filter Tree.

LSTM Long Short-Term Memory.

MFO Most Frequent Occurred.

MRSH Multi-Resolution Similarity Hash.

NIST National Institute of Standards and Technology.

OS Operating System.

PFN Page Frame Number.

PRE Precision.

PRF Pseudo Random Function.

REC Recall.

RNN Recurrent Neural Network.

ROC Receiver Operating Characteristic.

SDHASH Similarity Digest Hashing.

SPHF Similarity Preserving Hash Function.

TLSH Trend Micro Locality Sensitive Hash.

TPR True Positive Rate.

WPT Weighted Prefix Tree.

Publications

This thesis is based on the following peer-reviewed publications (ordered by year):

2017 Lorenz Liebler and Harald Baier. "Approxis: a fast, robust, lightweight and approximate disassembler considered in the field of memory forensics", International Conference on Digital Forensics and Cyber Crime. Springer, Cham, 2017.

2018 Lorenz Liebler and Harald Baier. "On Efficiency and Effectiveness of Linear Function Detection Approaches for Memory Carving", International Conference on Digital Forensics and Cyber Crime. Springer, Cham, 2018.
(Best Paper Award)

2018 Lorenz Liebler and Frank Breitinger. "mrsh-mem: Approximate matching on raw memory dumps", 2018 11th International Conference on IT Security Incident Management & IT Forensics (IMF). IEEE, 2018.

2019 Lorenz Liebler and Harald Baier, "Towards Exact and Inexact Approximate Matching of Executable Binaries", in Proceedings of the 6th Digital Forensic Research Workshop EU (DFRWS EU 2019), Oslo (Norway), April 2019.
(Best PhD Student Paper Award)

2019 Lorenz Liebler, Patrick Schmitt, Harald Baier and Frank Breitinger, "On Efficiency of Artifact Lookup Strategies in Digital Forensics", in Proceedings of the 6th Digital Forensic Research Workshop EU (DFRWS EU 2019), Oslo (Norway), April 2019.

1 INTRODUCTION

1.1 MOTIVATION

In the course of Digital Forensics and Incident Response (DFIR) the acquisition and analysis of volatile memory have gained significant attention by practitioners and researchers in recent years. The examination of the previously acquired memory of a running system gives deep insights into the state of a machine and empowers the investigation of recently performed actions and processes. The academic community mainly focused on the field of *structured memory forensics*. As the name suggests, this branch of memory forensics is based on the extensive extraction and interpretation of Operating System (OS) related memory structures. It should be clear that inferences gained from the analysis of the core implementations of the target should be very precise and more meaningful than any technique of *unstructured memory forensics*. However, the context of application, the complexity, and the variety of different interpretation levels introduce several long-lasting challenges.

The detection, the processing, and the analysis of OS and application related structures is based on the costly process of *maintenance* and updating the operability of tools and frameworks, where the overall problem is often denoted as *semantic gap* (Dolan-Gavitt et al., 2011). Thus, maintenance and community-driven expertise is a fundamental building block for the long-term utility of those complex, based-on-heuristics, and multi-layered frameworks. Often, the stability and applicability is influenced by eligible updates or changes to the core functionalities of the operating system itself. In addition, the interpretation could be hindered by erroneous images due to acquisition errors, also known as *memory smearing* (Pagani, Fedorov, and Balzarotti, 2019). This leads to inconsistencies and errors within the chain of interpretation by the memory forensics frameworks. In advances to foundational work done to lift an acquired dump into an interpretational state, examiners and practitioners are often responsible for creating the overlaying space of domain-knowledge backed plugins. Most often, those plugins introduce additional components and extensions to the underlying frameworks, which are mainly responsible for the process of heavy lifting until final interpretation.

Besides inferences not possible with classical disk- or file system forensics, obfuscated malicious software (malware) and evasive actions could barely hide their traces within the states of a target system reflected by its internal state in memory. Thus, memory forensics plays an important role in the current digital forensics community and research. However, adversary forces realize the capabilities of live- or post-mortem memory forensics. Recent research underlined the feasibility of *anti-memory-forensics* techniques, which could slow down, complicate, prevent, or attack the overall forensical soundness of an analysis (Block and Dewald, 2019).

With the increasing size of main memory, White et al. (White, Schatz, and Foo, 2013) formulate requirements for 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 performant instrument to differentiate between known and unknown data fragments (Breitinger et al., 2014). However, White et al. claim that Approximate Matching algorithms are not suitable in the course of memory forensics, as code in memory always differs from its version on disk.

1.2 RESEARCH QUESTIONS AND CONTRIBUTIONS

Considering the introductory motivation, this research further inspects the applicability of Approximate Matching in the field of memory forensics. Summarized and simplified, in this work we argue if concepts and techniques of existing schemes may be transferred from disk or network forensics to the field of memory forensics. Thus, the overall and general research could be formulated by a single and generic research question.

RQ: *Is it possible to strengthen main memory analysis by successfully transferring concepts of Approximate Matching to the domain of memory analysis?*

We further split the overall research question into smaller partial questions and shortly introduce our own contributions with respect to each question.

SYSTEMATIZATION OF KNOWLEDGE. As the field mainly evolved within the last decade, we need to outline recent research efforts and differentiate academic publications in terms of research criteria and research ambitions. A broad systematization of knowledge should further delimit our contributions from existing work and give a better impression of the field's status quo.

RQ1: *Which memory analysis methods have been proposed and what are their main properties in terms of type, scope, and context of application?*

A committed community of forensics practitioners and scientific researches try to keep pace with the fast-moving field of memory forensics, driven by the complexity, increasing number and shortened update cycles of target systems. Within the last 15 years the field mainly evolved and proposed approaches of acquisition and analysis require constant care to remain applicable over time. A first contribution of this work is the presentation of a systematization of knowledge by considering more than 140 publications related to the field of memory forensics.

APPROXIMATE DISASSEMBLING. We differentiate between multiple stages of research to succeed. First, a technical component is needed, which extracts and labels portions of code or data out of vast amounts of unknown data. In terms of carving, differentiating extracted fragments should be possible without any structural properties like the binary source format or the requirement of recreating a process-context with the help of structured analysis. Differentiating between code and data should empower to further transforming or normalizing extracted fragments.

RQ2: *What are applicable and interfaceable approaches for differentiating code and data applied on a considerably large and unstructured bulk of data?*

Detecting known and unknown code structures in large sets of data is a challenging task. An example is 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 constantly 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. We introduce an approximate disassembler named approxis. 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. To summarize, approxis is our first step to interface Approximate Matching with the field of memory forensics.

MEMORY CARVING. The integration of a preliminary processing step, as well as the considered context of application, expectedly requires the adaptation of existing Approximate Matching schemes. To be more specific, the transfer or application of existing schemes on acquired memory dumps requires the adaptation of those schemes in terms of parametrization and feature extraction.

RQ3: *Under which premise could we interface or extend Approximate Matching as memory carving technique? Which properties need to be respected and should be considered in further implementations and parametrizations?*

This research presents the fusion of two subdomains of digital forensics: (1) raw memory analysis and (2) Approximate Matching. Specifically, we describe a prototype implementation named mrsh-mem that allows comparing hard drive images as well as memory dumps and therefore can answer the question if a particular program (installed on a hard drive) is currently running / loaded in memory. To answer this question, we only require both dumps or access to a public repository which provides the binaries to be tested. For our prototype, we modified an existing Approximate Matching algorithm named mrsh-net and combined it with approxis. Literature claims that Approximate Matching techniques are hardly applicable to the field of memory forensics (White, Schatz, and Foo, 2013). Especially legitimate changes to executables in memory caused by the loader prevent the application of current bytewise Approximate Matching techniques. Our approach lowers the impact of modified code in memory and shows a good computational performance.

DATABASE LOOKUP PROBLEM. The phase of feature or data chunk extraction inevitably leads to the question of storing and handling those extracted artefacts. Several approaches of artefact storages have been introduced in recent years in the field of DFIR. Thus, an appropriate candidate has to be selected for our specific use case, considering different aspects of our application.

RQ4: *What are suitable candidates for storing extracted artefacts, and how is it possible to solve important aspects like common-block-handling?*

One concept to deal with data overload is data reduction, which essentially means separating the wheat from the chaff, e.g., to filter in forensically relevant data. Prominent techniques in the context of data reduction are hash-based solutions. Data reduction is achieved because hash values (of possibly large data input) are much smaller than the original input. Today's approaches of storing hash-based data fragments range from large scale multithreaded databases to simple Bloom filter representations. One main focus was put on the field of Approximate Matching, where sorting is a problem due to the fuzzy nature of the approximate hashes. A crucial step during digital forensic analysis is to achieve fast query times during lookup (e.g., against a blacklist), especially in the scope of small or ordinary resource availability. However, a comparison within different database and lookup approaches is considerably hard, as most techniques partially differ in considered use-case and integrated features, respectively. We discuss, reassess and extend three widespread lookup strategies suitable for storing hash-based fragments: Hashdatabase for hash-based carving, hierarchical Bloom filter trees, and flat hash maps (Garfinkel and McCarrin, 2015; Lillis, Breitinger, and Scanlon, 2017). We outline the capabilities of the different approaches, integrate new extensions, discuss possible features and perform a detailed evaluation with a special focus on runtime efficiency. Our results reveal major advantages for fhmap in case of runtime performance and applicability. Hbft showed a comparable runtime efficiency in case of lookups, but suffers from pitfalls with respect to extensibility and maintenance. Finally, hashdb performs worst in case of a single core environment in all evaluation scenarios. However, hashdb is the only candidate which offers full parallelization, transactional features, and a Single-level storage.

CODE CARVING. As it should be clear, hashing fragments of code-related structures is more or less strongly-related to code-similarity analysis or code-reuse detection. Thus, another important aspect is the adaptation of binary-analysis specific contextual extraction approaches, i.e., for the task of contextual feature or chunk extraction. We need to inspect the transferability of contextual feature extraction techniques and their integrability in terms of Context-Triggered Piecewise-Hash (CTPH) techniques.

RQ5: *Is it possible to improve or replace existing contextual trigger functions of common (CTPH-based) Approximate Matching with better suited techniques?*

In the field of memory carving, the context-unaware detection of function boundaries could lead to meaningful insights. For instance, in the field of binary analysis, those structures yield further inference, e.g., identifying binaries known to be bad. However, recent publications discuss different strategies for the problem of function boundary detection and consider it to be a difficult problem (Andriesse, Slowinska, and Bos, 2017). One of the reasons is that the detection process depends on a quantity of parameters including the used architecture, programming language and compiler parameters. Initially, a typical memory carving approach transfers the paradigm of signature-based detection techniques from the mass storage analysis to memory analysis. To automate and generalise the signature matching, signature-based recognition approaches have been extended by machine learning algorithms. We reassess the application of recently discussed machine learning based function detection approaches in our context. We analyse current approaches in the context of memory carving with respect to both their efficiency and their effectiveness. Additionally, we discuss the capabilities of different function start identification techniques by reducing the features to vectorised mnemonics.

BINARY MATCHING. As many Approximate Matching schemes are often discussed as appropriate tools for matching similar malicious or benign binaries, the evaluation of our proposed adaptations should be additionally evaluated in this strongly related field.

RQ6: *Can we transfer the newly introduced concepts to the field of binary matching?*

Recent research showed major weaknesses of predominant fuzzy hashing techniques in the case of measuring the similarity of executables (Pagani, Dell'Amico, and Balzarotti, 2018). To summarize, 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, Cheng, and Chen, 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. We present a new scheme called apx-bin, considering recent research results mainly 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, our scheme is more robust and shows stable results across all considered scenarios.

1.3 THESIS OUTLINE

The mentioned research topics of this thesis are structured as follows:

- *Give an overview of relevant fields of research:*
  In **Chapter 2** we provide a generous introduction into two important aspects of our own research: Approximate Matching and Binary Analysis. Each section provides a compact overview, a discussion of the current state of the art, relevant publications, and how they further influenced our own contributions. In each section, we will introduce the relevant publications and their cross dependencies within the scientific and, especially, the DFIR community. In the course of Approximate Matching, we will discuss the preliminary use cases, the possible distinctness of the differing approaches, as well as the applicability in the field of binary matching. In addition, we will introduce a well-known problem of the community, the Database Lookup Problem (DLP). Afterwards, we will inspect two well-known problems taken over from the field of binary analysis: the Function Detection Problem (FDP), as well as the Code Detection Problem (CDP). In addition, the concept for generating a correct and sound ground truth dataset is discussed.

- *Give an overview of the field of memory forensics:*
  In **Chapter 3** we will give a systematic overview of the field of memory forensics which has been advanced by a variety of publications and approaches. We focus on concepts of unstructured analysis and previous work discussing the application of Approximate Matching in the field of memory analysis.

- *Tackling the problem of code/data discrimination:*
  In **Chapter 4** we will introduce a concept for discriminating code and data by

considering large amounts of acquired and unknown data, named approxis. The underlying concept of approximate disassembling or dispatching a byte stream will be evaluated in three different scenarios: differentiating code and data, differentiating the architecture, and differentiating the compiler (and partially the version of the compiler).

- *Introduce Approximate Matching to the field of memory carving:*
  In **Chapter 5** we will present and discuss the possible integration of approxis into an existing Approximate Matching scheme. The overall goal is to extract loaded executables within memory; e.g., for determining specific executed applications or kernel versions.
- *Discuss replacements for pseudo-random based trigger functions (PRF):*
  In **Chapter 6** the possible adaptation of different function detection approaches is inspected. In detail the applicability of Recurrent Neural Network (RNN) or Weighted Prefix Tree (WPT) is discussed in the course of context-sensitive chunk extraction within existing CTPH approaches.
- *Reassess different approaches to store and maintain extracted artefacts:*
  In **Chapter 7** we will revisit the Database Lookup Problem and reassess three different approaches in terms of utility and performance.
- *Discuss the introduced Dispatcher in the context of binary matching:*
  In **Chapter 8** we inspect the capabilities of an improved feature extraction phase via approximate disassembling in the course of binary matching. We therefore evaluate our approach by the adoption of three different scenarios. In addition, we create an own adversary set of evaluation binaries and benchmark our own approach against four predominant fuzzy hashing techniques.
- In **Chapter 9** we conclude the thesis.

2

BACKGROUND AND RELATED WORK

The following chapter discusses the basic and related background of this thesis, as well as previous research efforts in overlapping disciplines of research.

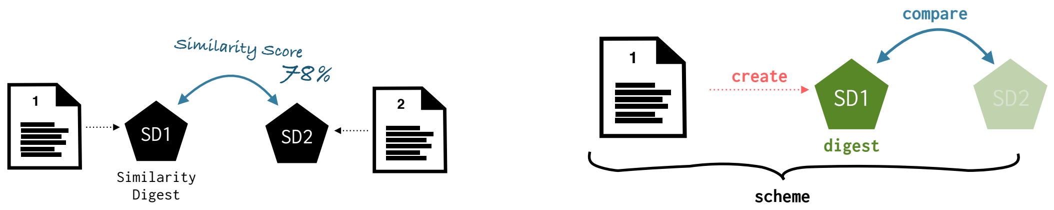
- In Section 2.1, we give a broad overview of Approximate Matching (AM) concepts, also known as fuzzy hashing. We give a short overview of relevant publications, specific implementations, and selected approaches previously discussed in the field of malware or binary matching. In addition, we introduce another important aspect of AM: The Database Lookup Problem.
- In Section 2.2, we introduce problems strongly related to the field of binary analysis: The problem of correctly disassembling the x86-64 instruction set, differentiating code from data, and the detection of function boundaries of stripped binaries.

2.1 APPROXIMATE MATCHING

The following Subsection 2.1.1 gives a short introduction into Approximate Matching and some of its major manifestations. Besides the basic idea and considered fields of application, the diverse implementations could differ heavily by their underlying concepts and implementations. To understand and better differentiate the concepts, we have to briefly introduce the considered approaches and their differences. Therefore, we first give a short summary of the concepts, research paths, and well-known implementations. Subsequently, we shortly introduce in Subsection 2.1.2 a major field of application of today's Approximate Matching concepts: matching, identifying, or clustering (malicious) binaries. Finally, we inspect in Subsection 2.1.3 one major issue of Approximate Matching suffers in general, the database lookup problem and the overall handling of block hashes.

2.1.1 *Approaches and Taxonomy*

Approximate Matching (a.k.a. Fuzzy Hashing) is a so-called Similarity Preserving Hash Function (SPHF). In contrast to cryptographic hash functions, those functions determine the similarity of two files. Introduced more than a decade ago to deal with spam, Approximate Matching is now considered to solve different (digital) forensic challenges. Approximate Matching is a rather new area of digital forensics and can



(A) Simplified concept of Approximate Matching algorithms.

(B) An Approximate Matching scheme normally consists of a creation and comparison function.

FIGURE 2.1: Approximate Matching: Simplified concept.

be seen as the counterpart to traditional (cryptographic) hash functions, i.e., Approximate Matching algorithms return similar fingerprints for similar inputs. In the following paragraphs we summarize the most relevant approaches and aspects for our research. 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, similarity preserving hash functions have been developed. As can be seen in Figure 2.1, schemes create a similarity digest of an input stream and can be used to generate a similarity score between two files. Most often a scheme consists of two separated functions: a function to create the digest and a function to compare two digests. The National Institute of Standards and Technology (NIST) defines four possible use cases that Approximate Matching techniques could deal with: similarity detection (1), cross-sharing detection (2), embedded object detection (3), and fragmentation detection (4) (Breitinger et al., 2014).

A SHORT HISTORY OF APPROXIMATE MATCHING. In recent years plenty of different approaches have discussed different concepts of providing a similarity preserving hash function in different domains of DFIR. Introducing all of the different concepts is out of the scope of this work. Figure 2.2 gives a simplified overview of relevant publications and approaches with a focus on binary matching and malware detection. A more comprehensive overview and a technical taxonomy is provided by Harichandran et al. (Harichandran, Breitinger, and Baggili, 2016). The coloured nodes describe publications proposing new or adapted concepts (light/-dark red nodes), publications which focus on an overall assessment or summary of existing approaches (green nodes) and publications which discuss Approximate Matching in the context of malware or binary matching (yellow nodes).

As can be seen, research on Approximate Matching already began in 2002 with the creation of the spamsum approach (Tridgell, 2002) to handle the arising amount of spam. The original spamsum algorithm was an important building block for further research and future provided concepts. Even today, ssdeep proposed in 2006 plays an important role in academia, the IT industry, and IT security in general (Kornblum, 2006). About the same time, Roussev et al. proposed two of their concepts md5bloom (Roussev et al., 2006) and mrsh (Roussev, Richard III, and Marziale, 2007). The different approaches and a previous study about the extraction of common features in binary data (Roussev, 2009) finally lead to a new approach called Similarity Digest Hashing (SDHASH) (Roussev, 2010) which is mainly based on the extraction of statistically improbable features.

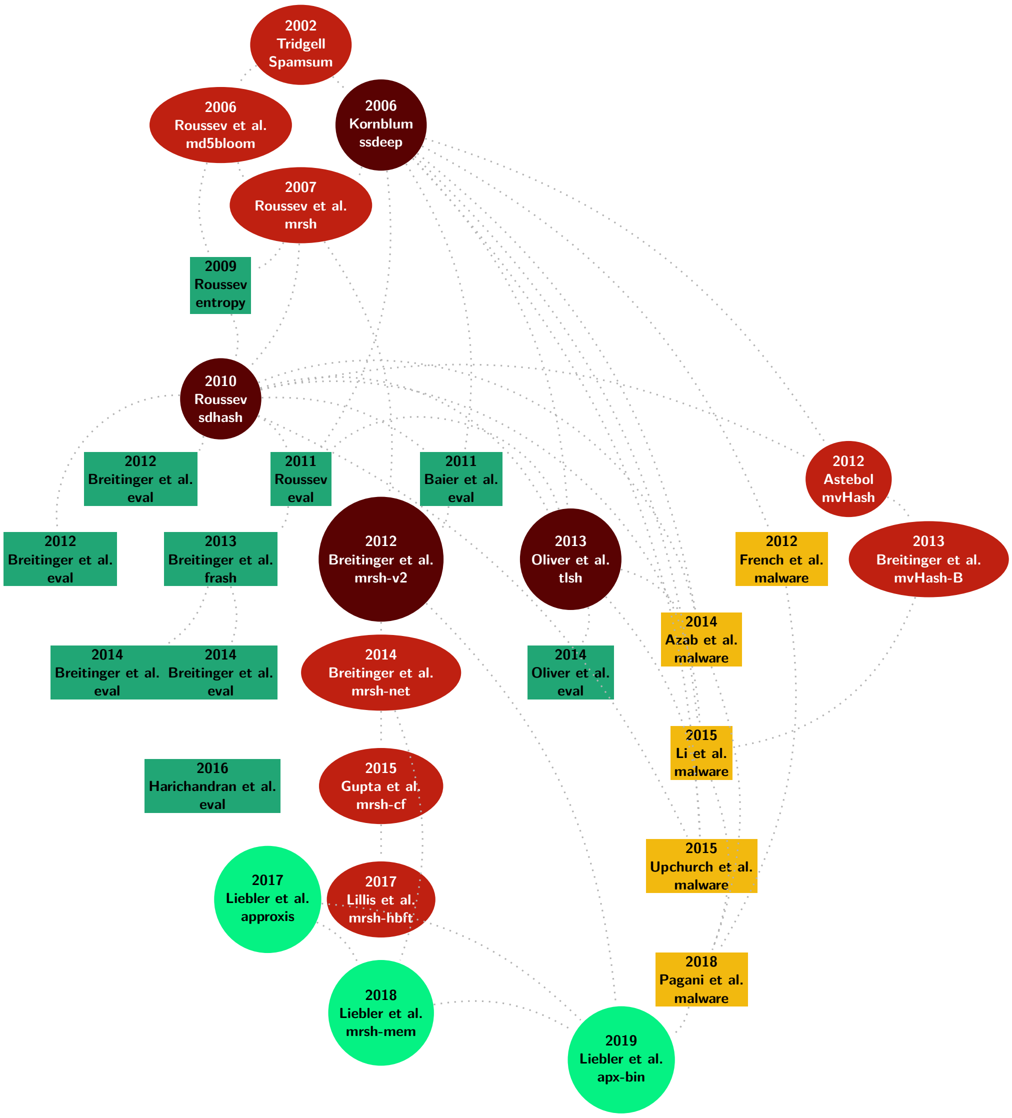

2002 Tridgell Spamsum
2006 Roussev et al. md5bloom
2006 Kornblum ssdeep
2007 Roussev et al. mrsh
2009 Roussev entropy
2010 Roussev sdhash
2012 Breitinger et al. eval
2011 Roussev eval
2011 Baier et al. eval
2012 Astebol mvHash
2012 Breitinger et al. eval
2013 Breitinger et al. frash
2012 Breitinger et al. mrsh-v2
2013 Oliver et al. tlsh
2012 French et al. malware
2013 Breitinger et al. mvHash-B
2014 Azab et al. malware
2014 Breitinger et al. eval
2014 Breitinger et al. eval
2014 Breitinger et al. mrsh-net
2014 Oliver et al. eval
2015 Li et al. malware
2016 Harichandran et al. eval
2015 Gupta et al. mrsh-cf
2015 Upchurch et al. malware
2017 Liebler et al. approxis
2017 Lillis et al. mrsh-hbft
2018 Pagani et al. malware
2018 Liebler et al. mrsh-mem
2019 Liebler et al. apx-bin

FIGURE 2.2: Compact overview of the different research and proposed Approximate Matching techniques with a focus on binary and malware matching.

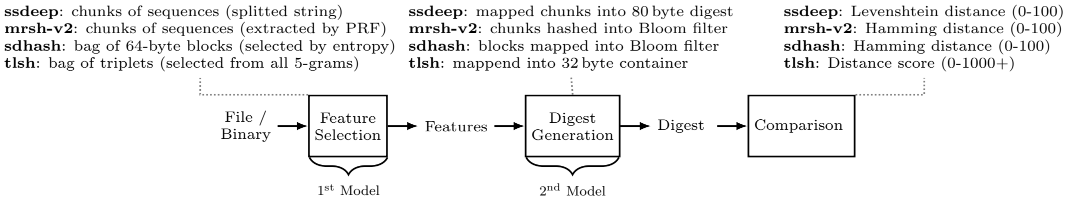

ssdeep: chunks of sequences (splitted string)
mrsh-v2: chunks of sequences (extracted by PRF)
sdhash: bag of 64-byte blocks (selected by entropy)
tlsh: bag of triplets (selected from all 5-grams)
ssdeep: mapped chunks into 80 byte digest
mrsh-v2: chunks hashed into Bloom filter
sdhash: blocks mapped into Bloom filter
tlsh: mappend into 32 byte container
ssdeep: Levenshtein distance (0-100)
mrsh-v2: Hamming distance (0-100)
sdhash: Hamming distance (0-100)
tlsh: Distance score (0-1000+)
File / Binary
Feature Selection
Features
Digest Generation
Digest
Comparison
1st Model
2nd Model

FIGURE 2.3: Approximate Matching approaches differed by steps of processing (similar to Ren, 2015).

In the following years, several publications proposed concepts of evaluation, attacks and improvements of existing approaches (Roussev, 2011; Breitinger, Baier, and Beckingham, 2012; Breitinger and Baier, 2012a; Baier and Breitinger, 2011; Breitinger, Stivaktakis, and Roussev, 2014; Breitinger and Roussev, 2014; Breitinger, Stivaktakis, and Baier, 2013). Most of them led to new or improved concepts, for example mrsh-v2, a derivative of the original Multi-Resolution Similarity Hash (MRSH) approach (Breitinger and Baier, 2012b).

We will give further insights and an detailed explanation of mrsh-v2 and its underlying concept throughout this section, as we used the present implementation for our own extensions and implementations. Besides the original mrsh-v2 implementation, additional modifications have been proposed. For example the adaptation in the course of detecting files via network analysis (Breitinger and Baggili, 2014) and handling the database lookup problem via the integration of Cuckoo-Filters (Gupta and Breitinger, 2015) or Hierarchical Bloom Filter Tree (HBFT) (Lillis, Breitinger, and Scanlon, 2017). In addition to general fuzzy hashing concepts like ssdeep and sdhash, Oliver, Cheng, and Chen, 2013 of Trend-Micro proposed a locality sensitive hashing scheme called Trend Micro Locality Sensitive Hash (TLSH) with an additional focus robustness in the field of malware analysis. Additional work underlined the robustness of tlsh considering advanced attacks like randomization (Oliver, Forman, and Cheng, 2014).

A future research path mainly focused on the applicability of fuzzy hashing for the task of identifying or matching malicious software (French and Casey, 2012; Azab et al., 2014; Li et al., 2015; Upchurch and Zhou, 2015). The contradictory and conflicting conclusions of previous research lead Pagani et al. (Pagani, Dell'Amico, and Balzarotti, 2018) to a detailed inspection of the different schemes in the context of matching executable binaries via Approximate Matching. The authors proposed different scenarios to evaluate and assess the four predominant schemes: ssdeep, sdhash, mrsh-v2 and tlsh. We will further introduce the discussion and conflicting findings of the mentioned publications in Subsection 2.1.2.

DIFFERING SCHEMES. The schemes for creating similarity preserving digests 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*. In the following we will briefly discuss the main concepts of the different schemes, similarly to its visualization shown in Figure 2.3:

- In the case of ssdeep (Kornblum, 2006) and mrsh-v2 (Breitinger and Baier, 2012b) 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 filters (mrsh-v2). In the case of ssdeep, the Levenshtein distance is used to compare two digests with each other. In the case of mrsh-v2 the Hamming distance is used to compare two or multiple Bloom filters.

- In contrast, 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, Cheng, and Chen (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 helps 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 can be seen in Figure 2.3. 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.

MRSH-V2 AND CTPH. From a high level perspective, Approximate Matching algorithms work as follow. First, the algorithm identifies features where a feature is usually a substring of the complete input (e.g., chunks of a particular length). These chunks are then mapped via (cryptographic) hash functions. Lastly, these shorter strings are then used to build a fingerprint / similarity digest.

For instance, let us have a closer look at algorithms of the mrsh family which form the basis for this work. These algorithms (e.g., mrsh-v2 Breitinger and Baier, 2012b) consider only the underlying byte sequence of a given input (no interpretation of the byte sequence). The given sequence is divided into chunks of size b (common values are $64 \leq b \leq 320$ bytes) . To do so, the algorithm uses a sliding window that rolls through the sequence byte-by-byte and considers 7 consecutive bytes at a time. This window is then hashed using a Pseudo Random Function (PRF) which returns a value between 0 and $b-1$. If $b == 0$, the end of a chunk is identified. As a consequence, if PRF behaves pseudo random, each chunk has approximately the size of b bytes. Once the end of a chunk is identified, a Chunk Hash Function (CHF) is used to compress the sequence (common CHFs are MD5, SHA or FNV-1a). Lastly, all chunk hashes are translated into a final fingerprint where different algorithms use different concepts. In the case of mrsh-net, a single large Bloom filter is used which is explained in the following paragraph (Breitinger and Baggili, 2014).

A Bloom Filter is a space-efficient, probabilistic data structure invented by Burton Howard Bloom in 1970 that consists of an array of m bits all set to zero (Bloom, 1970). In order to insert an element $s \in S$ into a Bloom filter, s is hashed using a hash function that returns values $|h(s)| \geq k \cdot log_2(m)$ bits[1]. Then, the numeric offset represented by the first $log_2(m)$ bits is used to set the corresponding bit in the Bloom filter; the second $log_2(m)$ bits are used to set the corresponding bit in the Bloom filter; this is repeated k times. For instance, assuming Bloom filter size $m = 64 = 2^6$ and $k = 2$, $h(s)$ should return a hash value of at least $(2 \cdot log_2(64) =)$ 12 bits, e.g., 011011 101101. Given that $011011_{bin} = 27_{dec}$ and $101101_{bin} = 45_{dec}$, bits 27 and 45 of the Bloom filter are set to one. To verify whether an element s' is in a given Bloom filter, it is hashed with the same hash function h. If all corresponding bits in the Bloom filter are set to one, the element was inserted into the Bloom filter with a certain probability (there is a chance for a false positive). If one of the bits is zero, the element was never inserted into the Bloom filter (there are not false negatives). Specifically, the false positive rate of a Bloom filter is influenced by three parameters: the size of the filter m, the amount of elements which are inserted into the filter n and the number of set bits per element k. The probability for a false positive can then be estimated with the formulas illustrated in Equation 2.1,

$$\begin{aligned} P_{FP} &= \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k \\ &= (1-p)^k, \text{with} \\ p &= 1 - \left(1 - \frac{1}{m}\right)^{kn}, \end{aligned} \tag{2.1}$$

where p is the probability of a bit being 0, after all n elements have been inserted (Breitinger, 2014). In order to create the *final fingerprint*, the first $k \cdot log_2(m)$ bits of the chunk hashes are utilized to set the corresponding bits in the Bloom filter. In other words, for each chunk k bits are set in the Bloom filter. A summary of the parameters is provided in Table 2.1.

Parameter	**Description**
b	Denotes the approximated chunk size
m	Denotes the Bloom filter size in bits
n	Number of elements inserted into a Bloom filter
k	Number of used sub-hashes; each sub-hash defines a bit in the corresponding Bloom filter

TABLE 2.1: Parameters of mrsh-net and their description.

The created fingerprints can be used to estimate the *similarity score* between two given files. Different Approximate Matching approaches create different fingerprints and thus utilize different techniques for similarity calculation. In the course of mrsh derivatives which utilize Bloom filters as similarity digest, the Hamming distance as metric is used.

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*. A user has to consider that such a score gives more

[1] Note, the original work suggests to use k different hash functions each returning a value between 0 and $m - 1$. However, we use single hash functions and therefore our explanation differs slightly.

exact inferences, although it 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., in which offsets have been adapted).

RELEVANCE. Obviously, the application of Approximate Matching concepts to the field of memory forensics requires the adaptation of existing schemes. Schemes implicitly damp occurring effects like fragmentation (virtual memory), partially loading (lazy evaluation), or patching (loader). However, components of additional carving should be considered. We will further inspect past efforts and discussions of whether Approximate Matching is applicable to the field of memory forensics in Section 3.

2.1.2 *Binary Matching*

Research has discussed the utility of the schemes in the context of binary or malware analysis. A predominant representative in this field is still ssdeep[2][3]. In the past, evaluations of the different approaches led to unclear results with missing insights. Most often, matching results have been published without giving any reasoning as to why specific schemes perform differently for specific scenarios. In addition, test scenarios could vary widely, depending on the utilized datasets and test methodology. An extensive overview of different research was given by Pagani, Dell'Amico, and Balzarotti (2018). In short, French and Casey (2012) claimed that ssdeep and sdhash have varying capabilities to match malicious binaries. Oliver, Cheng, and Chen (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, Dell'Amico, and Balzarotti (2018).

Pagani, Dell'Amico, and Balzarotti (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 as to why specific schemes tend to fail in specific scenarios.

In summary, 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, Forman, and Cheng (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

[2] `https://www.virustotal.com/` (last access 2021-08-01).

[3] `https://www.nist.gov/publications/approximate-matching-definition-and-terminology` (last access 2021-08-01).

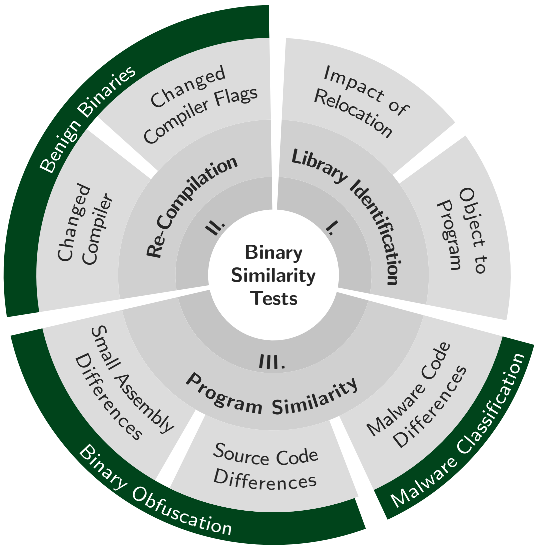


FIGURE 2.4: Overview of evaluation and test scenarios proposed by Pagani, Dell'Amico, and Balzarotti, 2018.

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.

EVALUATION SCENARIOS. Pagani, Dell'Amico, and Balzarotti (2018) examined the different schemes through the creation of three different scenarios. An overview of the different scenarios can be seen in Figure 2.4. 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 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 again causes problems for CTPH to recognize matching binaries. In the case of differing compiler flags ssdeep is again the worst performer. In contrast, 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 of 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 contrast, tlsh kept promising high similarity rates, because of its underlying frequency-based nature of n-grams.

SUMMARY. Recent research enlightens the reasons for 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. In the case of the different evaluation scenarios we can summarize the main findings of existing research: First, the distinction between data and code is of crucial importance and has an overall impact on the final inferences. Second, even small changes on the (source) code of samples or additional insertions influence the overall binary and code structure in a broad way. This especially has a great impact on CTPH-based approaches. To summarize, sdhash and tlsh clearly outperformed CTPH schemes. Each have their strengths and weaknesses in different disciplines.

RELEVANCE. We consider the task of matching executables or fragments of executables as an important aspect of further applications to the field of memory forensics. Carving segments of executed binaries out of memory not only has to consider the challenges of its application domain, namely physical memory, but also the challenges of malicious or benign differences on a binary level itself. We should additionally underline the relevance of the proposed evaluation binaries and scenarios. We will extend the original evaluation data set with binaries processed by additional modification passes, revealing additional weaknesses in existing schemes. Considering the fact that the x86 instruction set remains Turing-complete, even reduced to a single instruction, we additionally want to stress the fact that sooner or later all of the considered schemes will reach their boundaries of application (Dolan, 2013).

2.1.3 *Database Lookup Problem*

Approximate matching (a.k.a. fuzzy hashing or similarity hashing) is a common concept across the digital forensic community to do known file / block identification in order to cope with the large amounts of data. However, due to the fuzzy nature of approximate hashes current approaches suffer from the *Database Lookup Problem* (Breitinger, Baier, and White, 2014). This problem is based on the decision of whether a given fingerprint is a member of the reference dataset. The general database lookup problem is of complexity $O(n)$ in terms of the number of queries and hence exponential. To address this problem, techniques have been proposed such as multiple Bloom filters, a single large Bloom filter, a Cuckoo Filter or Hierarchical Bloom Filter Tree (Harichandran, Breitinger, and Baggili, 2016; Lillis, Breitinger, and Scanlon, 2017).

Besides the complexity, an investigator has to deal with *common blocks* which makes the identification of the correct match hard (Garfinkel and McCarrin, 2015). The extraction and correct assignment of a specific data fragment is of crucial importance, i.e., those identified chunks allow the inference of the original source (e.g., a potentially malicious file or a media file). Specifically, different files of the same type or application often share a non-negligible amount of *common blocks*, e.g., file structure elements in the file header. This leads to *multihits*. Hence, those blocks or chunks are not suitable for a unique identification of a specific file. To avoid this problem, lookup strategies should consider additional mechanisms to handle common blocks, e.g., by integrating functions of filtration or deduplication. Those requirements influence the applicability of a specific lookup strategy. The consideration of common blocks also influences the results of higher level analysis. An example is Approximate Matching used for the task of identifying similar binaries or detecting shared libraries (Liebler and Breitinger, 2018; Pagani, Dell'Amico, and Balzarotti, 2018).

We introduce three widespread lookup strategies suitable for storing hash-based fragments which have been proposed and are currently utilized in the field of digital forensics.

1. *hashdb*: In 2015, Garfinkel and McCarrin (2015) introduced *Hash-based carving*, "a technique for detecting the presence of specific target files on digital media by evaluating the hashes of individual data blocks, rather than the hashes of entire files". Common blocks were identified as a problem and have to be handled or filtered out, as they are not suitable for identifying a specific file. To handle the sheer amount of digital artefacts and to perform fast and efficient queries, the authors utilized a so-called *hashdb*. The approach was integrated into the bulk-extractor forensic tool. Both implementations have been made publicly available[4].

2. *hbft*: In the scope of Approximate Matching, probabilistic data structures have been proposed to reduce the amount of needed memory for storing relevant artefacts. Approaches to store artefacts comprise multiple Bloom filters (Breitinger and Baier, 2012b), single Bloom filters (Breitinger and Baggili, 2014) or more exotic Cuckoo filters (Fan et al., 2014; Gupta and Breitinger, 2015). One major problem of probabilistic data structures is the fact of losing the ability to

[4] https://github.com/simsong/ (last access 2021-08-01).

actually identify a file. In 2014, Breitinger, Rathgeb, and Baier (2014) provided a theoretical concept of structured Bloom filter trees for identifying a file. In 2017, a more detailed discussion and concrete implementation was provided by Lillis, Breitinger, and Scanlon (2017). The approach is based on "the well-known divide and conquer paradigm and builds a Bloom filter-based tree data structure in order to enable an efficient lookup of similarity digests". This leads to Hierarchical Bloom filter trees (*hbft*).

3. *fhmap*: In 2018, Malte Skarupke presented a fast hash table called *flat hash map*[5] (*fhmap*). The author claims that the implementation features the fastest lookups to date. A hash table features a constant lookup complexity of $O(1)$ given a good hash function. The database implementation provides an interface for accessing the hash table itself, however, it does not feature any image slicing, chunk extraction or hashing. Thus, in order to utilize and evaluate *fhmap* in our context, it has to be extended by additional concepts to extract data fragments comparable to Hash-based carving or fuzzy hashing.

SUMMARY AND RELEVANCE. In digital forensics the storage of forensic corpora is often discussed. Proposed approaches for artefact handling incorporate mandatory and non-mandatory features. Mandatory features in our considered use case are common block filtration and deduplication. In addition, techniques and implementations of closely related fields are also relevant and considerable. However, a direct comparison of the different approaches is hard. Besides offered features, a comparison in terms of performance and required resources is also relevant.

2.2 BINARY ANALYSIS

In this section we will introduce well-known problems in the field of binary analysis. Problems of relevance for this work, usually considered for the task of reverse engineering an unknown binary are as follows: First the following Subsection 2.2.1 gives a short introduction into the challenge of correctly disassembling x86-64 binaries. Afterwards, we will introduce the problem of reliably differentiating between code and data fragments in Section 2.2.2. Subsequently, we give required insights into the problem of function detection in Section 2.2.3. Research efforts of all of the aforementioned problems rely on the creation of a reliable set of ground truth binaries. In Section 2.2.4 we will introduce different aspects of establishing a reliable set of x86-64 ground truth binaries.

2.2.1 *Disassembling x86/x64 Instruction Set*

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. Additionally, executable sections could be interleaved by code and data sequences. As Wartell et al. (2011) already described, this system design

[5] "You Can Do Better than std::unordered_map: New and Recent Improvements to Hash Table Performance" presented by Malte Skarupke at C++Now in 2018; https://probablydance.com/2018/05/28/a-new-fast-hash-table-in-response-to-googles-new-fast-hash-table/ (last access 2021-08-01)

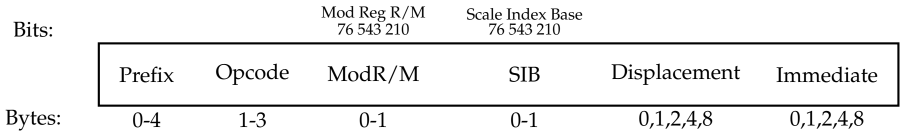


FIGURE 2.5: X86 Machine Instruction Format

trades simplicity for brevity and speed. In short, the process of disassembly in general is undecidable (Wartell et al., 2011; Andriesse et al., 2016). As can be seen in Figure 2.5, 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[6].

Recent research of linear disassemblers has shown the significant underestimation of linear disassembly and the dualism in the stance on disassembly in the literature (Andriesse et al., 2016). A more exotic form are length-disassemblers, which could be understood as a limited subset of linear disassemblers extracting only the lengths of an instruction. Besides the classical linear and recursive disassemblers, Shah (2010) introduced an experimental approach of fast and approximate disassembly. The approach is based on the statistical 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 32bit decoding scheme was proposed, which established a decent performance.

SUMMARY AND RELEVANCE. An important aspect of memory forensics is the extraction and analysis of code-related fragments. We further introduce the concept of approximate disassembling in the course of this work: the detection and extraction of sequences of code out of a large amount of unknown data (e.g., a previously acquired memory dump) via approximate interpretation of a processed bytestream. An approximate disassembler should process large amounts of unknown data. This desire clearly is contrary to the goal of classical disassembler engines, where computational performance is often understood as a secondary goal. Thus, we ignore approaches like 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. This should be additionally considered.

2.2.2 *Differentiating Code from Data*

Several publications presented in the course of binary analysis discuss approaches to differentiate code from data. In 2011, (Wartell et al., 2011) introduced a machine learning-based approach, disassembling and splitting x86 binaries into subsequences of bytes and classifying them as code or data. The approach uses a statistical data compression technique. The classification of the segments is performed with the minimum cross-entropy, identifying transitions between code and data segments. The classification makes use of a language model and different semantic heuristics to reclassify each segment as data or code.

[6] `https://software.intel.com/en-us/articles/intel-sdm` (last access 2021-08-01).

Zwanger, Gerhards-Padilla, and Meier (2014) introduced Codescanner, which can process an arbitrary input and detects code within different filetypes. The approach does not perform any disassembling of the input and uses different x86 constraints. The approach performs steps of data-prefiltering, byte-spectrum analysis, and code-pattern analysis. Several steps of prefiltering are additionally used, e.g., to ignore zero-byte paddings or high-entropy areas. During the byte-spectrum analysis a heuristic approach is used in contrast to a statistical approach, presuming "different semantic groups as being essential to the functionality of code". The classification step is performed by the "scarcity" of certain (opcode) bytes in code, expressed by a code score ratio specifying the quotient of frequent and rare bytes. In addition, common code patterns are taken into account, e.g., stack frames, function return sequences, and padding bytes.

In 2018, De Nicolao et al. (2018) introduced an approach called ELISA, "a technique to separate code from data and ease the static analysis of executable files". In addition, the approach performs a step of Instruction Set Architecture (ISA) identification. The approach showed a high accuracy in terms of identifying code sections on a byte-level.

SUMMARY AND RELEVANCE. 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. The major goals of our approach are the fast identification of code fragments, and in the best case, the approximate disassembly of those chunks to perform additional steps of normalization. Most of the introduced approaches obviously overlap with our further research ambitions. However, are not directly applicable in our context.

2.2.3 *Function Detection Problem*

The analysis of unknown binaries often starts with the examination of function boundaries. Functions are a fundamental structure of binaries and most often an initial starting point for advanced code analysis. As an important structural component of code, they give a schematic representation of the original high level semantics and provide a basis for further inferences. Whereas disassemblers are capable of reliably decoding the instructions of a binary, the problem of function detection is still an ongoing field of research. Binary analysis research claims that the problem is not yet fully solved. New techniques tend to improve in performance and generalizability, i.e., by the introduction of compiler- or even architecture-agnostic approaches (Potchik, 2017; Andriesse et al., 2016; Andriesse, Slowinska, and Bos, 2017; Shin, Song, and Moazzezi, 2015; Bao et al., 2014). Functions are used to infer the functionality of a given binary and thus, could be used to identify an unknown sample. In more general terms, two binaries that share many similar functions are likely to be similar as well (Jin et al., 2012). To summarize, functions could be used to identify, distinguish or interpret unknown code sequences.

Besides the field of extended binary analysis, the detection of function boundaries is implicitly relevant for other application domains. However, depicting the correct function detection technique requires considering the present environmental circumstances and constraints, e.g., in the context of memory carving most often only signature-based approaches are applicable. The examination of process-related code fragments is obviously one major benefit of memory-based forensical investigations. After successfully reconstructing the running binary out of a process context, steps of binary analysis and reverse engineering are applicable (Ligh et al., 2014). The reconstruction could be hindered by malicious or legitimate changes. Additionally, remaining fragments of already terminated processes are possibly ignored, due to missing structural properties. In the case of Linux operating systems the generation of an adequate memory profile could be cumbersome. The continuous development of operating system internals and its related structures require the constant maintenance of interpretive frameworks. Thus, even if the interpretation of operating system related structures is a fundamental component of memory analysis, carving could give a first solid impression or even be a last resort during examination.

In the course of function detection, machine-learning approaches have been proposed, which are trained to recognize signatures located in function prologues or epilogues (Shin, Song, and Moazzezi, 2015; Bao et al., 2014). Static function prologue signature databases have to be maintained over time and the detection performance of those techniques rapidly decreases for highly optimized binaries (Eagle, 2008; Guilfanov, 2012). Machine-learning-based approaches try to generalize this task and automate the process of signature detection. Besides those signature-related approaches, Andriesse, Slowinska, and Bos (2017) introduced a compiler-agnostic approach in the context of extended binary analysis, which is mainly based on structural Control Flow Graphs. Moreover, their research showed significant concern for all top-tier work on machine-learning-based approaches, mainly caused by the usage of a biased dataset.

LINEAR FUNCTION DETECTION. Considering the function identification process in the field of memory carving, the conditions exclude most of the extended and agnostic approaches. Those have been proposed in the field of extended binary analysis and require steps of binary lifting, control flow analysis or value-set analysis. In the course of memory carving, we further denote suitable function detection approaches as *linear* techniques. Those approaches do not rely on the reconstructability of binaries and could also be used for context-unaware memory analysis.

The enumeration of unknown functions was first established with the generation of signature databases. Signature databases focus on proprietary compilers, as open source compilers create an unmanageable diversity of function prologues (Eagle, 2008; Guilfanov, 2012). Especially in the case of Linux operating systems, a database lookup of saved signatures during carving a memory image would not be feasible. In Bao et al. (2014) a Weighted Prefix Tree (WPT) was introduced to potentially identify function start addresses. Therefore, they "weight vertices in the prefix tree by computing the ratio of true positives to the sum of true and false positives for each sequence" in a reference data set. The authors additionally introduce an additional step of normalization, which improves precision and recall. The authors created a set of 2,200 Linux and Windows binaries. The executables were generated with different build settings, i.e., the authors used GNU `gcc`, Intel `icc` and Microsoft Visual Studio. In addition, different optimization levels were selected during build time. Their approach, called Byteweight, was also integrated into the Binary Analysis Platform (BAP)[7].

In Shin, Song, and Moazzezi (2015) the authors provide an approach for function detection based on artificial neural networks. The paper proposes a function detection technique with the help of Recurrent Neural Networks. In contrast to our work, the approach of Shin, Song, and Moazzezi (2015) was performed without an

[7] `https://github.com/BinaryAnalysisPlatform/bap` (last access 2021-08-01).

Source	System			Description (ELF, Linux)
	WIN	LIN	OSX	
Byteweight	✓	✓	✗	ELFs (129): coreutils, binutils and findutils; used by Bao et al. (2014) and Shin, Song, and Moazzezi (2015);
Nucleus	✓	✓	✗	ELFs (521): real-world applications and the SPEC CPU2006 Benchmark Suite; see Table 2.3 for details;
CGC Corpus	✓	✓	✓	ELF binaries of custom-made programs specifically designed to contain vulnerabilities;

TABLE 2.2: Overview of exiting ground truth and evaluation dataset similar to Potchik (2017).

additional step of disassembling or normalization. The authors point out that the tracking of function calls over large sequences of bytes is not feasible. In fact, recognizing entry and exit patterns by training with fixed-length subsequences is eligible. For training and testing, the work is based on the same data set provided by Bao et al. (2014).

Andriesse, Slowinska, and Bos (2017) claim that the work of Shin, Song, and Moazzezi (2015) and Bao et al. (2014) suffers from significant evaluation bias, as the most of the samples contain large amounts of similar functions. The authors additionally mention that the viability of machine learning for function detection is not yet decided. The publication proposes a compiler-agnostic approach called Nucleus, which is mainly based on the examination of advanced control flow analysis and does not rely on any signature information.

Potchik (2017) introduced the integration of Nucleus in the Binary Ninja Reversing Platform[8] and proposes multiple strategies over multiple analysis passes rather than just relying on heuristics. The author mentions the possible reduction of complexity and scope reduction, by applying the technique with the highest confidence first. Similar to other fields, the approach proposes "a method to interpret the semantics of low-level CPU instructions" by the utilization of value-set analysis. The process of value-set analysis is performed on an extended intermediate language and thus should be architecture agnostic.

SUMMARY AND RELEVANCE. As we want to inspect code fragments in large amounts of data within a sliding window, many of the considered approaches are not applicable in our context or at least need to be modified. Thus, we have to consider the linear characteristic of our application, which in turn leads us to a signature-based or machine-learning-based approach like WPT- or RNN-based approaches.

2.2.4 *Reliable Ground Truth Binaries*

In recent publications, different sources of ground truth binaries have been proposed and criticized. In this paragraph we give a short overview of the different sources and outline some details of capacity and source. As we focus on the domain of Linux executable binaries, we formally introduce ELF binaries contained in different test suites. A comprehensive overview of the different test suites is given in Table 2.2 which have been public available[9] at the time of writing.

[8] https://binary.ninja/ (last access 2021-08-01).

[9] https://github.com/Vector35/function_detection_test_suite, https://github.com/trailofbits/cb-multios, http://security.ece.cmu.edu/byteweight/ (last access 2021-08-01)

samples	**arch**		**compiler**		**language**		**optimization**				
	32	64	gcc	llvm	C	C++	O0	O1	O2	O3	OS
SPEC	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
glibc	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
server	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗
count	200	321	321	200	360	140	100	100	100	100	100

TABLE 2.3: Overview of Nucleus (ELF) ground truth obtained by Andriesse et al. (2016) (gcc-510, llvm-370).

The work of Bao et al. (2014) and Shin, Song, and Moazzezi (2015) are criticized by Andriesse, Slowinska, and Bos (2017) for using a biased data set, with a large amount of overlapping and similar functions. Andriesse et al. (2016) outlined that the average binary in their SPEC-based test suite contains less than 1 % of shared functions, not considering bootstrap functions. We base our analysis on the data set introduced by Andriesse et al. (2016) and perform a detailed examination of the function structures in Section 6.1.4.

The Nucleus data set consists of approximately 4.2 GiB precompiled ELF files and its corresponding ground truth assembly structure. The process of data set generation depends on some major parameters: operating system, instruction set architecture, language, compiler, and optimization level. The 521 binaries consist of the SPEC CPU2006 Benchmark Suite and some real-world applications written in C and C++. The samples are compiled for x86 and x64 with five different optimization levels (O0-O3 and Os). The set contains dynamically and statically linked binaries, where some of them are stripped and some are equipped with symbols. For further details on the construction of the ground truth we refer to Andriesse et al. (2016). An overview of the binaries is given in Table 2.3.

SUMMARY AND RELEVANCE. In the realm of binary analysis, several datasets have been proposed and utilized in recent literature. Besides the size and diversity of proposed ground truth datasets, the quality and usefulness has to be considered as well. Several binary-related properties, like function size or function prologue distribution, should be considered and need to be determined.

3

SYSTEMATIZATION OF MEMORY FORENSICS

In this chapter we provide a comprehensive overview of past research efforts in the field of memory forensics. We therefore categorize different publications by their approximate type of contribution. We will further describe the mentioned categories in detail in Section 3.1 and introduce the raw results of our systematization. Afterwards, we will outline in Section 3.2 different aspects of unstructured analysis by selected publications. In Section 3.3 we will introduce research discussing the applicability of Approximate Matching in the field of memory forensics. Finally, we will define two major research boundaries, which have to be respected in the course of this work in Section 3.4.

3.1 CATEGORIES OF SYSTEMATIZATION

Considering the subfields of research and the ever changing historical relevance, several ways of differentiation are imaginable. We propose a differentiation by eight categories: Analysis Type (AT), Target System (TS), Acquisition (AC), Inconsistencies (IN), Artefact Type (AR), Obstacles (OB), Aspects (AP), and Tool/Framework introduction (FW).

- **Analysis Type (AT).** We distinguish the approaches by their general core functionality. First, the analysis based on interpretation of structures called Structured Analysis (S). Most tools and frameworks utilize structured analysis, i.e., the software interprets the complex system related structures. In detail, the frameworks deal with different formats of acquisition, the concepts of virtual memory management, the present architecture and the operating system (OS) related structures. Memory profiles are used to close the semantic gap and are required to perform a structural examination.
  Second, the extraction of artefacts via concepts of Memory Carving or Unstructured Analysis (U). There are also tools for unstructured analysis to extract information out of memory dumps. Those tools are important for different tasks like string or key extraction. Carving memory has the advantage of being more robust against malicious evasion or domain specific deallocations of important structures. In addition, those tasks can achieve a high IO throughput, are well parallelizable and offer a fast access to valuable insights.
- **Target System and Platform (TS).** Another criteria to differentiate publications is by their considered target, i.e., operating system and platform. In this context, the target systems vary over time, significantly influenced by relevance

and popularity. The most important operating systems are obviously Windows (W), Linux (L), and Mac OS X (M). In addition, mobile devices received increasing attention due to their rapidly growing popularity and market dominance. We further consider Android (A) and Symbian (S) for mobile memory forensics research.

- **Acquisition Type (AC).** Because of its criticality and importance, much effort was invested in researching and assessing new and existing acquisition methods. A wide variety of acquisition methods have been researched, discussed, and further developed in recent years besides the analysis of Live (L) systems. Basically, we can differentiate a form of acquisition in terms of a software-based (S) or hardware-based (H) technique. Hardware-based techniques include, for example, the use of DMA-based (D) attacks. A technique based on both hardware- and software-based aspects is the acquisition via cold boot (C), which exploits the remanence properties of main memory to extract data from RAM after a reboot. Some techniques require the installation of the acquisition procedure prior to the actual occurrence of an incident. These pre-installed (P) techniques can include both hardware- and software-based procedures. An important form of acquisition and the reliable basis for the development of tools is the acquisition of virtualized system (V). Virtualization-based acquisition methods represent one of the most atomic acquisition methods available. Various features of the operating system perform the copying of special memory areas, for example, in the case of swapping (W) or hibernation (I). Crash dumps (R) and caches (A) introduce other data sources to be considered. Other forms of acquisition can be based on exploiting vulnerabilities (U), memory page tables (T), or the utilization of customized firmware (F). Many of the approaches support the transfer of acquired data over network (N). The relevance and importance, as well as the correctness, of the techniques have been further discussed (D) in a large body of work.

- **Deals with Inconsistencies (IN).** Considering the different sources and methods of acquisition, as well as the volatile nature of working memory, the handling of inconsistencies plays an important role in today's research. The persistence (P) of artefacts within the memory over the actual time of use has been inspected, trying to answer the question of how long artefacts remain in main memory under different conditions. Inconsistencies caused due to the volatile nature of the main memory and the acquisition of a running target system are often denoted as memory smearing (S). In addition, we consider inconsistencies caused by the swapping of memory fragments (W) or effects caused during the process of hibernation (I). Established concepts of the operating system such as lazy evaluation (L) or newer mechanisms such as memory compression (C) have also been considered. Another important aspect is introduced inconsistencies caused by a selected acquisition method itself, considering the integrity of the acquired dump and the footprint (F) of the respective acquisition method.

- **Artefact Type (AR).** The progress of the research is largely tied to the needs of the analysts, with their enhancements forming the foundation for further implementations. The continuous extraction of new artefacts probably plays one of the most important and obvious roles in differentiating the various works. These artefacts include, for example, (open) files (F), network connections (N), (terminated) processes (P/T), media such as images or videos (V), databases

(D), kernel modules and drivers (K), pool allocations (O), timelines (T), keys and passwords (E), log files (L), communications (C), command history (N), GUI fragments (G), command line arguments (A), clipboard (B), environment variables (E), user sessions (U), browser artefacts (R), handles (H), memory mappings (M), stack artefacts (S), and heap (h).

- **Obstacles (OB).** Besides challenges considered in terms of inconsistencies, researchers and practitioners address other roadblocks caused by malicious intentions or by different system-dependent extensions: anti-memory forensics (A), OS security features (S), OS extensions and kernel extensions (E), profile creation and initialization (P), malicious software (M), DKOM-based obfuscation (D), device-specific challenges (V), and kernel fingerprinting (F).

- **Aspects (AP).** The category of general aspects includes publications about topics like virtualization (V), machine learning (M), user space analysis (U), application analysis (A), visualization (Z), correlation (C), formats (F), jurisdiction (L), ground truth generation (G), de-anonymization (D), and surveys (S). In the course of this thesis, we are especially interested in publications, addressing the applicability of cryptographic hash functions (H) and concepts of fuzzy hashing (Y) in the field of memory forensics.

- **Tool and Frameworks (FW).** The majority of publications are proposed with additional proof of concept implementations (I) and framework extensions. Those extensions (E) could be implemented for existing frameworks like Volatility (V) or Rekall (R).

Table 3.1 is a comprehensive overview of all considered publications (P) and their specific categorization. Beside the mostly peer-reviewed publications, we additionally consider selected talks (T) from relevant and impactful conferences. Awarded publications (A) are additionally highlighted.

Table 3.1: Overview of different publications in the field of memory forensics sorted by year of publication and last name of the first author. Columns: Target System (TS), Analysis Type (AT), Acquisition (AC), Inconsistencies (IN), Obstacles (OB), Aspects (AP), Implementation (FW), Artefact (AR).

Year (A | P | T) **Lead author** *Approach* (**A**ward/**P**ublication/**T**alk)	**TS**	**AT**	**AC**	**IN**	**OB**	**AP**	**FW**	**AR**
2004								
P Carrier *Tribble*	W		HP		A			
2005								
T Becher *Firewire*	LM		HLP					
2006								
A Schuster *PTFinder*	W	U						OP
P Adelstein *Discussion*								
P Kornblum *Rootkit*	W		HSV		M			
P Petroni Jr *Fatkit*	LW	S		MW	P	Z	I	
P Schuster *PoolFinder*	W	U			A			ON
P Walters *Fatkit*	W	SU			AM	H	ER	P
T Boileau *Firewire*			H					
2007								
P Arasteh *Stack*	W	S				U		S
P Dolan-Gavitt *VadWalk*	W	S			AM		EMA	
P Garcia *Summary*	LW	SU	D			S		
P Huebner *Persistent OS*	LW		HSIP	MWI		V		
P Kornblum *Buffalo*	W	S		WL				
P Schatz *Bodysnatcher*	W		SVD		AM	V		
P Solomon *TimeStamp*	LW	U		P				
P Vidas *Acquisition*	W	SU	HS	M	AD			P
P Walters *VolaTools*	W	S			ASD			PNM
T Rutkowska *Northbridge Exploit*			H		A			
2008								
P Case *FACE ramparser*	L	US			L	CZ		KPNF
P Cohen *PyFlag*						ZCA		RN
P Cozzie *LAIKA*	LW	U			M			P
P Dolan-Gavitt *WIN Registry*	W	US			AEM		A	D
P Enck *Anti CB - MECU*			C		A			
P Hargreaves *TrueCrypt*	W	U						E
P Kiley *Messager*	W	U	W					C
P Libster *Crash-Ctrl-Scroll*	W		SP	MF	AMK			
P Ruff *Overview*	W	SU	HLRVS	MFW	AM			
P Schuster *Nonpaged pool*	W	U	V	PF	M			OP
P Sutherland *Tool Footprint*	W		S	F		L		
P Van Baar *Mapped Files*	W	SU		M		H		
T McGregor *Anti CB*			C		A			
T Suiche *WIN Hibernation*	W	S		I	E		E	
2009								
P Aumaitre *WIN Firewire*	W		H		A			PE
P Chan *ForenScope*	L		C				I	
P Dolan-Gavitt *Robust Signatures*	W	U			AEM		ER	P
P Halderman *CB - AES/RSA*	LWM		C		S			E
P Hejazi *Stack*	W	SU				AU	I	PES
P Heninger *RSA reconstruction*			C					E
P Maartmann-Moe *Interrogate*	W	US						E
P Simon *Survey - Future*						HY		
P Zhang *KPCR*	W	S						KP
P Zhao *Hexview*	W	U	RSI					
2010								
A Okolica *PDB*	W	SU			P		E	DNPK
P Case *Kernel Structures*	L	S			PM		E	PFMAV
P Case *KMEM Cache*	L	S	A	P				PNFHM
P Müller *Anti CB - AESSE*	L		C		A		I	E
P Okolica *CMAT*	W	SU			MK	CF	I	PDNUH
P Rabaiotti *XBox*	WE		SU	F	S			
P Saur *Paging Structures*	LW		V		MKE	V		P
P Simon *Skype*	W	SU	U		A	UA	R	CEP
P Stevens *CMD History*	W	U					IRV	N
P Thing *Memgrab*	LA	U	SP	P	E		I	PC
P Vidas *Warmboot*			C					
P Witherden *Libforensic1394*	LWM	U	HD				I	E
P Zhang *WIN 7*	W	S						PDU
2011								

P Aljaedi *Live vs. image analysis*	L	U	L	FP	AM	A		PTER
P Beverly *Scan-Net NW Carver*	LWM	U	HW	C		CG	EA	N
P Case *TAILS live CD*	L	S				AD	IRV	FR
P Inoue *Dotplots OS x*	M	U	SHDP			ZG		
P Lin *SigGraph*	L	S			MK		I	FNPUM
P Müller *Anti CB - TRESOR*	L	U	HCP		A		I	E
P Okolica *Clipboard*	W	SU					IERO	B
P Okolica *WIN Drivers*	W	SU			E		E	N
P Simmons *Anti CB - Amnesia*	L		C		A			E
P Vömel *Survey*	W	U	*		AMS	S		*
2012								
A White *User Space*	W	S				U	IEV	hVLH
P Eschweiler *Discussion*			D	A				
P Gu *OS-Sommelier*	WL	SU			FP	GH		
P Müller *TREVISOR*	LW		HC		A	V		E
P Olajide *User Input*	W	U	V	P		A		
P Reina *SMMDumper*			F					
P Sylve *DMD (LiME)*	LA	S	PSN		V		IEV	K
P Vömel *Criteria AC*			D					
T Haruyama *Anti - One-Byte*	W	SU	S		A		T	
2013								
P Graziano *Actaeon*	H	U				V	IEVR	
P Gruhn *CB Pracitcability*			DC					
P Müller *FROST*	LA		C					FEC
P Stüttgen *OS-agnostic AC*	LWM		T	A				
P Thing *Symbian AC*	S		S					
P Vömel *Evaluation Platform*	W		D					
P Vömel *RKfinder*	W	SU			AMK	Z	IEVR	
P White *Hashtest*	W	SU			MD	UAH	IEVR	
2014								
A Richard III *Swap compression*	LM			MWC	SE		IEV	
A Stüttgen *Parasite LKM AC*	L		SU					
P Hilgers *DVM Volatility Analysis*	LA	S	C				IEV	hCNE
P Prakash *Evaluation*	W				MK	G		
P Roussev *Kernel FP - SDHASH*	LW	U			FP	HY		
P Suma *Win7 64bit DTB*	W	SU						P
P Sun *TrustDump*	LA		SPN					
2015								
P Balzarotti *GPU Malware*					AMK			
P Case *RK Detection*	M	S			MK		IEV	
P Cohen *Kernel Fingerprinting*	W	S			FP			
P Gruhn *WIN Pagefile*	W	SU	W					
P Heriyanto *Pracitcability*	LA		SD			S		
P Lindenlauf *CB DDR2-DDR3*			C	P				
P Saltaformaggio *GUITAR - GUI*	LA	S					I	GH
P Saltaformaggio *VCR - Camera*	LA	U						v
P Seitzer *Anti CB - Bytecode*	L		C		A			
P Stüttgen *Firmware*	LW		S		MK		IEVR	
P Taubmann *Android - CB BMA*	LA		C	PF			IEV	
P Wächter *Pracitcability Android*	LA		SD		S	S		
2016								
A Case *Objective-C Malware*	M	S			KMS		IEV	PU
P Bauer *Descrambling DDR3*			C					
P Gruhn *AC Criteria*			*	MP				
P Lee *Anti MF*	W	SU		P	A		T	PK
P Saltaformaggio *RetroScope*	LA	S						IN
P Sylve *Pool Tag Scanning*	W	U						O
2017								
A Block *Heap*	L	S					IER	PENUh
P Case *Discussion*	**	*	*	*	*	S	*	
P Cohen *YARA Scanning*	W	US			M			
P Barabosch *Quincy - injection*	W	SU			AM	G	IEVT	
P Case *Fuzz Testing*	LWM			M			VRT	
P Iyer *E-Mail Spoofing*	W	U						C
P Lapso *Visualization*	W					Z		
P Pridgen *RecOOP - runtimes*		S						
P Sylve *WIN Hibernation*	W	S	I	I	E	F	E	
P Yang *AMD AC firmware*	LA		FU					
P Yitbarek *Descrambling DDR4*			C					E
P Zheng *Anti CB - Android*	LA		C		A			E
2018								
A Lewis *WSL*	WL	SU			E		EVT	PVMHN
P Al-Sharif *MS Word Documents*	W	U						F
P Barmpatsalou *Mobile Forensics*	A					S		
P Bhatia *Timeliner Apps*	LA			P				IN
P Bhatt *Kernel FP Codeid-elf*	L	U			FP			

P Otsuki *Stack Win-x64*	W	S			M			US
P Palutke *Styx - anti AC*	L		S		A	V	ER	
P Rodríguez *ProcessFuzzyHash*	W	S				HY	IEV	P
P Stadlinger *Userland Heap*	L	S					ER	ENFUh
2019								
A Block *Hidden Injected Code*	W	S			AM		IER	
P Ali-Gombe *DroidScraper*	LA	S						FDEUPh
P Case *hooktracer*	W	S			M		IEV	
P Casey *Vivedump*	W	S					IEV	
P Kazim *Chat*	W	S						CE
P Latzo *Taxonomy*			D			S		
P Pagani *Discussion*	L	S				S		
P Pagani *Inconsistencies*	L		S	MP				Sh
2020								
P Case *OS X Page Queues*	M	S		M	E		IEV	
P Latzo *BMCLeech*			PH					
P Palutke *Anti MF*	WL	S			AM		IER	U
P Schneider *Tampering*						L		
P Thomas *Cryptocurrency*	W	S				Z	IEV	E
P Uroz *Sigcheck*	W	S		P	M		IEV	

As shown in Table 3.1, an ambitious and broad community has continuously advanced the field of memory forensics. Different aspects, problems, and applications of the field are steadily further developed. Aspects and topics of interests change over time and adopt to current environmental changes, e.g., target-specific updates.

3.2 UNSTRUCTURED ANALYSIS

The community and industry came up with different solutions for the analysis of acquired memory dumps, which can be differentiated into one of two major categories: memory carving (unstructured analysis) or structured analysis. Considering the field of unstructured analysis, several different approaches have been proposed:

MEMORY CARVING. In the beginning of memory forensics, analysts extracted important and meaningful OS-related structures out of the memory dump by the previous identification of robust signatures. In 2006, Schuster presented his tool PTFinder, a tool which utilized different signatures in the header of pool tags to identify those allocations (Schuster, 2006b). The approach has further been extended with a pool scanner for network connections and the researcher formulated foundational rules for pool-carving for Windows operating systems (Schuster, 2006a). In 2008, Hargreaves and Chivers introduced an approach to extract TrueCrypt encryption keys by linear scanning an image. The approach analyses every position within an image and tries to extract possible key structures according to a predefined pattern.

Considering the importance of signature-based analysis, the creation of robust signatures in the case of kernel data structures has been further inspected by Dolan-Gavitt et al. (2009), who presented an automated approach for generating robust signatures of kernel data structures. The researchers created profiles of commonly used fields and analysed their overall impact on the system stability.

Carving has been further discussed in recent years, for example to extract the command prompt history (Stevens and Casey, 2010), network-related artefacts (Beverly, Garfinkel, and Cardwell, 2011), visual content (Saltaformaggio et al., 2015b), Linux kernel versions (Bhatt and Ahmed, 2018; Roussev, Ahmed, and Sires, 2014), and documents (Al-Sharif, Bagci, and Asad, 2018).

APPLICATION OF YARA. The application of a YARA[1] rule as signature-based analysis for the examination of memory was recently discussed by Cohen (2017). The research underlines the idiosyncrasies, pitfalls, and needed adaptations for applying signatures to this domain. The author described a context-aware scanning scheme on the physical address space using the Windows Page Frame Number (PFN) database, which could be used to map each physical page to a corresponding process. By the examination of physical memory dumps, the approach still gains a reasonable performance, which is caused by an optimized IO throughput. Contrary to the application on hard disk images the authors discuss the applicability, expandability, and required adaptations of pattern matching rules in the course of memory analysis.

HASH TEMPLATES. Research proposed in 2008 (Walters, Matheny, and White, 2008) and 2013 (White, Schatz, and Foo, 2013) covered the utilization of cryptographic hash functions to perform code integrity checks, tamper detection, whitelisting, and blacklisting. 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 White, Schatz, and Foo (2013). As proposed by Walters, Matheny, and White (2008), offsets of variable code fragments have been used to normalize and hash executables on a page level. A database of hash templates was created which consists of hash values and their 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 $O(n \cdot m)$ for a comparison of n templates against m memory pages. Researchers (White, Schatz, and Foo, 2013) extended the approach and tried to improve the naive all-against-all comparison introduced by Walters, Matheny, and White (2008). 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. The identification of variable offsets by imitating the loading process of an executable seems legitimate. A normalization based on previously disassembling a present sequence of bytes in memory was not mentioned by the authors. A public available Volatility plugin[2] provides a whitelisting similar to White, Schatz, and Foo (2013). The approach performs a lookup on a page level of executables. Therefore, the memory is processed and sent to a hash server. In contrary to White, Schatz, and Foo (2013), the lifting of the code is performed on the server, which creates integrity hashes with the help of the virtual address of the process.

HASH-BASED CARVING. Considering the field of memory forensics and its introduced conditions, we inspected approaches that are related to our task of identifying fragmented code structures. Garfinkel and McCarrin (2015) presented an approach that covers the main considerations and pitfalls of our work in general terms. In contrast to a whole-file hashing, a concept called hash-based carving was introduced, which can identify files that are fragmented, files that are incomplete, or files that have been partially modified. The publication covers similar aspects of

[1] `https://github.com/VirusTotal/yara` (last access 2021-08-01).

[2] `https://github.com/K2/Scripting/blob/master/inVteroJitHash.py` (last access 2021-08-01).

our work within a general scope. The approach is mainly considered in the field of sector-based volumes and the sliding-window-based extraction is sized to 4 KiB. The overall process is computationally demanding but highly parallelizable. The authors make use of a previously introduced hashdb (Young et al., 2012) and outline their real-world experience by the utilization of hash-based carving. A major contribution of the work is the discussion of classifying blocks and the negative impact of common blocks, which are often shared between different documents. Their work shows the problem of a high false identification rate caused by large amounts of shared blocks within the processed document classes.

3.3 APPLICATION OF (FUZZY) HASHING

Different authors mentioned or questioned the application of Approximate Matching in the course of memory forensics (White, Schatz, and Foo, 2013; Ligh et al., 2014). In the specific case of matching executables on disc to its counterpart loaded in memory, most of the authors doubt the usefulness of cryptographic hash functions on the raw sequences in memory. One major reason are legitimate changes to the code sequences caused by the loading process itself. However, the application of Fuzzy Hashing was considered as possible and relevant for further research. As already introduced, Approximate Matching algorithms can be used to detect similarities among objects, but also to detect embedded objects or fragment of objects (Breitinger and Baier, 2012b; Roussev, Richard III, and Marziale, 2007). Investigators can use it to distinguish 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 (Gupta and Breitinger, 2015) and have already found their way into the field of memory forensics. In the following paragraphs, we will shortly introduce considerable examples of application.

PROCESS RELATED HASHING. In 2010, researchers (Ligh et al., 2010) proposed a script (`ssdeep_procs`) which enumerates running processes on a system, dumps them to hard disk and compares the extracted executables with the help of ssdeep (Kornblum, 2006). The Volatility plugin called `impfuzzy`[3] applies Fuzzy Hashing on the Import API of PE files to detect malicious changes. In 2018, researchers published a Volatility plugin called *ProcessFuzzyHash* to compute approximation hash values of processes contained in a Windows memory dump (Rodríguez, Martín-Pérez, and Abadía, 2018).

In 2012, researchers (Gu et al., 2012) presented an approach called OS-Sommelier. The approach was proposed for memory only, precise and efficient cloud guest OS fingerprinting via kernel code hashing. The authors identified two major challenges for this task: (1) differentiate main kernel code from the rest of code and (2) normalize kernel code to deal with variations, e.g., caused through Address Space Layout Randomization (ASLR).

KERNEL FINGERPRINTING. In 2014, Roussev, Ahmed, and Sires discussed the application of Approximate Matching as a content-based method for reliably identifying kernel versions. The researchers utilized SDHASH for the task of kernel fingerprinting. Identifying the correct kernel version is considered as one of the first

[3] `https://github.com/JPCERTCC/impfuzzy` (last access 2021-08-01).

important steps for establishing deeper inference via structured memory analysis. The researchers discussed the challenges of identifying the correct version for open source operating systems. The proposed approach was considered as more robust than relying only on small signatures, as the generated digests represent the whole content of the kernel image, retrieved from a disk image. The approach was demonstrated across different architectures without the need to parse and contextually interpret the memory dump (Roussev, Ahmed, and Sires, 2014).

3.4 DISTINCTION AND RESEARCH BOUNDARIES

Many approaches shown in Table 3.1 are context-aware and fall into the category of structured analysis. In contrast to the introduced approaches, our approach aims to extend current techniques of unstructured analysis and the creation of new forms of data-driven cross validation and cross verification. Thus, the overall approach should keep several properties of application, defined by the domain of Approximate Matching and memory carving:

1. **Computational Performance.** Our research should keep several criteria of computational performance and should follow properties introduced by different concepts of Approximate Matching. Expressed more informally, proposed approaches and techniques should always be considered for the task of data reduction and bulk data processing.

2. **Structural Independence.** Our research should not rely on any structural interpretation. Thus said, we do not consider any steps of structural analysis. In this research, we already consider steps of virtual-to-physical mapping or the recreation of an existing process context as steps of structural interpretation.

4

APPROXIMATE DISASSEMBLING - APPROXIS

4.1 INTRODUCTION

In this chapter we introduce a technical acquisition (carving) component called 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 usage in digital forensics, however, the technique may also be used in different fields like real-time systems. 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 Andriesse et al. (2016), we build up a decision tree of byte instructions. Each path of the tree represents the decoding process of a byte sequence into 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 extend the capabilities of a simple length disassembler.

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.

RELATED WORK AND PROJECTS. Recent research of linear disassemblers has shown the significant underestimation of linear disassembly and the dualism in the stance on disassembly in the literature (Wartell et al., 2011). 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. Besides the classical linear and recursive disassemblers, Shah (2010) introduced an experimental approach of fast and approximate disassembly. The approach is based on the statistical 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 32bit decoding scheme was proposed, which showed decent results.

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 (Wartell et al., 2011). The major goals of our approach are the fast identification and the approximate disassembly of code fragments. For further details and relevant research we refer to Chapter 2.2.

REQUIREMENTS OF APPROXIMATE DISASSEMBLING In this paragraph 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 contribution. As the application of Approximate Matching algorithms to portions of memory seems unfeasible due to an 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 differently 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 or compiler of the code.

These practical goals describe the purpose of this chapter, 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 significantly 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 White, Schatz, and Foo, 2013) relies on the correct identification of a running process. This offers new degrees of bypassing and obfuscation to the malware author. Our second requirement, *robustness*, means to identify a code fragment without process structures and thus being 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 consider 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.

	32bit (200 files)				64bit (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

TABLE 4.1: Overview of unigram and bigram mnemonic counts.

A *versatile* approach is desirable; one which is not dependent on an a-priori knowledge of the architecture of the target system (i.e. x86/x64). The requirement *versatility* means that the disassembler works reliably for different target architectures.

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 Andriesse et al. (2016). As we focus on the acquisition of byte sequences which rely on code only, we extracted the `.text` section of each binary file. It should be mentioned that the following Most Frequent Occurred (MFO)-distribution analyses is nothing new (Bilar, 2006; Shah, 2010). 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 4.1). We divided 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* and *mean* describe the assignment of the *total* amount of instructions to each *distinct* unigram or bigram. For example, the most frequently occurring mnemonic in the case of 32bit binaries represents $33,25\%$ of all instructions.

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 in Equation (4.1). Similar to Shah (2010) we want to avoid computational underflow by multiplication of probabilities.

$$\lambda = |\ln \frac{p}{1-p}| \tag{4.1}$$

BYTE TREE ANALYSIS. The former paragraph revisits the frequencies of most frequently occurring mnemonics. In a next step we analyze the byte frequencies on an 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 to 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 a tree. Each node of the tree represents a byte and stores a reference to all its corresponding children, the subsequent instruction bytes (see Figure 4.1).

As an example we inspect the byte sequence `488d` and its subsequent bytes after inserting our ground truth into the tree. In Figure 4.2 we can see the complete output of a single node. We should mention that the amount of the child nodes was shortened for better representation. We also save auxiliary information like the amount of counted bytes for a current node (line 2), the counts of different occurring instruction lengths (line 4), and the counts of all corresponding mnemonics (line 5). 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.

Input instructions:

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

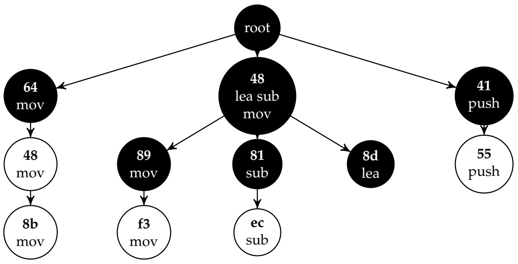


FIGURE 4.1: Oversimplified approxis bytetree example after inserting several instructions.

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

FIGURE 4.2: Inspecting a node of `lea` (`48 8d`) instruction

After inserting the 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 into a leaf node. Therefore we cropped all subsequent child nodes of the current node, which does not affect the instruction mnemonic. The reduced shape of the tree is highlighted in black in Figure 4.1. The impact of reduction can be seen in Table 4.2.

4.2 DISPATCHER APPROACH

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

DISASSEMBLING. We argue that length-disassemblers could be assumed to be very fast and lightweight. However, 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`[1] 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 value 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

[1] `https://github.com/gdabah/distorm` (last access 2021-08-01).

platform	**input bytes**	**original tree**			**reduced tree**		
		nodes	height	size	nodes	height	size
64bit	253.535.572	12.773.078	15	445M	87.224	10	7,5M
32bit	123.221.439	5.871.232	15	206M	35.211	9	3,0M

TABLE 4.2: Comparison of original and reduced bytetree.

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

TABLE 4.3: Simplified comparison of capabilities of an approximate disassembler compared to other classes of disassemblers.

that existing disassemblers are not designed for our field of application. Processing a large amount of raw data is outside of the scope of classical disassemblers. As existing length-disassembler engines reduce the amount of required 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. In Table 4.3 an overview and comparison of the considered classes of disassemblers is given.

BYTETREE DISPATCHING. To address the introduced requirement *lightweight* (see Section 4.1), 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.

The performance of the bytetree was evaluated by a set of 1318 64bit 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 4.4 (row `bt-dis`). We denote the performance in values of precision and recall, where

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

We examined the binary with the lowest precision (i.e., `xvminitoppm` with 84.40%), 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,

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%

TABLE 4.4: Precision and recall of approxis.

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. Dispatching a binary stream 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 Figure 4.3 could not be clearly disassembled with the tree from Figure 4.1. To detect those outliers and to extend approxis with other features, we integrate our results from Section 4.1. 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 Shah (2010) proposed a disassembler based on a set of logarithmic odds only, we argue that the decent 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 Section 4.1. We first focus on assisting our process of disassembling by resolving plausible instruction lengths. To summarize, 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 exorbitantly 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 falls below 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, since 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.

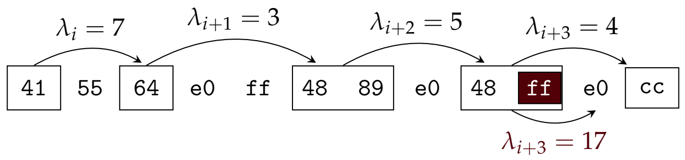


FIGURE 4.3: Selecting offsets with a predefined threshold $\tau = 16$.

Figure 4.3 illustrates the process of offset determination. We repeated the process of disassembling the set of 1318 64bit ELF binaries with assisted length-disassembling. The obtained results in Table 4.4 (row `bta-dis`) show an improvement in the case of precision.

CODE AND ARCHITECTURE DETECTION. Besides 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. A 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 Equation (4.2) as the average of all λ_i within a sliding window, with a predefined size $n+1$ 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.

$$\omega_x = \frac{\sum_{i=x}^{n+x} \lambda_i}{n} \tag{4.2}$$

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).

4.3 ASSESSMENT

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

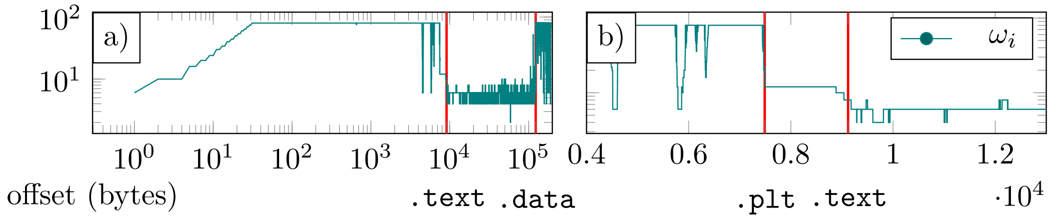


FIGURE 4.4: Approxis applied on `zip` (64bit); value of ω_i with cutoff set to 100;

arch	#files	#transition	detected
x86-64	400	1200	99 %
x86	392	1176	92 %

TABLE 4.5: Ratio of correctly detected transitions.

the possibility of determining 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.

4.3.1 *Code Detection*

The following evaluation addresses our defined requirement of *robustness*. To evaluate the code detection performance in the field of binary analysis, we first examined a randomly selected ELF binary. The result in Figure 4.4 illustrates the capabilities of approxis to differentiate code from data. Figure 4.4-a shows the initial reduction of confidence by the header. Figure 4.4-b shows that the `.text` section is clearly distinguishable and introduced by the `.plt` section, which is not filled with common sequences of instructions.

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 $\kappa + 1$ preceding and $\kappa + 1$ 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 Equation (4.3)). 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 4.5.

$$\tau_c = \tau_d = \begin{cases} 1, & \text{if } |\frac{\sum_{i=\theta-\kappa}^{\theta} \lambda_i}{n} - \frac{\sum_{i=\theta}^{\theta+\kappa} \lambda_i}{n}| > \delta \\ 0, & \text{otherwise} \end{cases} \tag{4.3}$$

4.3.2 *Architecture Detection*

The following evaluation addresses our defined requirement of *versatility*. 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`

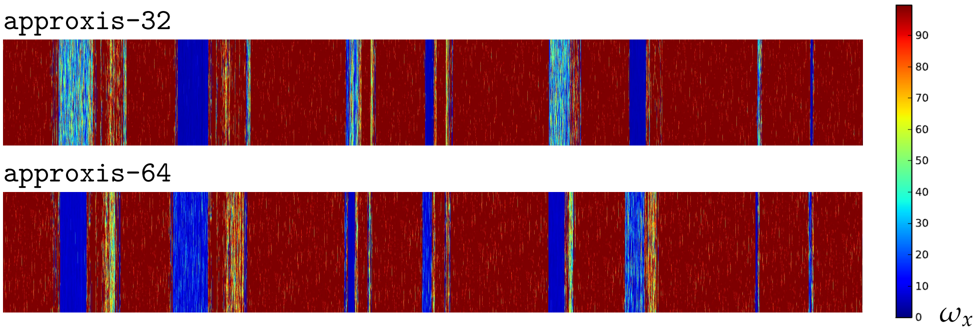


FIGURE 4.5: Comparison of code detection for x86 and x86-64 binaries.

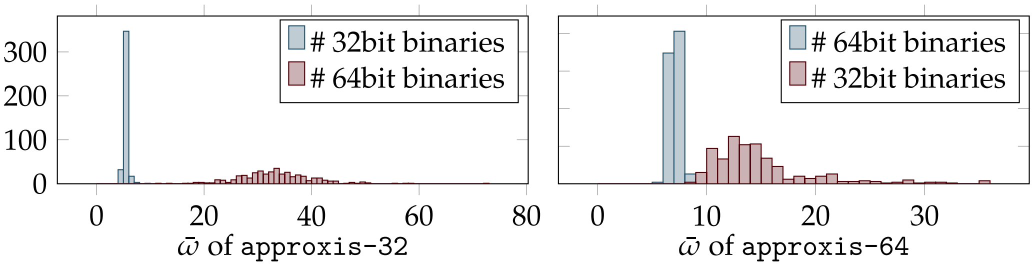


FIGURE 4.6: Architecture detection of approxis with a selected bin size of one.

and `cut`. As already mentioned 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 Figure 4.5-a / 4.5-b). As shown, the implementation is able to differentiate and align the embedded binary blobs within the image. Sections containing executable code, i.e., instruction byte patterns, are visualized blue.

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 64bit and 32bit 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 Figure 4.6 and outlines the capabilities of approxis to discriminate a present architecture.

4.3.3 *Computational Performance*

The following evaluation addresses our defined requirement of *speed*. The execution time of approxis was tested on a machine with an `Intel(R) Core(TM) i5-3570K CPU @ 3.40GHz` 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 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 does not 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 Section 4.2, the comparison of approxis with other disassemblers is somewhat misleading. As approxis extends 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 with competing approaches. We applied

Execution time				Description
approxis		distorm		disassembler
32	64	32	64	mode
29.084s	21.936s	1m20.770s	1m7.772s	Concatenated set of 64bit binaries from `/usr/bin`
27.859s	31.918s	1m43.999s	1m43.046s	Raw memory dump acquired with `LiME`[2]
1m15.521s	1m44.990s	1m58.278s	1m56.192s	Random sequences of bytes generated with `/dev/urandom`

TABLE 4.6: Execution time of approxis and distorm with different input data.

each disassembler in different modes and optimized our implementation of the distorm engine by removing unnecessary printouts and buffers. Table 4.6 outlines that the execution time of approxis relies on the processed input.

4.4 DISCUSSION

In this chapter, we demonstrated a first approach to detect, discriminate and approximate disassemble code fragments within a 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 also shows a good computational speed.

In Chapter 5 we will further discuss the integration of approxis into an existing Approximate Matching scheme.

5

INTERFACING CTPH - MRSH-MEM

5.1 INTRODUCTION

In this chapter we present a novel approach that allows detecting similarities between software stored on hard drives and loaded as modules into memory (Linux only). As presented in the previous chapter, we rely on approxis (Liebler and Baier, 2017), an approximate disassembler for performing carving of code-related structures. To compare the content of the dumps with the content of hard drives, we borrowed concepts from a subdomain of digital forensics called Approximate Matching. In a nutshell, these algorithms can be used to find similarities between different digital objects (e.g., compare the similarity between two text documents). We consider our approach as robust for memory-based carving of code-related fragments, as our implementation relies on the possibility of scattered code structures itself. Thus, our approach does not depend on critical system-related structures, the manual adaptation of signatures or the specification of any alignment properties. Using Approximate Matching for memory forensics is not new and was already discussed where most researchers questioned the applicability and runtime efficiency of those algorithms (Cohen, 2017; White, Schatz, and Foo, 2013; Ligh et al., 2014). We discuss the application of Approximate Matching in the scope of memory carving and release a prototype implementation which shows good computational performance. To the best of our knowledge, this is the first usable implementation of an Approximate Matching technique, which integrates an additional step of code carving to this degree.

In this chapter we will discuss multiple contributions.

- We interface Approximate Matching with an additional layer of approximate disassembling to process physical memory which is accomplished by integrating approxis into mrsh-net.
- We demonstrate the capabilities of our approach to identify code structures in large amounts of raw data by the extraction of allocable code sections from different resources (e.g., online repositories or hard disk images).
- We demonstrate an acceptable runtime performance for processing memory dumps with a reasonable and realistic size.
- Besides our prototype implementation, we demonstrate a first application to identify kernel structures in memory, i.e., we profile the current running Kernel version inside a previously acquired raw memory dump.

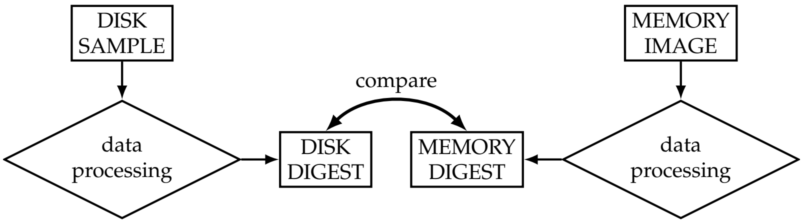


FIGURE 5.1: Overview of the overall approach and application of mrsh-mem

- Lastly, we show the detection of code fragments of a running process in user memory space.

As previously mentioned, this chapter presents mrsh-mem which is a combination of approxis and the Approximate Matching algorithm mrsh-net. The main goal is the possible comparison of memory images and hard-drive content, e.g., applications installed on the system and currently executed in memory. This will allow investigators to profile parts of the memory dump (whitelisting) or detect suspicious code patterns inside the memory dump (blacklisting). The proposed approach therefore enables the performance of robust unstructured analysis of a memory dump. A strongly generalized overview of our use case is shown in Figure 5.1. The left-hand side outlines the already existing Approximate Matching techniques. The opposite side is the content of this chapter, i.e., how to modify / normalize a memory image so that we can generate a fingerprint / memory digest which then can be compared against a traditional Approximate Matching fingerprint.

DOMAIN-SPECIFIC CONSIDERATIONS. Before discussing our approach, we highlight some considerations which impacted our design decisions. With respect to concepts of Virtual Memory Management, Approximate Matching and approximate disassembling, the following listing concludes some considerations, which have to be respected for processing raw physical memory:

- **Mappings of virtually contiguous regions do not have to be physically contiguous.** The fact that pages of a specific context do not have to be allocated contiguously in memory is an important issue for the overall concept of transferring Approximate Matching to the field of memory forensics. Found features should be considered in a page-sized scope. This should lead to future research and concepts of composing separated page sizes.

- **The page size can vary for different architectures and pages are aligned to its page size in memory.** We expect the page size to be at least 4 KB which is the most common page size. This is important when selecting the block size b as it should be smaller than the page size. Explanation: a large b will reduce the amount of chunks within a page boundary of a physical memory dump. Considering non-contiguous physical pages, this could lead to producing features that frequently overlap with adjacent pages (all details about b are discussed in Section 5.2.2).

- **Pages could be shared between processes, thus a physical page could refer to multiple virtual pages but not the other way round. The concept is also called *Shared Memory*.** This concept outlines, that virtual to physical mapping is actually not a one-to-one mapping. Especially in the case of shared libraries, it should be clear that those matches could not actually resolve a specific sample. This problem overlaps with the problem of handling *common blocks*, already introduced in the case of general Approximate Matching challenges and different database lookup strategies.

- **Each process context uses its own virtual mapping. We are not able to actually resolve the physical offset of a virtual address without translation or the analysis of system related structures.** Our overall approach of processing is context unaware. Thus, we are not able to actually resolve a virtual reference. It should be clear that examination on a higher level, e.g. the usage of recursive traversal or control flow graph analysis, are not applicable in a context-unaware scenario.

- **Not all requested pages of a process are allocated immediately. The concept is also called *Lazy Allocation*.** In contrast to traditional hard drive forensics, the looked-up data sample has not to be present in RAM completely during acquisition. We should consider this fact during examination of found chunks inside a target image.

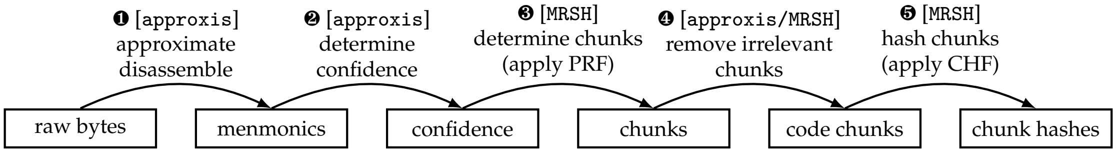
- **In case of high memory usage, the kernel is able to swap content to hard disk, which is denoted as *Swapping*.** Similar to *Lazy Allocations*, this concept should raise our awareness, that we should not expect a sample (i.e. in the case of loaded executables) which is fully loaded to memory at the time of acquisition.

- **We expect systems with 8 GB RAM to represent a reasonable upper limit for current consumer PC systems.** Similar to previous publications, which have determined the required Bloom filter size for their field of application, we should determine the maximum required size for storing memory dumps. Even if it is common that most of the acquired memory should be initialized with zero byte paddings, we assume this value as the possible upper limit, in case the acquired memory is well populated.

- **Executables are changed during the process of loading to memory.** Code on a hard disk differs from its representation in memory. Legitimate changes to code would obviously cause the original fuzzy hashing techniques to fail. The PRF and CHF would possibly interpret even legitimate updates to the code structures. The possible pitfalls of applying traditional fuzzy hashes are twofold. First, the process of chunk extraction could be disturbed, as the PRF could trigger at different offsets. Second, the hash value of a extracted chunk could differ, as the CHF works on a byte-level of not normalized code fragments.

5.2 INTERFACING APPROXIMATE MATCHING - MRSH-V2

In the following we will describe how we combined approxis with Approximate Matching. The workflow of data decoding and examination is depicted in Figure 5.2 and can be described as a multi-layered process. Note, even though the figure shows a clear separation between different steps, most of them are strongly interleaved and therefore it is hard to visualize the exact flow. A description of the steps depicted in the figure is given in the following (step one and two are nearly unmodified steps presented by Liebler and Baier, 2017):



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

FIGURE 5.2: Overview of the data processing steps. The process outlines the interleaved characteristics of the overall approach. We highlighted integrated components of approxis and mrsh.

❶ The raw bytes from the memory are disassembled using approxis which will return the mnemonic as well as the length of the instruction. The decoded mnemonic is especially important for further proceedings as the process of chunk extraction and chunk hashing.

❷ Using the confidence score λ produced by approxis and the concept of a simple running length counter allows differentiation between code and data. The running length counter counts repeating mnemonics, e.g., a nop-slide, which should not be considered. Note, for our approach we will focus on code and neglect data.

❸ Having the approximate disassembled code, we now identify the chunk boundaries based on the mnemonics. Therefore, we utilize a sliding window approach on the mnemonics (precisely, the rolling hash runs over a C-buffer that contains the byte representations of the mnemonics). All details are provided in Sec. 5.2.1.

❹ After identifying all chunks, an additional filter is applied to remove irrelevant chunks. To identify relevant chunks, we utilize the confidence score. For instance, the first three entries from chunk one (indicated by C_1) have a high confidence score (64, 64, 63), therefore we consider this chunk as not relevant (indicated by '[0]').

❺ Lastly, the relevant chunks (indicated by '[1]') will be hashed and stored into a database. While this example focused on creating a chunk-hash based on the mnemonic buffer, we can utilize other buffers as well for further comparisons, e.g., the raw byte buffer.

5.2.1 *Implementation Details*

The previous section outlined a high level perspective of the procedure whereas this section provides specific details about our concept. As mentioned, we are using a

Step	buf_by buf_ro	buf_mn	PRF(buf_mn[-7:])	BF (Hash)
1.	by: … 00 00 00 00 ★ e8 00 00 00 00 83 2d 00 00 00 00 01 74 02 f3 c3 … ro: 5	mn: 114	PRF(114) → ✗	[]
2.	by: … 00 00 00 00 ★ 83 2d 00 00 00 00 01 74 02 f3 c3 e8 00 00 00 00 … ro: 5 7	mn: 114 91	PRF(114 91) → ✗	[]
3.	by: … 00 00 00 01 ★ 74 02 f3 c3 e8 00 00 00 00 e9 00 00 00 00 66 0f … ro: 5 7 2	mn: 114 91 44	PRF(114 91 44) → ✗	[]
4.	by: … 00 01 74 02 ★ f3 c3 e8 00 00 00 00 e9 00 00 00 00 66 0f 1f 44 … ro: 5 7 2 2	mn: 114 91 44 330	PRF(114 91 44 330) → ✗	[]
5.	by: … 74 02 f3 c3 ★ e8 00 00 00 00 e9 00 00 00 00 66 0f 1f 44 00 00 … ro: 5 7 2 2 5	mn: 114 91 44 330 114	PRF(114 91 44 330 114) → ✓	['c42']
6.	by: … 00 00 00 00 ★ e9 00 00 00 00 66 0f 1f 44 00 00 e8 00 00 00 00 … ro: 5	mn: 115	PRF(114 91 44 330 114 115) → ✗	['c42']
7.	by: … 00 00 00 00 ★ 66 0f 1f 44 00 00 e8 00 00 00 00 48 83 ec 70 48 … ro: 5 6	mn: 115 14	PRF(114 91 44 330 114 115 14) → ✗	['c42']
8.	by: … 1f 44 00 00 ★ e8 00 00 00 00 48 83 ec 70 48 89 e7 e8 00 00 00 … ro: 5 6 5	mn: 115 14 114	PRF(91 44 330 114 115 14 114) → ✓	['c42', '2b4']
9.	by: … 00 00 00 00 ★ 48 83 ec 70 48 89 e7 e8 00 00 00 00 48 8b 54 24 … ro: 4	mn: 91	PRF(44 330 114 115 14 114 91) → ✗	['c42', '2b4']
10.	by: … 48 83 ec 70 ★ 48 89 e7 e8 00 00 00 00 48 8b 54 24 20 48 2b 54 … ro: 4 3	mn: 91 95	PRF(330 114 115 14 114 91 95) → ✗	['c42', '2b4']

FIGURE 5.3: Example of the overall processing pass with different buffers of the raw buffer (by), the buffer of decoded offsets (ro) and the decoded mnemonics (mn). The current decoded offset is denoted with ★ and shifted by the amount ro after each step.

multi-layered process which is reflected by the usage of multiple buffers. Most of the working buffers are limited in their size and thus, have to be swapped during processing (i.e. buf_lo, buf_ro, buf_pe, buf_mn). We skip the details of the buffer swapping for simplicity, but recommend not considering the implementation as multiple circular buffers. For the prototype (and hence for the runtime performance evaluation) we expect that the input stream (i.e. buf_by) can be stored in memory completely. For a better understanding of the overall processing and the usage of the mentioned buffers, we explain the procedure based on a comprehensive example in the following paragraph.

Figure 5.3 gives an example of how the different buffers are utilized. The example shows ten steps of processing, where in each step an instruction is decoded from the byte buffer by (buf_by) at the highlighted offset (⋆). The decoded instruction length and a corresponding mnemonic are saved into separate buffers after each offset in ro (buf_ro) and mn (buf_mn), respectively. Thus, the current buffers steadily increase with each decoded offset. For instance, in step 1, the first seven bytes in the byte buffer are 0xe8 00 00 00 00 83 2d. The disassembler decodes the first five bytes to the mnemonic 114, which represents a call instruction. The pointer moves to the next offset and repeats the decoding process similar to the first row. Thus, in step 2, the next byte sequence 0x83 2d 00 00 00 00 01 gets decoded to the mnemonic 91. Beside the mnemonic, we again store the corresponding length of the instruction seven in the buffer ro. In the third row, we can see that the previous pointer was increased by seven and the process repeats with the decoded instruction bytes 0x74 02.

Once the mnemonics are decoded, they are 'added' to our sliding window, e.g., in step 1 the mnemonic 114 is the first entry of the rolling hash. Given that the rolling hash has a length of seven, the last seven mnemonics serve as input for the

PRF. As the amount of mnemonic representatives is larger than 2^8, the hash value is calculated over a sequence of integers, which stores the mnemonic representatives decoded by approxis. As soon as the PRF determines a chunk boundary (see the output of the PRF in steps 5 and 8 of Figure 5.3), the decoded instructions (`buf_mn`) are hashed using the chunk hash function (CHF) `FNV-1a` (Fowler et al., 2011). The hash value is then stored into the Bloom filter, where the hash value is separated into $k = 6$ sub-hashes and each sets a bit of the Bloom filter. The procedure is identical to the original mrsh approach. In Figure 5.3, we represent the hash values by a shortened representation in the last column denoted as `BF`. The buffers `by` and `ro` are cleared out after each chunk extraction (see steps 6 and 9 in Figure 5.3), where the sliding window of the applied PRF keeps the buffer of the last seven mnemonics for the next decoding pass.

To filter code related chunks and to reduce the overall amount of Bloom filter inserts, we utilize the introduced value of confidence for consecutive instructions (see Chapter 4). Therefore, chunks are inserted into a Bloom filter as soon as they fulfill two properties. First, the chunk size has to contain at least ten consecutive instructions. Second, the overall amount of considerable meaningful mnemonic bigrams has to be at least 30 % for a current chunk.

Hashing the decoded byte sequences (`buf_mn`) will, for example, neglect all byte sequences which represent operand information within an instruction. More importantly, different opcodes on a byte level can be mapped to the same mnemonic representative. Besides hashing the decoded instructions, one could also hash the other buffers. In this prototype, we actually propose the hashing of two buffers, the decoded buffer of representatives (`buf_mn`) and the original input buffer (`buf_by`). This empowers an investigator to detect similar instruction sequences and inspect possible deviations on a byte level.

We summarize two central adaptations to the original mrsh and approxis implementation. In contrast to previous bytewise Approximate Matching approaches, the chunk boundaries are defined with the help of the *decoded byte sequences*, not the byte sequences themselves. Additionally, the code detection is not performed within a fixed-sized sliding window.

5.2.2 *Configurable Parameters*

An overview of the important configurable parameters is given in the following subsection. We additionally give a short explanation and reasoning of the parameters and the selected default values. We will first describe the important parameters which define the overall chunk extraction process. An overview of the parameters can be seen in Table 5.1.

SELECTING THE FEATURE (CHUNK) SIZE (b). As already introduced, the PRF approximately defines the extracted chunk sizes. Considering the minimum respected page size of 4KiB and the presence of non-contiguous memory mappings, we depict a default value of $b = 64$ with the defined parameter name `BLOCK_SIZE`.

CODE CONFIDENCE. The process of filtering chunks, which store code fragments, could be controlled by two parameters. The parameter `CODE_THRESH` describes the maximum value of λ which defines two consecutive instructions to be meaningful or not. The values of λ are stored within the buffer `buf_lo` during a decoding pass. If less than ten consecutive instructions are detected within a chunk, the chunk will

Parameter	Range	Default	Description
BLOCK_SIZE	/	64	Defined modulus and approximated size of a chunk (b).
CODE_THRESH	[0-100]	30	The value defines the threshold of code confidence. A sequence of instructions should be considered as code fragments, as soon as the value of confidence is lower than the defined CODE_THRESH.
CODE_COV	[0-1]	0.3	Defines the minimum required percentage of code coverage within a chunk, before it gets hashed and inserted into the Bloom filter.

TABLE 5.1: Parameters of the mrsh-mem implementation.

Parameter	Range	Default	Description
RLE_THRESH	[0-100]	10	Sets the threshold when the repeating sequences of instructions should be considered as not valid.
RLE_DRAIN	[0.1-1.0]	0.9	The value defines a factor, which lowers the running length counter significantly more quickly. If the value is lower than the defined threshold, we switch to stepwise decrementing the counter.

TABLE 5.2: Parameters of the mrsh-mem implementation for controlling the running length penalty.

not be inserted into the Bloom filter. Additionally, considering large chunks, we measure the total amount of instructions within a chunk. We require at least 30 % of the decoded instructions inside a chunk, before the chunk will be inserted into the Bloom filter. We denote this threshold of code coverage as parameter CODE_COV.

PENALTIES. The original implementation of approxis considers a large amount of repeating decoded mnemonics as less meaningful. An example could be misleadingly decoded sequences of non-allocated zero bytes or other padding instructions (e.g., NOP instructions). The running length counter of approxis counts subsequent similar decoded mnemonics. As soon as the running length counter instruction exceeds RLE_THRESH, a penalty is written into the buffer buf_pe. The saved penalty is added to the value of confidence stored in the buffer buf_lo afterwards. Besides the threshold, we additionally configure a factor, which decreases the current running length after a sequence of similar mnemonics was interrupted. We increase the RLE_DRAIN to respect the non-contiguous properties of physical memory dumps. Both parameters are summarized in Table 5.2.

DETERMINING THE BLOOM FILTER SIZE (m). In Section 5.1 we described the idiosyncrasies and properties of memory management. In this paragraph we explain the parameter adaptations and the following impacts on the needed Bloom filter size. For further details of the following formulas we refer to Breitinger, Baier, and White (2014). We consider 8 GiB as reasonable RAM size and select the expected input size (s) to be $s = 8$ GiB.

With the expectation that a modulus b defines a trigger point and thus the probability of a hit is reciprocally proportional to the average chunk size, we estimate the number of extracted chunks n for a given input image with size s. The calculation

Parameter	Range	Default	Description
BF_SIZE_IN_BYTES	/	128 MiB	Size of the Bloom filter (b, must be a power of two).
SUBHASHES	[5,7]	6	Number of used subhashes (k).
MIN_RUN	/	6	Minimum amount of correctly to be identified consecutive features (r).

TABLE 5.3: Parameters of the mrsh-mem Bloom filter implementation.

can be seen in Equation (5.1).

$$n = \frac{s \cdot 2^{30}}{b} = \frac{8 \cdot 2^{30}}{64} = 134,217,728 \tag{5.1}$$

In Breitinger, Baier, and White (2014) the authors mention that the choice of k is limited by the used FNV-1a hash function. Thus, the value of k is limited to $5 \leq k \leq 7$. Similar to mrsh-net we choose the value of k to be $k = 6$. A single Bloom filter of size 32 MiB could be used to monitor approximately 2 GiB of data, whereas as Bloom filter of size 2 GiB could approximately monitor 100 GiB of data (Breitinger and Baggili, 2014). Obviously, the filter has to be stored in memory during examination. Similar to Breitinger and Baggili (2014) we consider this size as still manageable even on casual or mobile systems. To determine the maximum needed size of the Bloom filter in dependency on the expected input size, we depict the corresponding formula from Breitinger, Baier, and White, 2014. In addition, we set the parameter r, which defines the minimum amount of correctly-to-be-identified consecutive features, to be $r = 6$. Considering Equation (5.2) and the above-mentioned parameters, we propose a Bloom filter size of $m \approx 7.0426 \cdot 10^8$ bits $\approx$ 84 MiB. An overview of the configurable parameters can be seen in Table 5.3.

$$m = -\frac{k \cdot n}{\ln(1 - \sqrt[k \cdot r]{p})}, \text{ where } p = (1 - e^{-kn/m})^k. \tag{5.2}$$

5.3 APPLICATION

The implementation of mrsh-net uses a single, large Bloom filter which reveals two notable disadvantages: memory consumption and the lack of file identification, i.e., the approach can only answer the question of whether a file is contained in a given Bloom filter, but we cannot say to which file a similarity exists. However, we decided to use this approach for realizing the prototype; we will evaluate possible strategies in future research to match chunks with a given file base.

The adaptation and integration of a single Bloom Filter (BF) gives us a good computational performance for initial white- or blacklisting of extracted chunks. However, to achieve better identification, we additionally create a database of extracted Chunk Hash Values (CHV). The current Chunk Hash Database (CHDB) consists of a single, large lookup tree, which stores all chunk hash values with a corresponding file name inside each leaf node. As we focus on computational speed and expect better solutions for a fast file identification, we do not consider the database in the case of runtime performance analysis or memory consumption. Figure 5.4 provides an overview of the general application. First, an investigator has to acquire the dumps

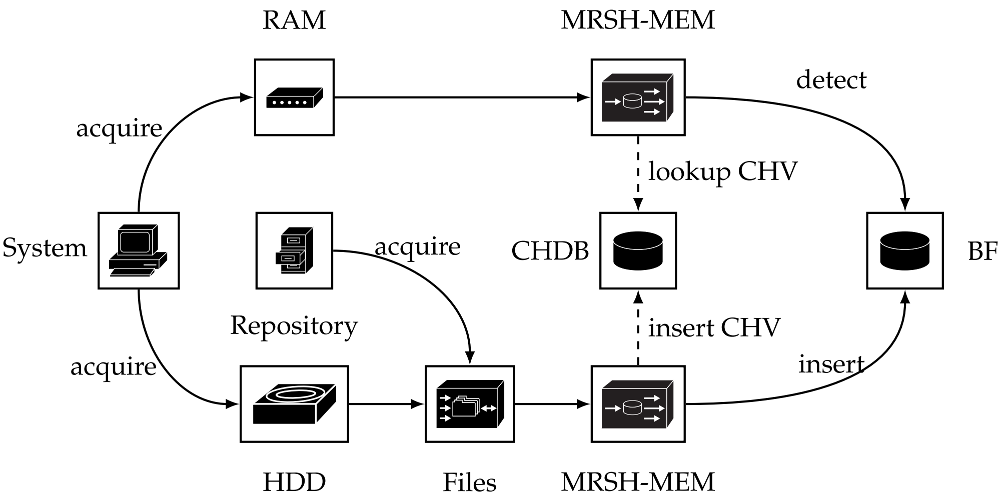


FIGURE 5.4: Overview of the application of mrsh-mem.

(memory and hard disk). Additionally, the acquisition of files from different repositories can be considered. All input files are processed with mrsh-mem and stored in the Bloom filter as well as in a database of known code fragments (CHDB). Applying mrsh-mem on the acquired memory dump will then answer the question of whether a particular memory fragment is found in the BF. The comparison against the database will allow answering the question of which file was matched.

Different versions of the same executable can share the same code base. Thus, similar chunk hash values can occur, which will be inserted into the Bloom filter digest. Leaf nodes in our CHDB, which are occupied by chunks of multiple versions of an executable (e.g., the same chunks have been extracted for multiple versions of a file) are denoted in the following plot as *multiple* hits. Chunk hash values which only appeared for a single version are marked as *single* hits.

For testing purposes, we acquired memory and hard disk fragments from an existing Debian 8 installation, which was originally setup inside a virtualized environment for common network analysis tasks. In detail, we inspected a Debian 8 installation (Debian 3.16.7 x86_64 GNU/Linux) running with the help of Virtual Box (Version 5.2.6 r120293). The system contains several real world applications and was used for several weeks without a reboot. To acquire the memory we used LiME[1] (Linux Memory Extractor) which is a Loadable Kernel Module for memory acquisition. We inserted the module into the running Kernel and acquired 2 GiB of the memory in raw format.

5.3.1 *Identify Present Linux Kernel Version*

In our first application we identify the presence of Kernel fragments and the Kernel version of the target system by analyzing the acquired raw memory dump with mrsh-mem. This process can support further structured analysis and possibly enhance the task of profile determination. In our application we utilize a set of available Linux images of a public Debian repository[2]. The Kernel files (i.e., vmlinux/vmlinuz) have been obtained by the corresponding deb-Packages, the .text sections

[1] https://github.com/504ensicsLabs/LiME (last access 2021-08-01).
[2] http://ftp.us.debian.org/debian/pool/main/l/linux/ (last access 2021-08-01).

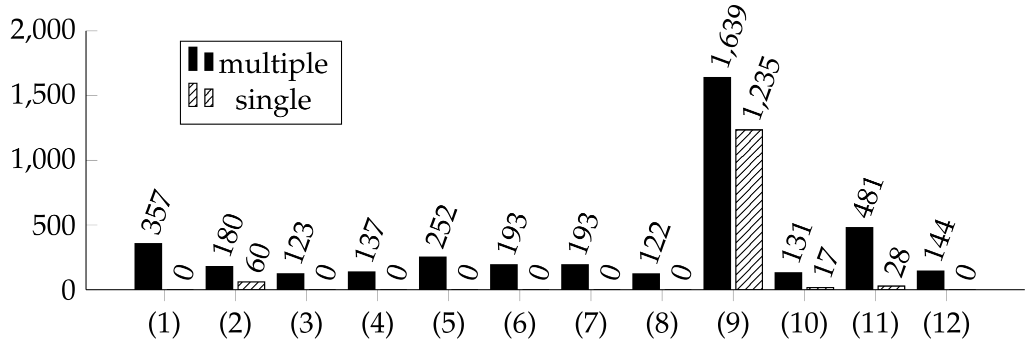


FIGURE 5.5: The detected chunk sequences and the overall counts for each Kernel version. As can be seen, the present Kernel version of our target system, i.e. 3.16.0-4-amd64 (9), shows a significant amount of detected chunks.

ID	Kernel	ID	Kernel	ID	Kernel
1)	3.2.0-4-amd64	2)	4.13.0-0.bpo.1-amd64	3)	4.14.0-0.bpo.2-rt-amd64
4)	4.14.0-0.bpo.3-amd64	5)	3.2.0-4-rt-amd64	6)	4.14.0-3-amd64
7)	4.15.0-rc8-amd64	8)	4.14.0-0.bpo.2-amd64	9)	**3.16.0-4-amd64**
10)	4.14.0-3-rt-amd64	11)	3.16.0-0.bpo.4-amd64	12)	4.14.0-0.bpo.3-rt-amd64

TABLE 5.4: Extracted Linux Kernel images from the Debian repository (marked with an identifier). The actual present Kernel in the extracted memory image is highlighted (9).

have been extracted and the images have been processed with mrsh-mem. We additionally store the extracted Linux Kernels and their corresponding chunk hash values in our introduced CHDB. Subsequently, we query the CHDB with 12 different Kernel images (see Table 5.4 for an overview of all inserted Linux Kernels).

While we expect that most of the Linux Kernels from the repository share a reasonable amount of similar code chunks, this can obviously vary for different versions. To determine the actual Kernel version of our target system, we analyzed the detected chunks in two ways. First, we determined the total amount of detected chunks for each processed Kernel version. Second, we examined those chunks which are only mapped to a *single* Kernel version by the CHDB and do not share *multiple* of those chunks with other Kernel versions.

After performing the step of chunk identification with mrsh-mem, we additionally identified the related Kernel version(s) for each chunk. The amount and distribution of detected chunks by its corresponding kernel version(s) can be seen in Figure 5.5. The statically linked Kernel images share a reasonable amount of similar code fragments (bar *multiple*). However, the actual Kernel version clearly occupies most of the extracted chunks and thus we could distinguish the present Kernel from the other images (see column (9) in Figure 5.5). Next, we only considered chunks which are mapped to a single Linux Kernel and do not count shared code fragments between different versions, i.e., we filter out identified chunks which are related to multiple Kernels. The examination of distinct mapped chunks in Figure 5.5 (bar *single*) underlines the presence of our expected Kernel version (vmlinuz-3.16.0-4-amd64).

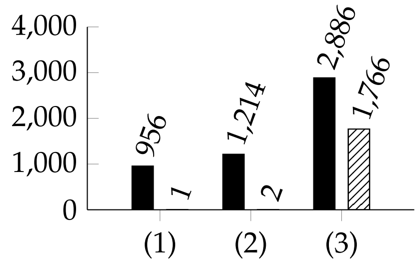


FIGURE 5.6: Examination of a memory dump of our target system while Wireshark was running (ELF executable amd64; version 1.12.1).

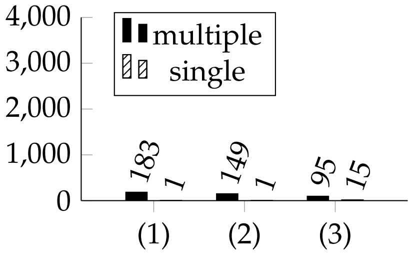


FIGURE 5.7: Memory dump of our target system after rebooting the virtual machine and thus, without a running Wireshark instance.

CONSIDERATIONS. Discussing the examination of the Kernel `.text` section in memory leads to the question of whether mrsh-mem can be used for detecting advanced Kernel infection techniques. Different hijacking techniques should lead to the presences of modifications in the memory-located version of the original Kernel. However, the process of Kernel loading is quiet complex and the Linux Kernel binaries could additionally contain modification instructions, e.g., so-called alternative instructions (`.altinstructions`[3]). Those instructions patch the original code during loading. At this point we leave the question of whether mrsh-mem is usable for advanced code integrity checks of Linux Kernels unanswered for further research.

5.3.2 *Identify Application in User Memory*

Kernel memory mappings should be considered contiguous in most of the cases. To determine the capabilities in user space memory, we performed a task of process and application identification. We inspected the raw memory dump on the presences of application-related code fragments. In detail, we acquired three different versions of the Wireshark Protocol Analyzer[4] from a Debian repository[5] (see Table 5.5). The acquired ELFs were dynamically linked and stripped. We extracted the allocable `.text` sections of the different executables and processed them with mrsh-mem, where each executable stored approximately 4130 chunks. The chunks were also inserted into the CHDB for the evaluation of *single* and *multiple* hits.

We ensured that an instance of Wireshark 1.12.1 was running at the time of memory acquisition. Figure 5.6 illustrates the capabilities of detecting and discriminating a running (or formerly running) application in memory. The amount of single occupied chunks (1766) clearly identifies the actual running Wireshark version (1.12.1).

To investigate possible false positives and to examine the discrimination between a running and not-running process we repeated the procedure after rebooting the system. Thus, we were not expecting to find presence of Wireshark. The results are shown in Figure 5.7 and the plot indicates very low numbers / matches. Precisely, the bars show some hits in the case of multiple-occupied chunks. To lower the values of false positives, we propose the adaptation and increase of the `MIN_RUN` parameter. We additional suggest a minimum required chunk size, as most of the false positives were smaller than 40 bytes.

[3] https://lwn.net/Articles/531148/ (last access 2021-08-01).

[4] https://www.wireshark.org/ (last access 2021-08-01).

[5] http://ftp.us.debian.org/debian/pool/main/w/wireshark/ (last access 2021-08-01).

ID	Version
(1)	2.4.4-1_amd64
(2)	2.2.6+g32dac6a-2+deb9u2_amd64
(3)	**1.12.1+g01b65bf-4+deb8u13_amd64**

Table 5.5: List of extracted Wireshark versions. The actual running version is highlighted (3).

Execution time		Chunks	Description
insert	lookup		
46.0s	48.0s	6,888k	Concatenated set of 64bit binaries from `/usr/bin`
50.0s	50.0s	1,609k	Raw memory dump acquired with `LiME`
197.0s	192.0s	10,538k	Random byte sequences, generated with `/dev/urandom`

Table 5.6: Insert and lookup runtime performance of mrsh-mem for different input images.

5.3.3 *Runtime Performance*

In the following paragraph we examine the runtime efficiency of mrsh-mem. In detail, we measured the runtime for disassembling, chunk extraction, chunk hashing and Bloom filter handling. Note, we differentiate between Bloom filter creation and Bloom filter lookup. As mentioned, the processed byte sequences can significantly influence the overall disassembling performance. Therefore, we study the runtime performance for three different images: a concatenated set of 64 bit ELF binaries, a raw memory dump acquired with LiME and a random sequence of bytes. Lastly, we removed all unnecessary functionalities (e.g., printout mechanisms) and compiled our binary with an optimization set to `O2`[6].

The runtime efficiency test was performed on a Lenovo Thinkpad x250 with an Intel Core i5 2x 2,2 GHz and 8 GB RAM. The performance of the built-in Solid State Drive was also determined, where the read performance was 508 MB/s and the write performance was 513 MB/s. The overall results are shown in Table 5.6. The column of chunks defines the amount of triggered chunk boundaries for each image and for one pass. Similar to our previous evaluation of approxis shown in Table 4.6, the runtime decreases with an increasing amount of randomized and thus, uncertain dispatching cycles. In other words, approxis has to repeatedly interpret the same byte sequences at a given offset until the offset is finally dispatched.

5.4 SUMMARY

Approximate Matching techniques are known among the digital forensics community and have been utilized in different fields of application. Current implementations empower whitelisting or identifying fragments of data in the field of classical disk or network forensics. The application of Approximate Matching on memory reveals several pitfalls. Similar to other unstructured analysis techniques, our introduced approach has to consider several idiosyncrasies of inspecting the physical memory space. Therefore, we first discussed those considerations and limitations.

We introduced a new specimen mrsh-mem by interfacing Approximate Matching (mrsh-net) with an additional step of approximate disassembling (approxis). We described the implementation details of merging both techniques as well as the

[6] `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html` (last access 2021-08-01).

needed adaptations and changed parameter settings. The integration of an additional step of disassembling stabilizes the original bytewise application, as our approach works on disassembled and thus normalized instruction sequences. To the best of our knowledge, this is the first implementation which interfaces Approximate Matching with an additional step of approximate disassembling.

We showed the feasibility of our approach by comparing a memory dump against code fragments gained from different resources, i.e., code extracted from a hard disk as well as a repository. Our introduced approach detects allocable code fragments without the need of manual inspecting an executable or the manual definition of any matching rules. Our first prototype empowers easily creating a database of different applications and perform the examination of raw memory dumps. As former publications claimed Approximate Matching to be slow, we showed that our current prototype achieves a good computational performance, without the usage of any parallelization.

FUTURE WORK. The current implementation of the chunk hash database (CHDB) and the usage of a single Bloom filter shows promising results. However, an advanced database should be considered, which empowers actually identifying and naming a specific fragment found in memory. A general discussion of this task was already started by Garfinkel and McCarrin (2015) and the authors showed major challenges caused by shared blocks across different samples of a specific file type, e.g., office documents. As we process executable sections of code, different results should be expected which require the reassessment of a chunk-based file identification in our specific field of application. A promising candidate for further developments could be the adaptation and integration of Hierarchical Bloom Filter Trees (HBFT) (Lillis, Breitinger, and Scanlon, 2017) or the utilization of already introduced hash databases (Young et al., 2012). We will further discuss this topic in Chapter 7.

In addition, the process of chunk hashing should be further investigated. The extraction of chunks based on mnemonic representatives or the original byte sequences themselves, could be further extended by its intermediate representation. Thus, the definition and extraction of code-related chunks could be improved. We will discuss this topic in Chapters 6 and 8.

6

LINEAR FUNCTION DETECTION

6.1 INTRODUCTION

In this chapter we address the previously mentioned problem of extracting reliable chunks of the processed input stream. In detail, we inspect and reassess different concepts of function boundary detection as a replacement of existing pseudorandom functions (PRF) utilized for chunk extraction of CTPH schemes.

We first give a short introduction to the problem of function detection and describe the already introduced condition of *linearity* (Section 6.1.1). For a more detailed and formal explanation of the task of function detection, we refer to previous work (Bao et al., 2014; Shin, Song, and Moazzezi, 2015; Andriesse, Slowinska, and Bos, 2017). The already introduced approaches in Section 2.2.3 are shortly discussed and categorized. We depict two applicable approaches for signature-based function detection and discuss their functionality: Recurrent Neural Network (see Section 6.1.2) and Weighted Prefix Tree (see Section 6.1.3). In contrast to recent publications, we consider runtime performance as an important constraint. We outline the concrete adaptations to the models and discuss the utilization of mnemonics only. This reduces the overall feature size and improves the runtime performance of the classification. Our analysis underlines the applicability by reaching those goals even with significant reduced feature vectors.

In Section 2.2 a detailed overview of different available and appropriate datasets is given. Because of the outlined reasons, the following section and its evaluation is based on the ground truth dataset proposed by Andriesse et al. (2016).

6.1.1 *Linear Function Detection*

The task of function detection is one of the main disassembly challenges (Andriesse et al., 2016). The problem of function detection (e.g., with the help of static signatures) could be illustrated by the inspection of functions compiled with different optimization levels invoked to the compiler. The function structure and function prologue heavily changes due to optimization and compiler settings. An example can be seen in Figure 6.1, which was adopted from Shin, Song, and Moazzezi (2015). The example shows the remarkable impact of changing compiler flags. Most of the instructions in the function prologues of `mul_inv` heavily differ from each other. The process of function detection itself is most often divided into subtasks. We adhere to

LISTING 6.1: C-Source

```
int mul_inv(int a, int b) {
  int b0 = b, t, q;
  int x0 = 0, x1 = 1;
  if (b == 1) return 1;
  while (a > 1) {
    q=a/b, t=b, b=a%b, a=t;
    t=x0, x0=x1-q*x0, x1=t;
  }
  if (x1 < 0) x1 += b0;
  return x1;
}
```

LISTING 6.2: gcc-O0

```
<mul_inv>:
push %rbp
mov  %rsp,%rbp
mov  %edi,-0x24(%rbp)
mov  %esi,-0x28(%rbp)
mov  -0x28(%rbp),%eax
...
```

LISTING 6.3: gcc-O3

```
<mul_inv>:
cmp   $0x1,%esi
mov   %edi,%eax
je    400878
cmp   $0x1,%edi
jle   400878
mov   %esi,%ecx
mov   $0x1,%r8d
xor   %edi,%edi
...
```

FIGURE 6.1: Adopted example from (Shin, Song, and Moazzezi, 2015) with a function written in C and compiled with two different levels of optimization (gcc 4.9.1).

similar notation of previous work and refer to those publications (Andriesse, Slowinska, and Bos, 2017; Bao et al., 2014; Shin, Song, and Moazzezi, 2015).

The subtasks of function detection could be differentiated into *function start detection* and *function end detection*. The problem of *function boundary detection* is a superset of *function start* and *function end detection*. In the course of this work, we mainly focus on the task of function start detection. We borrowed most of the following definitions from Shin, Song, and Moazzezi, where C defines a given code base of a binary, which consists of several functions $f_1, \dots, f_n \in F$. A function $f_x \in F$ consists of a sequence of instructions I, with $i_y \in I$. The task of detection could be simplified into three basic tasks for a given target function f_t:

1. **Function start:** The first instruction of $i_s \in I$ of $f_t \in F$ within C.
2. **Function end:** The last instruction of $i_e \in I$ of a $f_t \in F$ within C.
3. **Function boundaries:** The tuple of instructions, which define the boundaries of a function, i.e., determine $(i_s, i_e) \in I^2$ of $f_t \in F$ within C.

The scope of function detection normally adheres to disciplines of analysing stripped binaries in the context of reverse engineering, malware analysis or code reuse detection. However, most of these applications are not limited in their form of application and are not subject to any time limits. As we transfer the function detection problem to the field of process context-unaware domains, we have to consider the constraints of *linear* processing. A concrete field of application is the examination of function boundaries in the domain of unstructured and context-unaware memory analysis. As previous publications showed fundamental improvements in the case of compiler- and architecture-agnostic techniques, we emphasize the motivation behind signature-related techniques, as we could not rely on features like binary lifting, intermediate representations or control flow analysis. Our goal is the integration of a function start identification approach, which works on an instruction buffer within a single sliding window pass. So we refine the problem of function detection to be a *linear function detection* problem. This leads to the evaluation of signature-based approaches.

Shown in Table 6.1, different approaches have been suggested. In this work we focus on signature-related (Sig.) and linear approaches (Lin.). The approach of Shin, Song, and Moazzezi (2015) is based on Bi-directional Networks, which are not applicable in our context. Those networks require the presence of a complete binary at

Approach	Sig.	Lin.	Comment
Guilfanov (2012)	✓	✓	Impractical variety Linux systems
Bao et al. (2014)	✓	✓	Weighted Prefix Trees applicable
Shin, Song, and Moazzezi (2015)	✓	-	Bi-directional RNN not applicable
Andriesse, Slowinska, and Bos (2017)	✗	✗	Process-context recreation needed
Potchik, Brian (2017) [1]	✗	✗	Process-context recreation needed

TABLE 6.1: Overview of different function detection approaches.

application time. The lookup of signature databases on Linux operating systems was already mentioned as impracticable. As we try to carve functions out of a memory, we could not rely on the recreation of a previously detected process.

6.1.2 *Recurrent Neural Networks (RNNs)*

Shin, Song, and Moazzezi (2015) provide a detailed overview of the different characteristics of Recurrent Neural Networks. Besides the work of Shin, Song, and Moazzezi (2015), we refer to Lipton, Berkowitz, and Elkan (2015) and Hochreiter and Schmidhuber (1997) for a detailed introduction. We outline differences between our approach and the approach of Shin, Song, and Moazzezi (2015) later.

The processing of sequences with Recurrent Neural Networks is a promising strategy for many different fields. In contrast to feedforward neural networks, the cells keep different states of previously processed input and consider sequences which have an explicit or implicit temporal relation. A sample x_t of a sequence is additionally labelled with its corresponding time step t of appearance, where a corresponding target y_t of labelled data also shares this temporal notation. The basic architecture of those nets could strongly vary for different fields of application. In Figure 6.2 a simplified and unfolded model is shown, which takes multiple input vectors over time and outputs a single vector after several time steps. The term unfolding denotes the unfolding of the cyclic characteristic, by displaying each timestep. Each of the edges in between the columns span adjacent time steps. The model in Figure 6.2 depicts a many to one relation.

LONG SHORT-TERM MEMORY (LSTM). Training Recurrent Neural Networks reveals several pitfalls, namely, the problems of vanishing and exploding gradients. There are different extensions of RNNs which try to consider those issues; one of those specimen are LSTM based Networks (Hochreiter and Schmidhuber, 1997; Gers, Schmidhuber, and Cummins, 1999). Those networks replace traditional nodes in the hidden layers with memory cells (see Figure 6.3). Each cell consists of an input node (g) and an internal state (c). The memory cells contain self-connected recurrent edges, to avoid vanishing or exploding gradients across many time steps. Those edges are named gates, where each of the LSTM cells has a forget gate (f), an input gate (i) and an output gate (o). A multiplication node ($\prod$) is used to connect those components with each other. The final memory cell reaches an intermediate state between long-term and short-term memory of classic RNNs. Those types of cells outperform simple RNNs in the case of long-range dependencies (Lipton, Berkowitz, and Elkan, 2015). The original work of Shin, Song, and Moazzezi is based on Bi-directional

[1] https://binary.ninja/2017/11/06/architecture-agnostic-function-detection-in-binaries.htm (last access 2021-08-01)

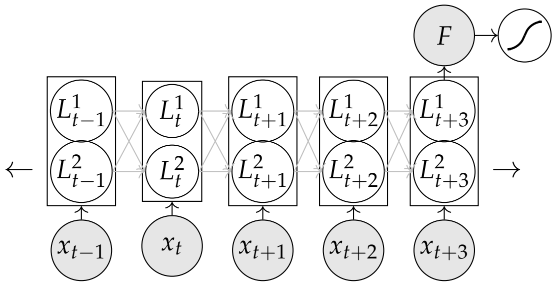


FIGURE 6.2: Simplified RNN model with two LSTM-layers, where the model represents a many to one structure. The final state is processed by a Fully Connected Layer F and a sigmoid layer, which outputs a probability.

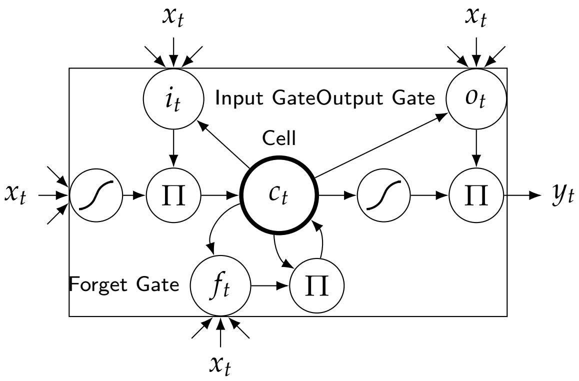


FIGURE 6.3: LSTM Memory cell proposed by Gers, Schmidhuber, and Cummins, 1999, which extends the model of Hochreiter and Schmidhuber, 1997 with additional forget gates (f_t).

RNNs. The authors mention that those models are applicable in the context of having access to the entire binary at once. As we discuss the applicability of RNNs in the context of linear processing large amounts of raw data, we have to consider the temporal component and focus on a LSTM-based model.

TRAINING AND CLASSIFICATION. There are different strategies for training and updating a neural model. The predominant algorithm for training is backpropagation, which is based on the chain rule. The algorithm updates the weights of the model by calculating the derivative of the loss function for each parameter in the network and adjusts the values by the determined gradient descent. However, the problem is a NP-hard Problem and different heuristics try to avoid to become stuck in a local minimum. A common strategy for training is stochastic gradient descent (SGD) using mini-batches. In the course of RNNs with time-related connections between each time step, the process of training is often denoted as backpropagation through time (BPTT) (Lipton, Berkowitz, and Elkan, 2015). Bao et al. (2014) proposed a flipped order of prologues during processing to support the training phase. Additionally, the work shows a better performance for bidirectional structures than unidirectional structures like LSTM. Each model was trained with 100.000 randomly-extracted 1000-byte chunks. A one-hot encoding converts a byte into a $\mathbb{R}^{256}$ vector.

6.1.3 *Weighted Prefix Trees (WPTs)*

Bao et al. (2014) introduced the application of weighted prefix trees for the task of binary function detection, called Byteweight. The approach uses a weighted prefix tree, and matches binary fragments with the signatures previously learned. The path from the root node to the leaf node represents a byte sequence of instructions. Inside the tree, the weights are adapted to the previously processed ground truth. In the original implementation an additional step of value-set analysis and control flow recovery process is proposed for boundary detection.

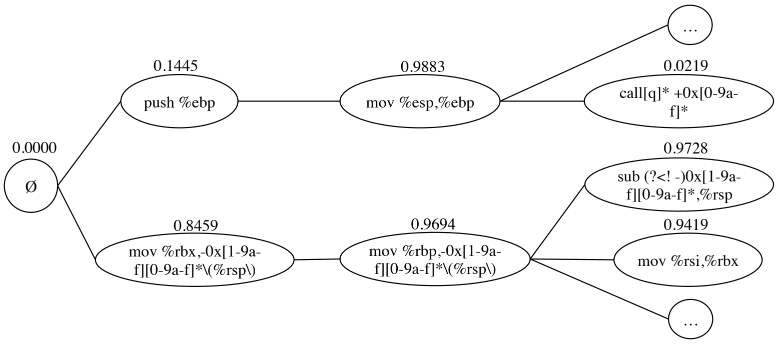

0.0000
Ø
0.1445
push %ebp
0.9883
mov %esp,%ebp
…
0.0219
call[q]* +0x[0-9a-f]*
0.8459
mov %rbx,-0x[1-9a-f][0-9a-f]*\(%rsp\)
0.9694
mov %rbp,-0x[1-9a-f][0-9a-f]*\(%rsp\)
0.9728
sub (?<! -)0x[1-9a-f][0-9a-f]*,%rsp
0.9419
mov %rsi,%rbx
…

FIGURE 6.4: Normalized prefix tree proposed by Bao et al. (2014).

BYTEWEIGHT. Similar to this work and the work of Shin, Song, and Moazzezi (2015), Byteweight focuses on the task of function start identification. More formally, the authors denote the problem as a simple classification problem, where the goal is to label each byte of a binary as either function start or not. Their approach was demonstrated on raw byte sequences and previously disassembled sequences. The reference corpus is compiled with labelled function start addresses. In contrast to raw bytes, the usage of normalized disassembled instructions showed a better performance in the case of precision and recall. The authors proposed a twofold normalization: immediate number normalization and `call-jump` instruction normalization. In Figure 6.4 an overview of the normalized prefix tree is given. A given sequence of bytes or instructions is classified by inspecting the corresponding terminal node in the tree. As soon as the stored value exceeds a previously defined threshold t, the sequence is considered as a function start. We do not consider subsequent steps of advanced control flow graph recovery as proposed by the authors.

TRAINING AND CLASSIFICATION. A corpus of input binaries is used during the learning phase. The maximum sequence length l defines the upper limit of the resulting prefix tree height. The first l elements are used for training, where elements could be disassembled instructions or the raw bytes themselves. The likelihood that a sequence of elements corresponds to a function start (i.e., represented as a specific path in the trie) is saved in each corresponding node as specific weight. Considering the example in Figure 6.4, the instruction `push %ebp` were truly function starts in 14.45% of all cases. The weights of a prefix are lowered if they do not correspond to a function start. As described in Equation (6.1), the weight of a specific node W_n is the ratio between positive function starts (T_+) and all matches ($T_+ + T_-$). The classification of an input sequence is performed by matching the given elements against the tree. The weight of the last matching terminal node describes the final weight of a sequence and is compared to t. For the process of training and classification, the authors proposed an input size of $l = 10$ consecutive instructions and a threshold of $t = 0.5$.

$$W_n = \frac{T_+}{T_+ + T_-}. \tag{6.1}$$

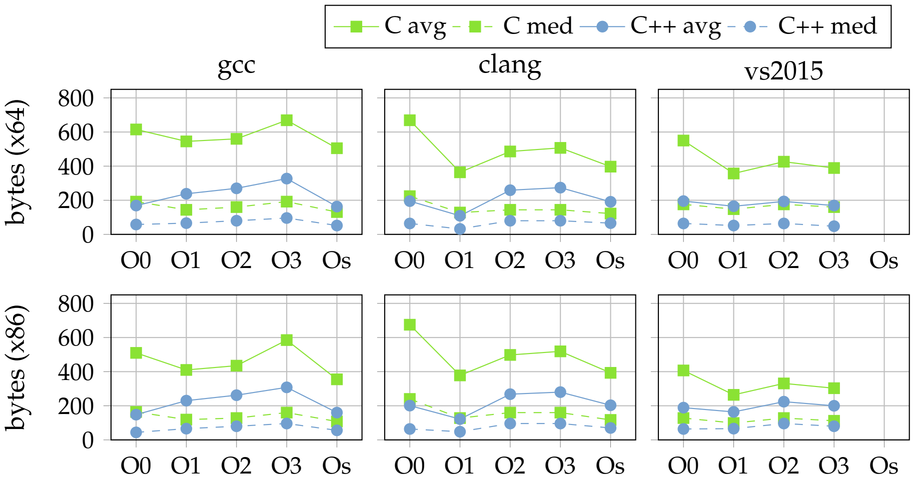


FIGURE 6.5: Average and median function size in bytes.

6.1.4 *Training and Test Dataset*

During the following examination, we focus on three properties of our used data set: *Function Sizes*, *Function Prologue Distribution* and *Mnemonic Distribution*. As already described in the introduction, the examination of the underlying code structure should give us additional insights for better design and parameter decisions.

FUNCTION SIZES. The examination of the frequencies of different function sizes is required for further analysis and inferences. The function size defines the possible size of extracted features, before a single feature vector overlaps with a subsequent function start. In addition, the function size gives better insights into the possible dimension of further function prologues examinations. We examined the function size (in bytes) of the different binaries. As can be seen in Figure 6.5, the function sizes vary for different compiler and optimization levels. The median function size illustrates that in every language, compiler and optimization setting, the amount of small functions (i.e., smaller than 200 Bytes) is significant. The average value of function sizes outlines the presence of large functions, where in all settings the average size is always lower than 800 Bytes.

FUNCTION PROLOGUE DISTRIBUTION. To gain a better understanding of the function detection problem with the help of signature-based detection mechanisms, we examined the present ground truth set and the distribution of common function prologues. We extracted the functions of each binary and aggregated them into a comprehensive set. Figure 6.6 illustrates the population of function prologues by comparing the ratio η between distinct function prologues ρ_d to the overall amount of function prologues ρ for a specific language, compiler and optimization level (see Equation refeq:proldis).

$$\eta = \frac{\text{\# distinct function prologues}}{\text{\# function prologues}} = \frac{\rho_d}{\rho} \tag{6.2}$$

We discriminate the function prologues by their consisting number of instructions i, which are considered and which have been decoded to a single mnemonic. The plot underlines the common axiom that function prologues strongly vary for different compilers, languages and optimization levels. The plot visualizes the impact

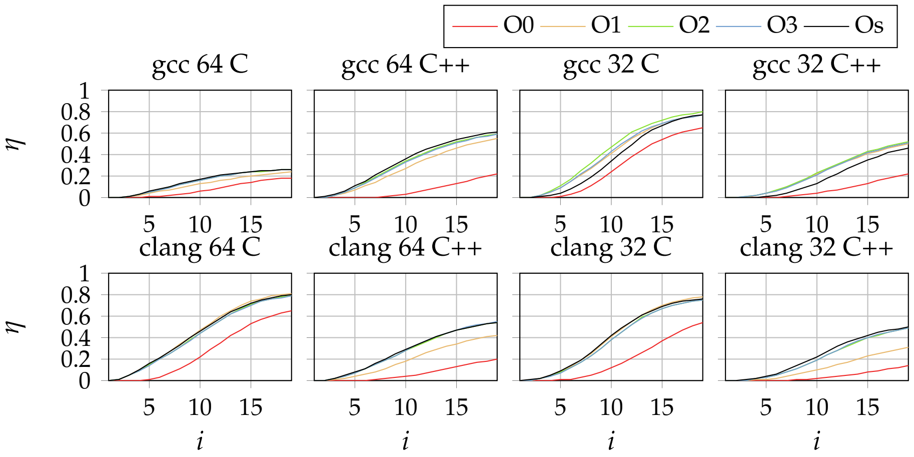


FIGURE 6.6: Illustration of distinct function prologues; η denotes fraction of distinct function prologues to the number of all functions for i instructions (mnemonic).

on the diversity of the prologue instructions dependent upon the selected optimization levels and helps to discuss an appropriate input size.

With the x86 instruction set an instruction could have variable length, where one instruction could vary between one and fifteen bytes. For further details we refer to the Intel Instruction manual[2]. Considering the previous examination of function sizes and the median value of 200 Bytes per function, we do not expect all instructions of a prologue sequence to reach the maximum amount of fifteen bytes. It is clear that a large chosen amount of input mnemonics raises the diversity the model has to deal with, but also increases the possible classification quality. Considering the plots in Figure 6.6, the diversity significantly increases for several settings, even after a considerable short amount of consecutive prologue instructions. For example, inspecting `clang-32-C` with optimization (i.e., `O3`), approximately 40% of all function prologues of 10 consecutive instructions represent a distinct instruction sequence. Figure 6.6 also underlines that non-optimized binaries (`O0`) often share the same beginning instructions (mnemonic).

MNEMONIC DISTRIBUTION. We use the ground truth of assembly files to determine the distribution of mnemonics in the used ground truth dataset of Nucleus (Andriesse et al., 2016). Additionally, we extract the bigrams of mnemonics, which could be often found in the course of assembly-based code and similarity analysis. The following values give us an initial overview of the mnemonics distribution. For details see Table 6.2. Roughly said it is an overview of the instruction distribution of already decoded byte sequences. We split the set of assemblies by its architecture and determined the *total* amount of unigrams and bigrams. In our case a unigram consists of a single mnemonic. The *total* amount of occurring mnemonics also represents the total amount of occurring instructions. The column of *distinct* values describes the set of all occurring mnemonics. The columns *max* and *mean* describe the assignment of the *total* amount of instructions to each *distinct* occurring unigram or bigram. In detail, *the most frequently occurred mnemonic* in the set of 32 bit files, namely `mov`, represents 11,714,270 instructions. Thus, `mov` represents approximately 33.25% of all instructions in the course of 32 bit files.

[2] `https://software.intel.com/en-us/articles/intel-sdm` (last access 2021-08-01).

	32 bit (200 files)				**64 bit (321 files)**			
	total	distinct	max	mean	total	distinct	max	mean
unigrams	35,232 k	322	11,714 k	1531	61,441 k	436	21,627 k	1859
bigrams	35,232 k	11632	5,889 k	17	61,441 k	16059	10,360 k	28

TABLE 6.2: Overview of unigram and bigram mnemonic counts.

6.2 ADAPTATIONS

In this section we discuss concrete adaptations and realizations of linear function detection techniques. Besides the reassessing of machine-learning-based approaches we aim for a reduction of the used feature sizes to improve the theoretically runtime performance. Therefore, we propose an additional step of approximate disassembling via approxis. In previous work the created pipelines are partially based on the processing of features on a byte-level. The input sequences for training and evaluation are one-hot encoded into $\mathbb{R}^{256}$. As our approach is based on integerized mnemonics, the distinct occurring mnemonics define our underlying vocabulary size.

We first describe the general pipeline and the used set for training and evaluation. We explicitly address the problem of imbalanced classes, i.e., the ratio between function starts and general offsets. Afterwards, we introduce the proposed models and concrete adaptations (Section 6.2.1 and 6.2.2).

GENERAL PIPELINE OF FEATURE EXTRACTION. In Figure 6.7 an overview of the single steps of feature extraction is given. We only considered allocable code sections (i.e., `.text`) of the used ground truth ELFs (Andriesse et al., 2016). Thus, we filtered all offsets which are known to be data ❶. Similar to Bao et al. (2014) and contrary to Shin, Song, and Moazzezi (2015), we propose a layer of disassembling for further feature extraction ❷. We additionally reduce the used vectors to mnemonics only. We decode the raw byte sequences into an approximate disassembly with an integerized mnemonic for each instruction. Besides reducing variances in the underlying byte structure, this additionally reduces the overall amount of data which needs to be processed and saved. The classification is not performed at each byte offset, but rather at every instruction offset. Thus, we reduce the overall vector input size and therefore improve the runtime performance.

As we have to deal with a heavily imbalanced set of classes, we first determine the positive (function beginnings) and negative (code offsets) classes for each file ❸. For each class, we created vectors of N_{max} consecutive instructions at each instruction offset, represented by a single and integerized mnemonic. Considering the function prologue distribution in Section 6.1.4, most of the displayed distributions showed tendencies to stabilize after the first 20 instructions. The future vector size should also consider the determined average and median function sizes, as we try to avoid feature vectors which overlap into subsequent functions. Where Shin, Song, and Moazzezi proposed the processing of 1000-byte chunks for RNN-based classification and Bao et al. suggested the usage of 10 consecutive instructions for the creation of Weighted-Prefix-Trees, we considered a maximum vector size of $N_{max} = 20$ mnemonics for creating our data set. This empowers us to vary different input sizes during different model evaluation passes.

We created two sets for both classes and performed an additional step of deduplication for each class over the whole set. This results in two sets of distinct feature vectors ❹. In Table 6.3 an overview of the imbalanced distribution of classes is given.

Set	T_-	T_+	$T_+ \cup T_-$	$T_+ \cap T_-$
Count	18,575,407	207,714	18,779,238	3,883

TABLE 6.3: Overview of class distribution, i.e., number of distinct function starts (T_+) and distinct inner function offsets (T_-) for all used 64 bit binaries.

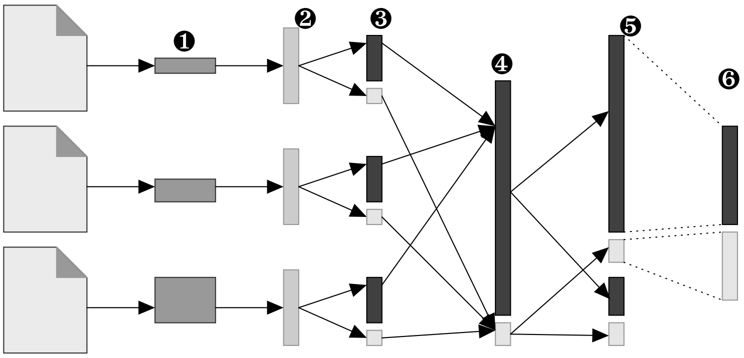
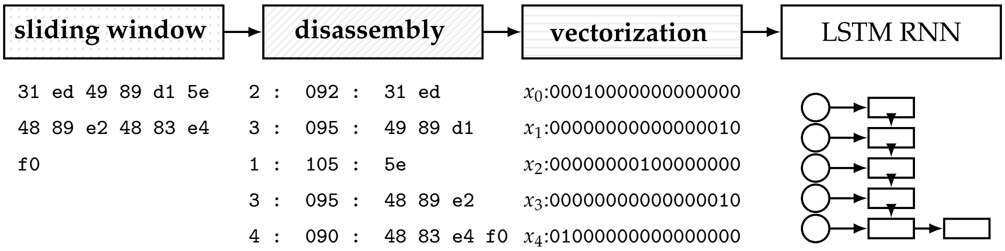


FIGURE 6.7: Simplified overview of the general pipeline of feature extraction.



```
31 ed 49 89 d1 5e    2 :  092 :  31 ed          x0:00010000000000000
48 89 e2 48 83 e4    3 :  095 :  49 89 d1       x1:00000000000000010
f0                   1 :  105 :  5e             x2:00000000100000000
                     3 :  095 :  48 89 e2       x3:00000000000000010
                     4 :  090 :  48 83 e4 f0    x4:01000000000000000
```

FIGURE 6.8: The general processing pipeline during RNN-based classification.

We also examined possible overlaps between each class, i.e., vectors of mnemonics which occur in the positive T_+ and negative class T_-. This is obviously caused by the strong reduction to mnemonic feature vectors. We relabel the negative classes of $T_+ \cap T_-$ to provoke false positives. We outline this decision in Section 6.3. The two distinct sets have been shuffled before and split into a training and test set ❺. In the case of RNN-based training, we additionally added a step of oversampling T_+ and undersampling T_- ❻.

6.2.1 *Recurrent Neural Network (RNN)*

In the following subsection, we describe the final model settings for our linear RNN-based classification. The basic steps of processing are displayed in Figure 6.8. The extracted and integerized mnemonics are transformed into a one-hot encoding and fed into the RNN for further processing. This pipeline is consistent for all further training and evaluation steps performed on our RNN settings. For a better overview, we collated all major differences between our approach and the approach of Shin, Song, and Moazzezi (2015) in Table 6.4.

INPUT VECTORS. We are not expecting to train with an input time series larger than 20. In particular, we try to avoid long series for classification, to minimize the impact on runtime performance later. After the examination of distinct function prologues shown in Figure 6.6, we varied the input vector size between 16 and 20 consecutive and integerized instructions. For the final model, we depicted the vector size of instructions as $N_i = 16$. Similar to Shin, Song, and Moazzezi, we reversed the order of our input vectors, which showed a significant improvement in nearly all of our evaluations. Summarized, the network expects a reversed one-hot encoded input vector $x_t \in \mathbb{R}^K$ for each decoded instruction i_t, where K describes the current size of the vocabulary.

HIDDEN CELLS AND LAYERS. As already described, we use Long Short-Term Memory cells for creating our RNN model. In detail, we use a two-layered model with a different number of hidden cells. To reduce the complexity we use LSTM contrary to bi-directional RNNs as suggested by Shin, Song, and Moazzezi. We initially compared two-layered RNNs with one-layered models. In most of the cases a two-layered model improved the accuracy, which caused us to perform all of the further proceedings with a two-layered model. We vary the number of hidden cells in the subsequent evaluation between 32 and 512 for our two-layered setting. We finally choose the amount of $N_h = 256$ hidden cells.

TRAINING AND OPTIMIZATION. Similar to Shin, Song, and Moazzezi (2015), we performed our steps of optimization by the usage of *stochastic gradient descent*. We tried different concepts of gradient adaptation and initially used *RMSProp* for optimization with different initial learning rates. In the further proceedings of the evaluation, we switched to *AdamOptimizer* which showed a similar performance. The gradients were updated with the help of mini-batches, where the size of the batches was also varied during evaluation and performance tuning. After several manual test runs, we set the final batch size to $N_b = 2048$.

DROPOUT. We reduced the risk of overfitting our models by adding an intermediate Dropout layer. A Dropout step randomly turns off activations of neurons between our two LSTM layers. The Dropout is not applied on the recurrent connections itself (Hinton et al., 2012). A defined value of $N_d = 0.75$ sets the probability if a connection is *not* deactivated. We keep this value for all of our trained models.

OUTPUT LAYER. For a binary classification of the function starts we process the output of the final LSTM state by a fully connected layer. The final layer of the setup is a sigmoid layer, which is used for transforming our output into a binary classification. A single probability is generated with the help of an additional *sigmoid cross entropy* layer or *weighted cross entropy* layer. With the usage of a *weighted cross entropy* layer, we could additionally set the focus on improving our final recall or precision rate.

TRAINING AND SAMPLING. To handle the heavily imbalanced ground truth, we performed an additional step of oversampling the positive and undersampling the negative class. Those steps had a remarkable influence on the performance in terms of precision and recall. During training we randomly selected the half of the negative class members (inner function offsets) and oversampled the positive class members (function starts) to an equal degree.

	Our approach	Shin, Song, and Moazzezi
Model	LSTM based RNN	bi-directional RNN
Input Entity	one-hot encoded mnemonics	one-hot encoded bytes
Input Size	16	1000
Vocabulary	$K_{64} = 436, K_{32} = 322$	$K_{64} = K_{32} = 256$

TABLE 6.4: Comparison of our approach with Shin et al.

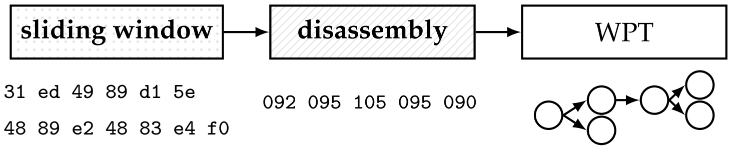


FIGURE 6.9: The general processing pipeline during WPT-based classification.

6.2.2 *Weighted Prefix Tree (WPT)*

The major adaptation in the case of our proposed WPT is the utilization of single mnemonic representatives instead of raw bytes or normalized disassembly instructions. The basic steps of processing are displayed in Figure 6.9. We selected different lengths for trie creation up to $l_{max} = 20$ instructions. The mentioned vocabulary size K (see Section 6.1.4) gives us the theoretical upper limit for the possible trie size. We depict an initial threshold value of $t = 0.5$ and $l = 10$, which was proposed by Bao et al. (2014).

TREE PRUNING. Similar to the original approach we additionally performed a step of tree pruning. In detail, we deleted all intermediate nodes with no negative counts ($T_{-} = 0$) and thus, all first occurring intermediate nodes with $W_i = 1$ are transformed to terminal nodes. This reduces the overall tree size from approximately 1.9 million nodes to 264,834 nodes.

Thus, our approach does not process raw byte sequences and differs from the previously introduced approach of Shin, Song, and Moazzezi (2015). Bao et al. (2014) compared the capabilities of raw byte sequences and additionally normalized the full instruction sequences, where the normalized version showed slightly improved results in terms of precision and recall. The process of normalization additionally reduced the overall size of the used prefix tree. In detail, the trie structure was pruned from 2,483 elements to 1,447 elements.

6.3 EVALUATION

In this section, we discuss the different evaluations of our proposed models. We inspect the time of creation and training our models, without considering the pass of feature extraction (Section 6.3.1). Afterwards, we examine the performance in terms of Accuracy (ACC), Precision (PRE) and Recall (REC) for both models (Section 6.3.2). Finally, we discuss in detail the possible application of weighted prefix trees (Section 6.3.3). Our evaluation aims to answer the following questions:

1. Could we use only mnemonics for the task of function start identification?

2. Which of the considered linear techniques is better suited for our context?

CLASSIFICATION GOALS. We consider our models for the fast identification of function starts in large portions of raw data. As described in Section 6.2, the reduction of our proposed features leads to overlapping class members. We relabelled those members to positive members and further consider recall more important than precision: we accept higher values of false positives by lower values of false negatives with a constant high value of true negatives. We propose an additional step of classification for lowering the value of false positives afterwards.

METHODOLOGY. To argue if our models generalise the task of function detection, we performed a 10-fold cross validation by dividing our set of *distinct* integerized mnemonic vectors into ten equally sized sub-sets. We used nine of those sub-sets for training and one for evaluation. In the case of WPT-based training, we repeated this task 10 times. In our current evaluation we focus on 64 bit ELF binaries of the Linux operating system. Our used test set consists of nearly 1.9 million unique vectors with approximately 20 k positive cases. The set of training consisted of 16 million negative and nearly 190 k positive class members.

6.3.1 *Training Performance*

First, we examine the general runtime performance in the case of model creation and training. Both models have been fed with the already prepared integerized mnemonic vectors and a maximum length of 20. In the case of our RNN model, we already cropped the initial feature vectors to a maximum size of 16.

As already described in Figure 6.8, in case of training our proposed RNN the processing began with the reversing of the input vector and the one-hot encoding. Considering those steps, the training of our model took approximately 2.5 hours for one epoch. As we trained our model for 10 to 20 epochs, the process most often took *several days* on a machine with 64 cores and more than 256 GiB of RAM.

In contrast, the creation of the weighted prefix tree was performed on an ordinary Laptop with Intel Core i5 2x 2,2 GHz Processor and 8 GiB RAM. The time of building the initial tree, calculating the final weights, pruning the tree and performing an additional step of lookup took approximately *180 seconds* in total. Our current prototype implementation was realized in Python.

6.3.2 *Function Start Identification*

For our final models, we described the settings in Section 6.2.1 and Section 6.2.2. As can be seen in Table 6.5, the application of our RNN performs similarly to our proposed WPT. In both cases we gain a high value of accuracy with lower values of precision and recall. However, the results of the WPT clearly outperforms the RNN in case of precision. As already described in Section 6.2.1, the adaptation of the underlying RNN model could influence the classification in terms of focusing on better values of recall or precision. However, contrary to our WPT implementation, the adaptation of those RNN parameters must be done before the training of the model. The WPT approach enables influencing the value of recall by parametrizing the lookup, which is more flexible than approaching the perfect parameters by multiple time-consuming training passes.

Model	ACC	PRE	REC	Population	TP	TN	FP	FN
WPT*	0.9943	0.7532	0.7233	1,878 k	15 k	1,852 k	4,924	5,747
RNN	0.9861	0.4300	0.7674	1,878 k	16 k	1,836 k	21,128	4,831

TABLE 6.5: Performance of the function start identification for x86-64 ELF binaries. (*:average values of 10-fold cross validation)

			ACC			**PRE**			**REC**			**Time (sec.)**		
t	*m*	*l*:	10	14	18	10	14	18	10	14	18	10	14	18
0.6	0		0.994	0.994	0.994	0.786	0.766	0.758	0.682	0.722	0.727	11.04	11.13	11.15
	4		0.994	0.994	0.994	0.788	0.768	0.760	0.679	0.718	0.724	11.06	11.16	11.15
	8		0.993	0.993	0.993	0.820	0.789	0.778	0.506	0.546	0.551	11.06	11.13	11.37
0.5	0		0.993	0.993	0.993	0.709	0.686	0.678	0.738	0.774	0.780	11.40	11.85	11.87
	4		0.993	0.993	0.993	0.712	0.689	0.681	0.735	0.771	0.777	11.42	11.08	11.56
	8		0.993	0.992	0.992	0.750	0.714	0.702	0.554	0.590	0.595	11.58	11.55	11.42
0.4	0		0.993	0.992	0.992	0.668	0.647	0.641	0.769	0.797	0.802	11.31	11.48	11.88
	4		0.993	0.993	0.992	0.675	0.653	0.647	0.764	0.793	0.798	11.80	11.65	11.32
	8		0.992	0.992	0.992	0.728	0.693	0.683	0.574	0.603	0.608	11.32	11.90	11.95
0.3	0		0.991	0.991	0.991	0.594	0.575	0.570	0.804	0.831	0.836	11.29	11.41	11.53
	4		0.992	0.991	0.991	0.607	0.587	0.582	0.800	0.826	0.831	11.50	11.43	11.41
	8		0.992	0.992	0.991	0.675	0.641	0.632	0.606	0.633	0.637	11.05	11.13	11.27

TABLE 6.6: WPT classification by varying different parameters t, l and m.

6.3.3 *Examination of WPT*

As the previous evaluations underlined, we achieved similar or better classification capabilities in less training time by the utilization of our WPT model. For the WPT model, we could additionally adapt the classification goal during a lookup pass. This leads to a detailed examination of our WPT model.

To improve the quality of WPT-based classification, we examined the different parameters and their influence on the overall classification task. In detail, we varied different parameters, like the maximum sequence length l and the threshold t. In addition, we introduce a third parameter m, which denotes the minimum required length of a matching sequence inside the tree. After the inspection of the prologue distributions in Section 6.1.4, we selected a range from 0 to 18 (instructions) for the parameters m and l. Respecting our classification goal, we focus on runtime performance by accepting a higher amount of false positives instead of false negatives. Inspecting the classification in Table 6.6, we could argue that the reduction of false negatives could be achieved by increasing the considered vector size l. The processing time for the nearly 1.9 million test vectors was approximately 12 seconds for all of the parameter variations. We should mention that during our initial implementation we did not focus on any runtime optimizations.

6.4 DISCUSSION

In this chapter, we inspected the capabilities of linear function detection and underlined the need for signature-based detection methods in the course of memory carving. After performing a detailed analysis of our underlying ground truth, we introduced several considerations and model adaptations. The main adaptation of our approach is the utilization of mnemonics only.

Our analysed models showed good classification results in terms of accuracy, where we achieved for RNN- and WPT-based models an accuracy value of above 98%. The utilization of menmonic-based weighted prefix trees showed good capabilities for our considered context of application. The application of WPT performs the classification of 1.9 million offsets in 12 seconds and reaches with simple parameter adaptations an acceptable value of recall beyond 80%.

Considering the mentioned techniques within multiple steps of classification, our proposed WPT could be used for the fast identification of possible function starts and drastically reduces the amount of offsets which need to be reclassified.

7

REVISITING THE DATABASE LOOKUP PROBLEM

7.1 INTRODUCTION

In Section 2.1.3 we already gave a broad introduction in the Database Lookup Problem (DLP) and three different more or less widespread lookup strategies for storing hash-based fragments. As all of the mentioned approaches strongly differ (either in their original use case, or in their supported capabilities), we outline the motivation behind our choice of the depicted candidates. Therefore, we first describe the conditions of application and introduce the forensic use case which formalizes additional requirements and mandatory features (see Section 7.1.1). Afterwards, we explain our three candidates of choice in Section 7.1.2. Besides the required features of our considered use case, we need to discuss the already present features and capabilities of the different approaches. Thus, we outline the existing features and capabilities for each candidate in Section 7.1.3. Considering our depicted candidates and the overall goal of reassessing those in terms of capabilities and performance, the goals of this assessment are as follows:

1. Assess the aforementioned proposed techniques for the task of fast artefact handling (i.e., hbft, fhmap and hashdb). Identify the **capabilities** of those techniques and the possible handling of *common blocks*.

2. Inspect the feasibility and discuss concepts to integrate the missing feature of multihit prevention (**filtration** of *common blocks*), similar to hashdb, into hbft or fhmap.

3. Discuss possible extensions of existing techniques in order to be able to compare the approaches.

4. Assess how the different approaches compete with respect to **runtime performance** and resource usage.

7.1.1 *Use Case and Requirements*

In this work we address the problem of querying digital artefacts out of a large corpus of relevant digital artefacts. Sample applications are carving or Approximate Matching. Both applications suffer from the *Database Lookup Problem*, i.e., how to link an extracted artefact within a forensic investigation to a corresponding source of a forensic corpus efficiently (i.e., in terms of required storage capacity, required memory capacity or lookup performance). Besides those, our digital forensic scenarios reveal additional pitfalls and challenges.

We consider the extraction of chunks (i.e., substrings) out of a raw bytestream, without the definition of any extraction process. A major challenge of matching an artefact to a source is occurring *multihits*, i.e., one chunk is linked to multiple files. This was first mentioned by Foster (2012). A multihit is also called a *common block* or a *non-probative block*. For instance, Microsoft Office documents such as Excel or Word documents share common byte blocks across different files (Garfinkel and McCarrin, 2015). Similar problems occur during the examination of executable binaries which have been statically linked (e.g., share a large amount of common code or data). In summary, multi matches are a challenge in identifying an unknown fragment with full confidence. In addition, storing multihits also increases memory requirements and decreases lookup performance.

Multihits can either be identified during the *construction phase* of the database (e.g., by deduplication or filtration) or during the *lookup phase*. By filtration of common blocks during a construction phase, the overall database load becomes reduced and the lookup speed is increased as only unique hits are considered. Two different strategies of multihit prevention during the construction phase were proposed. First, as introduced by Garfinkel and McCarrin (2015) *rules*, are defined to filter out known blocks with a high occurrence (and thus low identification probability of an individual file). Such an approach requires extensive pre-analysis of the input set and its given structures. A second approach is the filtration of common blocks during construction by the additional integration of a deduplication step. Aside from hashdb, none of our candidates provides deduplication or multihit prevention techniques so far. We refer to Garfinkel and McCarrin (2015) for further details and solely focus on the utilized database in the following subsection.

Features of adding and deleting artefacts have the major benefit of not needing to re-generate the complete database every time a new artefact needs to be included. While deleting inputs may be less frequent, adding new items to an existing storage scheme seems obvious and indispensable. While fhmap and hashdb support adding and deleting hashes from their scheme, this feature is not yet available in the current prototype of hbft. In detail, adding new elements to a hbft is possible, however, the tree needs to be re-generated as soon as a critical point of unacceptable false positives is reached. The definition of buckets also limits the capabilities of adding further files to the database. Loosing the capabilities of deleting elements out of a binary Bloom filter is the main reason for making features of deletion impossible to realize. In summary, adding and deleting hashes from a database is a mandatory or optional feature, depending on the specific use case.

7.1.2 *Depicted Candidates*

In this section we present three widespread lookup strategies suitable for storing hash-based fragments: (1) Hashdatabase for hash-based carving (hashdb), (2) hierarchical Bloom filter trees (hbft) and (3) flat hash maps (fhmap).

LMDB / HASHDB. To store the considered blocks Garfinkel and McCarrin (2015) make use of hashdb, a database which provides fast hash value lookups. The idea of hashdb is based on Foster (2012) and Young et al. (2012). In 2018, the current version

(3.1[1]) introduces significant changes compared to the original version mentioned by Garfinkel and McCarrin (2015).

The former implementation of hashdb originally supported B-Trees. Those have been replaced by the *Lightning Memory Mapped Database* (LMDB) which is a high-performance and fully transactional database (Chu, 2011). It is a key-value store based on B+ Trees with shared-memory features and copy-on-write semantics. The database is read-optimised and can handle large data sets. The technique originally focused on the reduction of cache layers by mapping the whole database entirely into memory. Direct access to the mapped memory is established by a single address space and by features of the operating system itself. Storages are considered as primary (RAM) or secondary (disk) storages. Data which is already loaded can be accessed without a delay as the data is already referenced by a memory page. Accessing non-referenced data triggers a page-fault. This in turns leads the operating system to load the data without the need of any explicit I/O calls. In summary, the fundamental concept behind LMDB is a single-level store, the mapping is read-only and write operations are performed regulary. The read-only memory and the filesystem are kept coherent through a Unified Buffer Cache. The size is restricted by the virtual address space limits of an underlying architecture. As mentioned by Chu (2011), on a 64 bit architecture which supports 48 addressable bits, this leads to an upper limit of 128 TiB of the database (i.e., 47 bits out of 64 bits).

HBFT. The concept of hierarchical Bloom filter trees (hbft) is fairly new. This theoretical concept was introduced by Breitinger, Rathgeb, and Baier (2014) and later implemented by Lillis, Breitinger, and Scanlon (2017). The lookup differs from the Approximate Matching algorithm mrsh-v2, as hbft only focuses on fragments to identify potential buckets of files. A parameter named *min_run* describes how many consecutive chunk hashes need to be found to emit a match. A good recall rate was accomplished for *min_run* = 4. The tree structure is then traversed further if a queried file is considered a match in the root node. Each of the nodes is represented by a single Bloom filter which enables traversing the tree. A traditional pairwise comparison can be done at possible matching leaf nodes. For details of the actual traversing concept we refer to the original paper Lillis, Breitinger, and Scanlon (2017).

Just like previous mrsh implementations the lookup structure can be precomputed in advance. First, the tree is constructed with its necessary nodes. Then the database files are inserted. Thus, the time for construction can be disregarded in the actual comparison phase. More precisely, the tree structure is represented as a space efficient array where each position in the array points to a Bloom filter. The implementation uses a bottom up construction which fills trees from the leaf nodes to the root. The array representation does not store references to nodes, its children, or leaves explicitly. Every reference needs to be calculated depending on the index in the array. Efficient index calculations are only applicable for binary trees. The lookup complexity within the tree structure is $O(\log_x(n))$, where x describes the degree of the tree and n is the file set size.

FHMAP. Flat hash maps have been introduced as fast and easy to realize lookup strategies. Up to now, they have been discussed in different fields of application. Similar to hbft, the actual implementation of fhmap represents a proof of concept implementation with good capabilities but limited features.

[1] http://downloads.digitalcorpora.org/downloads/hashdb/hashdb_um.pdf (last access 2021-08-01).

The concept of flat hash maps is an array of buckets which contain multiple entries. Each entry consists of a *key-value* pair. The *key* part represents the identifier for the *value* and is usually unique in the table. An index of the bucket is determined by a hash function and a modulo operation. The position i equates to: $hash(key) \bmod size(table)$. A large amount of inserts into a small table causes collisions, where multiple items are inserted in the same bucket. A proper hash function needs to be chosen in order to maintain a lookup complexity of $O(1)$. The function needs to spread the entries without clustering. The amount of inserted items is denoted by the *load factor*, i.e., the ratio of entries per buckets. A high load factor obviously causes more collisions. The table becomes slower since the buckets must be traversed to find the correct entry. The lower the load factor the faster the table. However, more memory is required since buckets will be left empty on purpose. If a slot is full the entries are re-arranged.

The whole table is implemented as a contiguous (*flat*) array without buckets which allows fast lookups in memory. With linear probing the next entry in the table is checked if it is free. If not the next one is checked until either a free slot is found or the upper probing limit is reached. The table is re-sized as soon as a defined limit is reached. The default *load factor* of this table is 0.5. Specific features should speed up the lookup phase: open addressing, linear probing, Robin Hood hashing, prime number of slots and an upper limit probe count. Robin Hood hashing introduced by Celis, Larson, and Munro (1985) ensures that most of the elements are close to their ideal entry in the table. The algorithm rearranges entries: elements which are very far away will be positioned closer to their original slot, even if it is occupied by another element. The element which occupies this specific slot will also be rearranged from its possibly ideal slot to enable each element to have approximately equal distances for each element to its ideal position in the table. The algorithm takes slots from rich elements, which are close to their ideal slot, and gives those slots to the poor elements which are very far away, hence the name.

7.1.3 *Feature Analysis*

In what follows we shortly inspect the capabilities and properties of all considered techniques. We discuss the current state of each approach in the case of existing features and properties. Table 7.1 provides a summary of the discussed capabilities. Note, the marks (*) and (†) mean that these attributes are introduced or discussed, respectively, in the course of this work. For instance, an important extension in the case of fhmap is the integration of an appropriate chunk extraction and insertion technique.

- **Block Building.** In the case of hashdb the database building and scanning of images is now possible without the use of *bulk extractor* which was originally proposed to extract chunks. It builds and hashes the blocks with a fixed sliding window which shifts along a fixed step size s. Obviously this produces quite a lot of block hashes to be stored in the database. Similar to the original mrsh-v2 algorithm, the current hbft implementation identifies chunks by the usage of a Pseudo-Random-Function (PRF). As soon as the current byte input triggers a previously defined modulus a new chunk boundary is defined. The current implementation sticks to the originally proposed `rolling_hash` Breitinger and Baier, 2012b. Thus, the extraction of chunks relies on the current context of an input sequence and not on a previously defined block size. Those CTPH algorithms prevent issues by changing starting offsets of an input sequence.

	hashdb	**hbft**	**fhmap**
Storing Technique	LMDB	Bloom filter tree	Hash table
Block Building	Fixed sliding window	**Fixed size*** / rolling hash	**Fixed size*** / **rolling hash***
Block-Hashing	MD5	FNV-256	FNV-1
Multithreading	All phases	**Block building***	**Block building***
Multihit Handling	✓	*	*
Add / Remove Hashes	✓/ ✓	**Partially /** †	✓/ ✓
Prefilter	"Hash Store"	Root Bloom filter	✗
False Positives	✗	✓	✗
Storing Type	Single-level storage	Primary storage	Primary storage
Not limited to RAM	✓	✗	✗
Persistent Database	✓	✓	*

TABLE 7.1: Features of hashdb, hbft and fhmap. New implementations are marked with asterisks (symbol *, table cell is coloured green) and potential techniques are marked with a dagger (symbol †, table cell is coloured red).

The fixed defined modulus b_m approximates the extracted block size on average. Unlike hashdb or hbft, the fhmap implementation itself obviously does not feature any block building or hashing. As it will just serve as a container, we have to extend the capabilities to extract, hash and store fragments during evaluation.

- **Block-Hashing Algorithm.** In case of hashdb the authors propose the cryptographic hash function *MD5* and an initial blocksize b of 4 KiB. A value which is obviously inspired by the common cluster size of today's file systems. hbft makes use of the FNV hash function to hash its chunks and sets 5 bits in its leaf nodes and the corresponding ancestor nodes. Since the root Bloom filter is considerably large, the adapted version of FNV which outputs 256 bits is required. Finally, fhmap makes use of FNV-1 for hashing an input.

- **Multithreading support.** With respect to multithreading support, hashdb (or *Lightning Memory Mapped Database*) allows multithreaded reading operations to improve the runtime performance. However, up to now, no theoretical or practical concepts are available to integrate mulithreading support into hbft and fhmap, respectively. Nevertheless the block building phase may use multithreading for both hbft and fhmap.

- **Multihit handling.** hashdb associates hashed blocks with meta data. A meta data describes the count of matching files for a specific block. The saved counts are used to remove duplicates from a database and to rule out multi-matches in advance. In detail, all hashed blocks which have an associated counter value higher than one are removed (Garfinkel and McCarrin, 2015). The current prototypes of both hbft and fhmap do not provide any functionality of deduplication. Thus, a feature for filtering common blocks and common shared chunks is missing. We address this problem in Section 7.2 and extend both prototypes.

- **Add and remove hashes.** hashdb supports adding new hashes into a database (with integrated deduplication). First, the new files are split into blocks and hashed. Afterwards, hashes are inserted into the database. The implementation also supports deletion of hashes from a given database by subtracting a database from the original. In order to insert new files into an existing hbft database, the tree needs to be rebuilt with the new file hashes if the tree was limited for the original file set. One can save the original block hashes in order to avoid rehashing. The current concept of the structure does not provide any functionality for deleting a given chunk hash. This is obviously primarily caused by the nature of Bloom filters. The original implementation of fhmap supports adding and deleting entries by default. After determining the index in the table, the entries are re-arranged as soon as a slot is full. After adding or deleting, the table is optionally re-sized to a final *load factor* of 0.5.

- **Prefiltering of non-matches.** hashdb provides prechecking by a *Hash Store*. A *Hash Store* is described as a "highly compressed optimized store of all block hashes in the database"[2]. In the case of hbft the root Bloom filter provides an easy discrimination between a match and a non-match and thus yielding prefiltering of non-matches. The current version of fhmap does not provide any prechecking or prefiltering mechanisms. The additional implementation of a Bloom filter may solve this problem and is the object of future research.

- **False Positives.** Similar to a probabilistic lookup strategy like Bloom filters, the currently proposed *Hash Store* of hashdb causes false positives. Even if the concept of prechecking produces false positives, hashdb still performs a complete lookup of the queried hash value. Thus, the overall lookup does not suffer from any false positives. A major disadvantage of utilizing a probabilistic lookup strategy like in case of hbft is the possible collision of lookups. Thus, the lookup strategy suffers from false positives. The expected value of false positives is controlled by the size of the root Bloom filter and the handled amount of inserts. As fhmap performs a full lookup on the stored hash values, the approach does not suffer from any possible false positives.

- **Limited to RAM.** The current implementation of hashdb integrates capabilities of loading and storing entries from and to a disk. Thus, the approach is not limited to any memory boundaries. The overall design and construction of the hbft tree heavily relies on the memory constraints. The original implementation was created as a RAM-resident solution only. The proposed parametrization and initialization mainly focuses on memory boundaries of a target system. In the case of fhmap, the structure of the contiguous array is directly created in memory and thus, the current approach is limited to the given memory boundaries of system.

- **Persistent Database** The database of hashdb is persistent. The current implementation of hbft offers the possibility to save and load a database to and from a disk. The recent prototype of fhmap was only proposed as a simple proof of concept. No features for saving or restoring a disk-based database have been considered so far.

7.2 EXTENSIONS TO HBFT AND FHMAP

In this section we discuss our candidates hbft and fhmap with respect to both already implemented and potential future extensions. The extended code is available

[2] http://downloads.digitalcorpora.org/downloads/hashdb/hashdb_um.pdf (last access 2021-08-01).

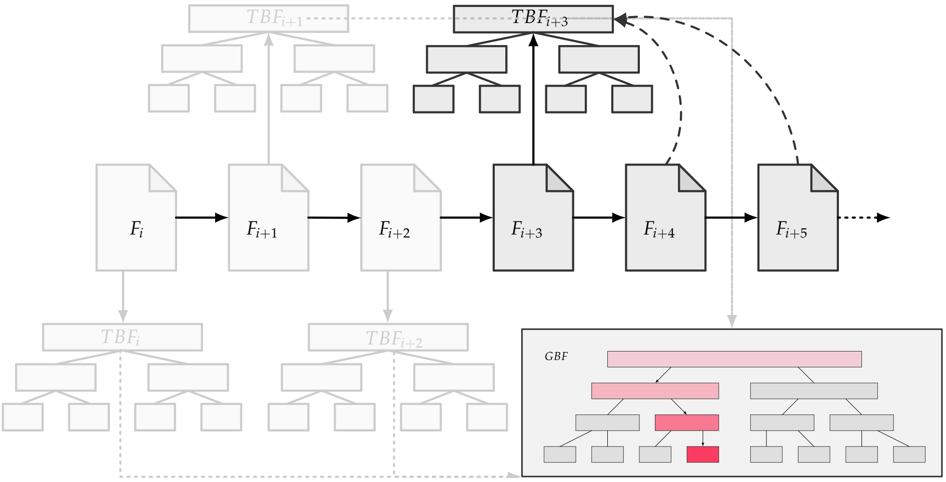


FIGURE 7.1: Tree-filter-based multihit prevention with temporary hbfts per file. Temporary Bloom filter trees (TBF_i) are used to filter multihits for a current file F_i and all its subsequent files. A global hbft (GBF) represents all unique elements in the processed set.

for both hbft[3] and fhmap[4]. In contrast, hashdb already fulfills most of the required capabilities and is the only ready-to-use approach for our considered context. In Section 7.2.1 we first discuss strategies for the handling of multihits in the case of hbft, benchmark them, and depict a proper candidate. In Section 7.2.2 we discuss the possible multihit prevention via deduplication for fhmap. The integration of persistence for fhmap will not be outlined in detail in the course of this work. In Section 7.2.3 we discuss the chunk extraction process in the case of hbft and fhmap. We will additionally introduce a concept for the parallelization of the chunk extraction via a rolling hash function. Finally, we will outline in Section 7.2.4 some theoretical extensions and thoughts.

7.2.1 *Multihit Prevention hbft*

The prevention of multihits (i.e., the filtration of common blocks) could be differentiated at the construction or the lookup phase. In the following we discuss two approaches of multihit prevention, one realized during the construction phase and one during the lookup phase.

TREE-FILTER BASED. By the utilization of a temporary hbft for each file, multihit chunks could be marked during the construction phase. Each file is processed sequentially one after another. As can be seen in Figure 7.1, a temporary hbft stores the chunks of a currently selected file. The following files are compared against the temporary hbft. A multihit is highlighted within the tree by a counter. The currently compared chunk of a processed file is also labelled with a counter. After processing all of the subsequent files, unique chunks of the current tree are saved into a global hbft. The next file generates a temporary hbft again. However, only chunks with a zero counter are considered during the pass. The final tree stores unique chunks in different leaf nodes. Thus, it could be guaranteed that the tree does not feature the same chunk in a different leaf node.

[3] `https://github.com/ishnid/mrsh-hbft` (last access 2021-08-01).
[4] `https://github.com/skarupke/flat_hash_map` (last access 2021-08-01).

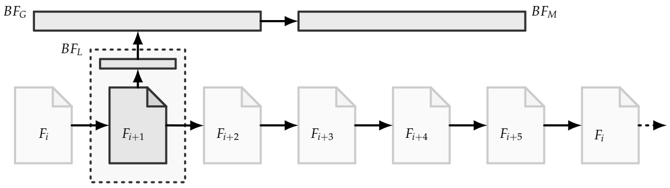


FIGURE 7.2: Global-filter based prevention of multihits. Three different Bloom filter are used to filter multihits: A local Bloom filter (BF_L), a global multihit Bloom filter (BF_M), and a global Bloom filter (BF_G).

Obviously, this approach has a higher building time but offers additional features. A possible advantage of utilizing counters for each chunk could be the definition of a threshold for accepted multihits. Thus, by the definition of a threshold a tree is generated which stores the maximum amount of elements in each leaf node. This enables identifying chunks related to multiple files. Considering an interleaved multihit between two unique hits, an investigator could infer the gap between both.

GLOBAL-FILTER BASED. A straightforward approach could be the utilization of a separate Bloom filter which represents all multihits for a target file set. Therefore, two Bloom filters are generated with an adequate size (i.e., a size which respects an upper limit of false positives).

As shown in Figure 7.2, a single *global* Bloom filter stores all chunks of a file set. A second *global multihit* Bloom filter will store all multihits for a corresponding set. A temporary *local* Bloom filter is generated for a specific file and gets zeroed out before another file will be processed. The local filter enables distinguishing multihits within a file itself. Recalling the informal definition of a multihit, a multihit within a file itself but with no matches in other files could still be used for unique identification and is desired to keep. The local filter emits possible multihits on a file base. Those are ignored and not further processed in the global filter. If a chunk is neither in the local filter, nor in the global filter, it will be inserted in the global filter. We consider such a chunk as a unique chunk until proven otherwise. If a chunk is already in the global filter, an identical chunk has been seen before. Such a chunk will be further considered as a multihit and gets inserted into the multihit filter. This process is repeated for each file. The result is a global filter which stores all occurred multihits. A set (including multihits) could be stored in a global hbft with an additional check against the multihit filter. An additional step of deduplication could shift the prevention of multihits to the construction phase.

EVALUATION AND SELECTION. We benchmarked[5] the two introduced approaches in the case of construction and lookup runtime. For our benchmark we make use of the t5-corpus[6] and each node in the tree will represent one file of the corpus. The corpus consists of 4,457 files with a total size of approximately 1.9 GiB. The interface was extended to additionally handle a single large image as input. We constructed an image by concatenating all 4,457 files of the set. The amount of extracted chunks by the utilization of the *rolling_hash* with a block size $b = 160$ produces 8,311,785 chunks. In total 457,793 of the chunks are multihits (shared in more than one file).

Figure 7.3 outlines the results of the benchmark. The construction phase does not differ for the *file* or *image* lookup. The prevention is handled during the construction

[5] performed on a laptop with Intel I7 2.2 GHz, 8 GiB of RAM and a SSD

[6] http://roussev.net/t5/t5.html (last access 2021-08-01).

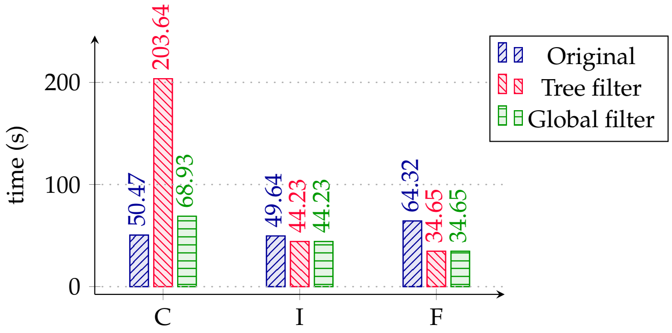


FIGURE 7.3: Benchmark of multihit prevention approaches for hbft: Construction (C), Lookup Image (I), Lookup Files (F).

phase, as we focus on an improved lookup performance. The deduplication handling causes longer construction times in both cases. The tree filter approach clearly exceeds the global filter approach in terms of construction time. Depicting an appropriate candidate (i.e., Tree-filter or Global-filter) is a trade off between performance and matching capabilities. By the utilization of a global filter, we loose the ability to match multihits to their root files.

Lookup timings are identical for both techniques as both approaches filter multihits before the construction of the tree. Once the search for a file hits a leaf node it can be assumed that the query will not match any other leaf nodes. This holds true considering the false positives caused by the probabilistic Bloom filters. Even if it depends to the overall use case how often a filter has be rebuilt from scratch, the time needed to construct a tree filter becomes intolerable for larger data sets. Therefore, this enhancement will not be pursued any further and we propose the usage of a global filter.

7.2.2 *Multihit Prevention fhmap*

Originally and naturally, hash tables do not feature our considered handling of multihits by design. Each of the inserted keys should be unique. Implementations behave differently upon multihit insertions. Some databases will simply overwrite existing values, while others will not insert the value at all as soon as a key is already occupied. However, there are hash tables which support duplicated keys. This section presents an algorithm which prevents multihit insertions in any hash table.

Each inserted chunk is represented by a hash value, i.e., the actual *key*. The value of the corresponding key is a reference to the filename from which the chunk originates. We assume that the corresponding chunks are multihits if two hash values are identical. In order to rule them out in the final database, each chunk will be looked up first. If a key is not in the table it can be safely inserted. If a key is already present in the table it is very likely a multihit. As already explained, multihits which occur in a file itself, but not in any other file, should be kept. If the found value of the key (i.e., the filename) is equal to the value of the key which needs to be inserted, it is a multihit in a file itself. The insertion is ignored since the chunk is already represented in the table. If the found value is not equal to the query value, the chunk is a multihit within the file set. The entry will be marked as a duplicate in the table. This procedure is done for all chunks in the file set and comparable.

In the second processing step each entry is checked for the duplicate mark. If a mark is found the entry will be deleted from the hash table. This algorithm also reduces the amount of inserted chunks while keeping unique insertions only. Fewer insertions also means less collisions and the increase of lookup speed. We will inspect the runtime, also in terms of deduplication, in the following section. Keeping the multihits is possible and would be comparable to the concept of hashdb which keeps internal counters to inserted chunks. However, this would force the database to handle multihits during the lookup phase.

7.2.3 *Chunk Extraction hbft and fhmap*

Chunk extraction means the following. A given input sequence of bytes is divided into chunks by the definition of a fixed modulus b (common values are $64 \leq b \leq 320$ bytes). The extraction algorithm iterates over the input stream in a sliding window fashion, rolls through the sequence byte-by-byte, and processes seven consecutive bytes at a time. A current window is hashed with a *rolling_hash function*, which returns a value between 0 and b. If this value hits the value $b-1$, a trigger point is found and thus defines the boundary of a current chunk.

We consider an example of block building and querying an image of 2 GiB. Reading in the image, constructing the block hashes via *rolling_hash*, and hashing the blocks via FNV-256 takes approximately 43 seconds. Querying each chunk against the hbft takes approximately 8.5 seconds (with each chunk present in the hbft). Thus, the extraction without lookup takes about 83.4 % of the overall query time for a single process.

Besides the evaluation of lookup strategies, we also discuss the possible parallelization of the extraction process itself. A possible parallelized version of the *rolling_hash* is depicted in Figure 7.4. We evenly split the input S into parts P_i which start at byte s_i (offset) and end at e_i (offset). The split blocks are further processed by a rolling_hash function which subsequently defines the chunk boundaries within each block. Thus, each P_i contains 0 or more chunks C_i. The challenge arises at each chunk boundary, i.e., the last chunk in P_i and the first in P_{i+1} (marked red). The chunk boundaries are implicitly defined by the block boundaries and do not match with the original chunk boundaries of S. The naive alignment of blocks, which would set the start address of a block to the end address of a subsequent block (i.e., $s_{i+1} = e_i$), would process chunks at the borders incorrectly (e.g., compared to non-parallelized processing of S). To produce consistent chunk boundaries in the parallelized and non-parallized version, we could move the starting point into the range of a preceding block. This would create an overlap which gives the rolling_hash a chance to re-synchronize. We could also run over the end e_i until we identify a match with the leading chunks of its successor (i.e., block processed by P_{i+1}).

EVALUATION ROLLING HASH. Our prototype implementation uses the producer-consumer paradigm. The image is read as a stream of bytes by the main thread. Depending on the amount of cores the first `image_size/cpu_num` bytes are read. Those bytes are passed to a thread which performs the rolling hash algorithm and hashes identified blocks. In parallel, the next (`image_size/cpu_num`) + `overlap` bytes are read and processed by another thread. In Table 7.2 times needed to process a 2 GiB image into block hashes with and without threading are shown. In the case of multi-threading, the time needed for resynchronization is already included. Obviously the needed CPU time will increase since it is spread over multiple threads. The actual elapsed time is remarkably lower. A speedup of approximately 30 seconds can be

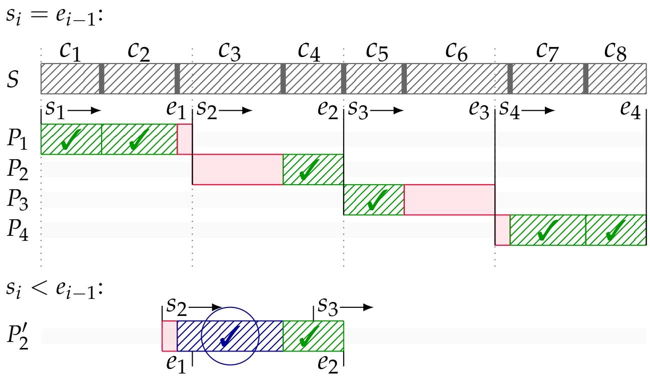


FIGURE 7.4: Introducing wrong blocks at the borders of image blocks

	Singlethread	Multithread (8 Threads)
Real	43.82 s	13.59 s
CPU	35.87 s	49.25 s

TABLE 7.2: Times of building/hashing blocks of an 2 GiB image in seconds (s).

achieved. This increases the processing speed by a factor of 3.2 by keeping consistent block boundaries. The query could be parallelized as well, but was not implemented in the course of this work.

7.2.4 *Theoretical Extensions*

This subsection will continue with an analysis of possible extensions which have not been integrated.

NEW FILE INSERTION IN HBFT. Adding new elements to a hbft is considerably easy, as long as the tree does not reach a critical load factor. A naive approach is the initial design of an oversized tree structure to create additional empty leaf nodes. A parameterization always has to consider the impact on the overall false positive rate. The new files can then be split into blocks and hashed into the tree. The already introduced *Global-filter* can be used to filter multihits with low dependencies. Therefore, only the original *global* and *multi* Bloom filters have to be updated and saved after a finished session. With a further growing amount of additional files being inserted, the empty leaf node pool will starve and the false positive rate for the structure will become unacceptable. At this point the database needs to be resized and rebuilt. Original block hashes can be saved to disk to shorten the build time. However, in many cases, this suggestion is infeasible. Adding additional files to the tree requires careful and storage-intensive pre-planning. Most of the times the database is optimized for a given file set. Introducing empty leaf nodes possibly adds additional levels to the tree. This pessimistic growth would slow down the lookup phase and adds memory overhead which could indeed never be required.

FILE DELETION IN HBFT. Hashes cannot be deleted from Bloom filters (except Counting Bloom filters). This in turn leads to the major drawback that hbft structures need to be partially rebuilt without the deleted file. In order to delete a specific file from the data structure, all related nodes from a leaf node up to its final root

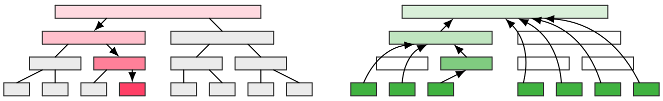


FIGURE 7.5: Deletion of elements in a *hbft* could be performed in two steps. First: delete all Bloom filters which were influenced by the deleted file in a top-down or bottom-up approach. Second: Re-insert block hashes affected by the deletion in a bottom-up approach.

node are affected. Such affected filters need to be deleted and re-populated again. Also file chunks which are represented by the affected nodes need to be re-inserted again. Lastly, Concerning the depicted tree structure in Figure 7.5, every file hash is needed since the root filter holds the block hashes for every file in the set. Splitting the root node into several filters would reduce the amount of recreated hashes from scratch. During a lookup, this would requiring horizontally processing a sequence of root-filters first. Figure 7.5 describes the problem of deleting files in a hbft structure. Deleting a file from the tree comes with considerable effort and computational overhead.

7.3 EVALUATION

The following performance tests focus on a runtime comparison between hashdb, hbft and fhmap. Each phase ranging from creating a database to the actual lookup is measured individually. Beside the overall *Memory Consumption* (7.3.2), we consider three major phases: *Build Phase* (7.3.3), *Deduplication Phase* (7.3.4) and *Lookup Phase* (7.3.5). The assessment of required resources and performance limitations of the candidates should respect the proposed environmental conditions. In particular, the considered techniques are scaled for specific environments where hashdb explicitly targets large scale systems with multiprocessing capabilities. Even if the presence of an adequate infrastructure is a considerable assumption, we aim for similar evaluation conditions and therefore will limit resources (e.g. the number of processing cores). Again, it should be clear that hashdb as a single-level store clearly stands out compared to our memory-only candidates hbft and fhmap. However, we strive for a comprehensive comparison in our introduced use case by including a fully-equipped database with desirable features.

7.3.1 *Testsystem and Testdata*

TESTSYSTEM. All of the tests were performed on a laptop with Ubuntu 16.4 LTS using an underlying ext4 filesystem. The machine features an Intel I7 Processor with 2.2 GHz, 8 GB RAM, a built-in HDD, and a built-in SSD. After each evaluation run the memory and caches have been cleared in order to avoid runtime or storage benefits in a subsequent evaluation pass (i.e., we make use of a "cold" machine). Each test was repeated three times and the results have been averaged. Building and lookup phases are influenced by the underlying storage drive. A benchmark of both drives reported a read data rate of 128 MiB per second for the HDD and 266 MiB per second for the SSD. Thus, reading a 2 GiB file into memory takes approximately 16 seconds from the HDD and eight seconds from the SSD. We further used the SSD throughout the following tests. Both drives have been benchmarked using the Linux tool *hdparm*.

TESTDATA. Tests are performed on random data and synthetic images. The considered file set consists of 4,096 files. Each file has a size of 524,288 bytes resulting in an image of 2 GiB. The image was created by concatenating all files together. We depict a global blocksize b of 512 bytes for all candidates and all tests. This should lead to a comparable compression rate and equal treatment in different phases of processing. Since each file is a multiple of b in length, a fixed-size extraction of blocks will not have to cope with any alignment issues.

FURTHER EXTENSIONS. Fixed block size hashing was additionally implemented for hbft and fhmap to allow for a uniform comparison. However, the originally proposed chunk hashing function FNV-256 remained unchanged. FNV-1 is used for fhmap since the length of FNV-256 is not necessary. In case of the parallelized rolling hash all cores can be kept busy as long as the process of reading is faster than the actual processing of extracted chunks. If the wait time for I/O is slower than the processing time, only one thread will process the image blocks. Remaining cores will not be utilized and the block building is bottlenecked by a slow I/O bus. hashdb can operate multi-threaded in all reading steps. As introduced in the course of this work, hbft and fhmap feature multithreading in the block building phase only. To allow a comparison, timings for single-threaded usage of hashdb is given as well, where hashdb implements a check of available system cores. This function was temporarily altered to always return a value of one. However, the implementation uses a producer-consumer approach and will spawn an additional thread anyways. By the usage of *taskset* we finally forced the execution on a single core.

7.3.2 *Memory Consumption*

The three approaches feature different memory requirements. After ingesting the 2 GiB test set hashdb produces files totalling 405.9 MB on disk. Since there is no theoretical background to calculate the data structure's size in main memory, it is approximated using the top command. The in memory size of the structure is about 900 MB.

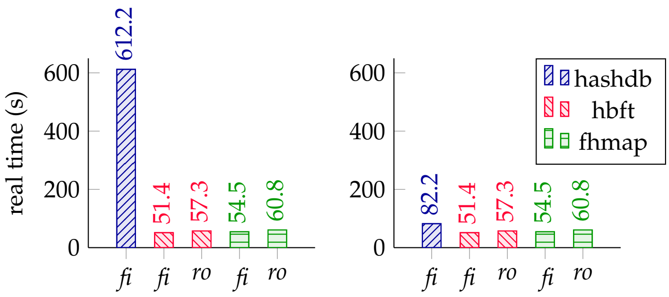
In the case of fhmap, the author[7] mentions some storage overhead for the handling of key-value-pairs. The overhead will be at a minimum of eight bits per entry and will be padded to match with the actual key length. Assuming a key length of 64 bit, then the overhead for fhmap would be 64 bit as well. Assuming the test set of 2 GiB with a block size $b = 512$, then the total amount of blocks n will be $4,194,304$. The total size s in main memory with a *load factor* of 0.5 would then result in $s = \frac{n \cdot 64\, bits \cdot 3}{0.5} \approx 200\, MiB$. The allocation size on disk would be halved to approximately 100 MiB caused by the *load factor*.

The size s of a hbft depends on various parameters and is mainly influenced by the data set size µ, the block size b, and a desired false positive rate *fpr*. In our scenario we approximate the root filter size for the parameters µ=2 GiB, b=512 and *fpr*=10^{-6}. This would lead to an approximated root filter size of $m_1 = \mu * 2^{15.84} \approx$ 14 MiB. The tree consists of $\log_2(4096) = 12$ levels. Thus, the total amount of needed memory is approximately $12 \cdot 14$ MiB $= 168$ MiB. The size on disk will be approximately the same since the array and corresponding Bloom filters need to be saved. For further details of the hbft parametrization and calculation, we refer to Lillis, Breitinger, and Scanlon (2017).

[7] https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/ (last access 2021-08-01)

Technique	DISK	RAM
hashdb	405.9 MiB	900.0 MiB
fhmap	100.0 MiB	200.0 MiB
hbft	168.0 MiB	168.0 MiB

TABLE 7.3: Harddisk and memory consumption of hashdb, fhmap and hbft for processing 2 GiB of input data.



(A) Single thread

(B) Multiple threads (8)

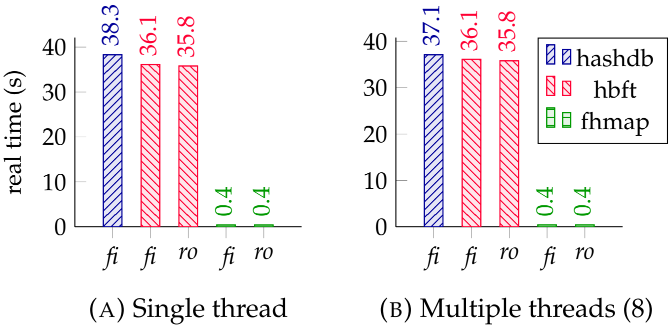
FIGURE 7.6: Build time performance of *hashdb* / *hbft* / *fhmap*. In case of *hbft* and *fhmap* we consider fixed blocks (*fi*) and the extraction per rolling_hash (*ro*).

An overview of the required storage for each technique can be seen in Table 7.3. In conclusion, hbft is the most memory-efficient approach, followed by fhmap and hashdb. Nevertheless, it should be noted that only hashdb is able to work with databases which do not completely fit into RAM. In contrast hbft and fhmap will only work if the databases fit into RAM.

7.3.3 *Build Phase*

The creation of a database consists of several steps including the initialization of the different structures for data storage and handling. The extraction of blocks by splitting an input stream is considered for fixed blocks (similar to hash-based carving) or varying blocks (similar to Approximate Matching). In detail, we integrated a fixed-block extraction (*fi*) and the extraction per rolling_hash (*ro*) for hbft and fhmap. Afterwards, in all cases the blocks are hashed. In the case of hashdb MD5 is used while hbft and fhmap use FNV-2561 and FNV-1 respectively.

The overall runtime for a single-threaded execution is presented in Figure 7.6a. Results for both block building approaches are displayed and evaluated in the case of hbft and fhmap. The high discrepancy of runtime in the case of hashdb is also caused by the setup of its metadata and relational features. The high difference from CPU to real time is related to read and write operations. Figure 7.6b shows the results for a multi-threaded execution (eight threads). The timings for the hbft and fhmap block building algorithms keep constant since multithreading is not yet implemented in the building phase. With the utilization of eight threads, hashdb cuts down its processing time tremendously. However, the approach is still slower than both block building algorithms of hbft and fhmap.



(A) Single thread (B) Multiple threads (8)

FIGURE 7.7: Deduplication performance of *hashdb* / *hbft* / *fhmap* with zero occurring multihits in a set.

7.3.4 *Deduplication Phase*

Recalling the utilized set of random data, randomly generated data does not feature any multi hits and thus, nearly all of the extracted chunks result in unique hits. Our considered version of hashdb allows deduplication of an existing database per configuration. The associated counter value for all hashed blocks are checked and all blocks with a value higher than one are deleted. The remaining chunks are written to a new database which finally ensures unique matches during lookup. The size of the newly established database stays the same. The introduced and implemented multihit mechanisms of hbft and fhmap in Section 7.2 are executed in memory only. The runtime results of the deduplication phase are displayed in Figures 7.7a and 7.7b.

Timings do not differ remarkably for single- or multi-threaded scenarios. The deduplication procedure for hashdb is slightly slower since it needs to read the database from disk first. As the *rolling hash* produces less blocks than a fixed-size extraction, the overall amount of inserted chunks decreases and thus, the runtime of deduplication improves. The fast deduplication time of fhmap is caused by three facts: First, there are no special structures which have to be additionally set up or evaluated. Second, the deduplication happens in the building phase as well, so there is no clear separation in building and deduplication for hash tables. Lastly, the random set does not feature any multi hits. hashdb and hbft need to process their temporary databases and filter out multi hits before inserting unique chunks in the actual database. In the case of fhmap, previously marked multihit-entries are simply deleted from the database.

7.3.5 *Lookup Phase*

The lookup consists of splitting an image into blocks, hashing those blocks, and querying them against the database. As we are only interested in efficiency (but not in detection performance), we make use of a simple approach to simulate full and partial detection scenarios. We create four different images which are queried against the databases. All images have a fixed size of 2 GiB. Each of the four images is constructed to match either 100 %, 75 %, 50 %, or 25 % of the database. Again we point out that the different file matching sizes are only used to investigate the efficiency behaviour dependent upon different matching rates, i.e., the matching rate is the input parameter. Images with matching rates below 100 % are partially filled with random bytes to reach the desired size of 2 GiB. The size of every inserted file

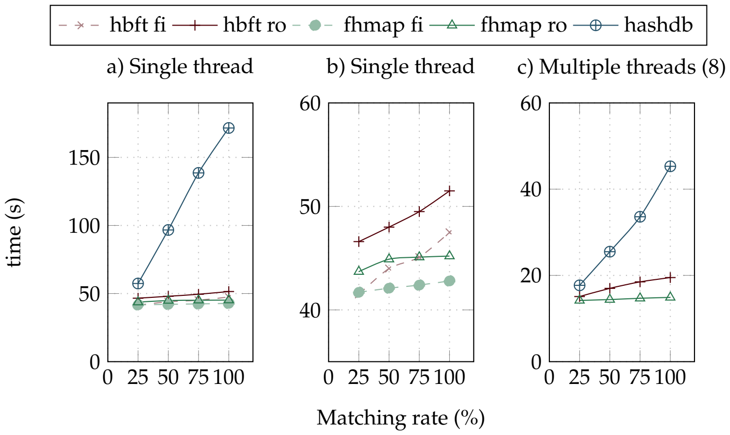


FIGURE 7.8: Lookup performance evaluation (real time).

is a multiple of b=512 bytes. Thus, images are crafted which do not cause alignment issues for a fixed-block extraction. It has to be considered that the rolling hash produces less blocks which additionally vary in size.

Results of the benchmark are shown in Figure 7.8. In Figure 7.8a and Figure 7.8b the lookup performance in the case of single-threaded evaluations is shown. Figure 7.8c displays the parallelized version with a total amount of eight running threads.

As shown, fhmap features the fastest lookup followed by hbft and lastly hashdb. The results underline the performance of fhmap in all cases and the impact of fixed blocks, in contrast to the overhead caused by computing a rolling hash. A significant speeding up is gained by our proposed parallelization of the rolling hash. The plot shows stable lookup results for all matching rates with fhmap outperforming its competitors. Lookup times for hbft and fhmap increase slightly by a rising matching rate. The lookups of hashdb are higher due to its complex internal structure.

Pre-filters for hbft and hashdb speed up the lookup time for non-matches notably. In the case of hbft the root Bloom filter will rule out non-matches instantly. Otherwise, a query needs to inspect subsequent nodes, shown by the slightly increased lookup times for higher matching rates. If a chunk does not match *hashdb's* compressed *Hash Store* the actual database is not queried either. A *Hash Store* claims to have a false positive rate of 1 in 72 million with a database containing 1 billion hashes. However, every hash will be queried against this store first before searching the actual database. The presented flat hash map does not feature any pre-filtering so far. Each key will be queried against the database. Tests performed with different sized databases did not differ remarkably.

7.4 DISCUSSION

Our result is that fhmap is best in terms of runtime performance and applicability. hbft showed a comparable runtime efficiency in the case of lookups, but hbft suffers from pitfalls with respect to extensibility and maintenance. Finally, hashdb performs worst in case of a single core environment in all evaluation scenarios, however, it is the only candidate which offers full parallelization capabilities and transactional features.

In this work we discussed and evaluated three different implementations of artefact lookup strategies in the course of digital forensics. Several extensions have been proposed to finally perform a comprehensive performance evaluation of hashdb,

	hashdb	hbft	fhmap
Multithreading	++	0	0
Add Hashes	++	-	++
Remove Hashes	++	- -	++
Limited to RAM	++	-	-
Transactions	++	-	-
Persistent Database	++	+	+
Prefilter	+	+	0
False Positives	+	-	+
Memory Usage	-	+	+
Build Phase (Single)	-	++	++
Build Phase (Multiple)	+	++	++
Deduplication Phase (Single)	-	-	+
Deduplication Phase (Multiple)	-	-	+
Lookup Phase (Single)	-	++	++
Lookup Phase (Multiple)	0	++	++

TABLE 7.4: Final comparison of hashdb / hbft / fhmap in the case of performance and offered features.

hbft, and fhmap. We introduced concepts to handle multihits for hbft and fhmap by the implementation of deduplication and filtration features. Moreover, we interfaced fhmap with a rolling-hash-based extraction of chunks. For a better comparison with hashdb, we additionally parallelized the extraction of chunks.

Results show that fhmap outperforms hbft in most of the considered performance evaluations. While hbfts are faster than hashdb in nearly all evaluations, the concept introduces false positives by the utilized Bloom filters. Even if hbfts have small advantages in the case of memory and storage efficiency, their complexity, fixed parametrization, and limited scope of features make such an advantage negligible. However, specific use cases with tight memory constraints could make hbfts still valuable.

Discussions of hashdb in terms of performance should consider the underlying concept of single-level stores. Shifting the discussion to offered features and a long-term usage with an ongoing maintenance, hashdb and fhmap are more suitable. One thing to note is that hashdb is the only implementation that is able to deal with databases which do not fit into main memory. In addition, it supports transactional features.

In Table 7.4 a final comparison of all three candidates in terms of performance and supported features is given. The final overview underlines the trade-offs between the concepts, where fhmap shows a constant performance in most of the mentioned categories.

A concept similar to single-level stores for digital artefacts with stable results in all of the mentioned categories is desirable. Where most of the considered challenges rely on an high amount of engineering effort first, the direct integration of a multihit prevention into a single-level store could be an interesting field of research. Concepts to close the gap between performance-oriented memory-resistant lookup strategies and transactional databases are needed.

8

BINARY MATCHING - APX-BIN

8.1 INTRODUCTION

In this chapter we will further inspect the possible adaptation of the previously presented mrsh-mem scheme in the course of binary matching and advantages by the utilization of a multi-layered extraction scheme for code-related artefacts.

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, its applicability is often discussed and extended evaluations were necessary. Recent research (Pagani, Dell'Amico, and Balzarotti, 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 for 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 influence the capabilities of fuzzy hashing schemes. As shown in Figure 8.1, 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 is 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 adaptation of compiler flags). The introduced pitfalls only belong to the bare benign cases, where maliciously motivated obfuscation adds a new dimension of challenges that a scheme should deal with.

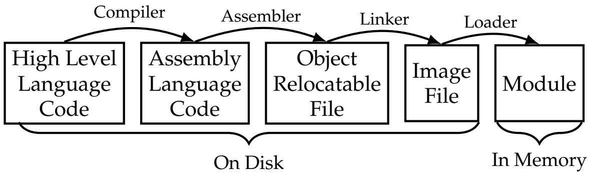


FIGURE 8.1: Compilation of a binary until it is loaded into memory.

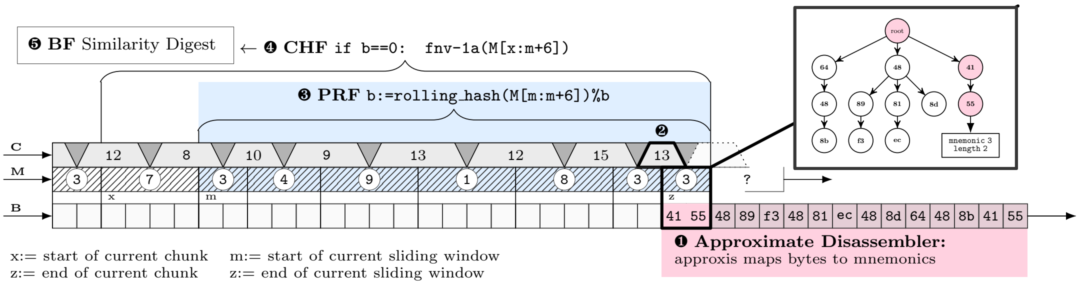

❺ BF Similarity Digest
← ❹ CHF if b==0: fnv-1a(M[x:m+6])
❸ PRF b:=rolling_hash(M[m:m+6])%b
❷
C
M
B
12
8
10
9
13
12
15
13
3
7
3
4
9
1
8
3
3
?
x
m
z
41 55 48 89 f3 48 81 ec 48 8d 64 48 8b 41 55
root
64
48
41
48
89
81
8d
55
8b
f3
ec
mnemonic 3 length 2
❶ Approximate Disassembler:
approxis maps bytes to mnemonics
x:= start of current chunk
z:= end of current chunk
m:= start of current sliding window
z:= end of current sliding window

FIGURE 8.2: 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).

CONTRIBUTIONS. In this chapter 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. In Section 2.1.1 we have already given a short overview of four different schemes and summarized related research.

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, Dell'Amico, and Balzarotti, 2018). Our approach is based on mrsh-mem, an algorithm introduced in Chapter 5. A short summary of the utilized evaluation scenarios and their key findings can also be found in Section 2.1.2.

In Section 8.2 we will summarize the key properties of 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, Dell'Amico, and Balzarotti (2018) by the definition of a provisional score-model in Section 8.2.1. Moreover, our pre-evaluation underlines the difficulty of withstanding all introduced challenges. In Section 8.2.2 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 8.3. As demonstrated, our proposed candidate outperforms four competitive fuzzy hashing implementations in most of the considered tests. In addition, we further extended the originally proposed scenarios by a set of new evaluation binaries, presented in Section 8.3.4. However, the overall strength of the approach is the stable inference and matching capabilities across all scenarios. Finally, we conclude this chapter and postulate future improvements in Section 8.4.

8.2 INTERFACING APPROXIMATE MATCHING - MRSH-MEM

We will shortly summarize previous findings and newly introduced concepts. The original mrsh-v2 scheme is a member of Context-Triggered Piecewise-Hashing (Breitinger and Baier, 2012b) 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). The algorithm iterates over the byte stream in a sliding window fashion, rolls through the sequence byte-by-byte, and processes seven consecutive bytes at a time. A current window is hashed with a PRF which returns a value between 0 and b. This value defines the trigger point 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 CHF and the hashes are stored in a Bloom filter (Bloom, 1970).

The overall concept shows 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.

In the case of mrsh-mem the technique does not process the raw byte sequences (**B**), but rather a mapped (mnemonic) stream (**M**). For a better overview of the following explanations we refer to Figure 8.2. To map a varying sequence of bytes to a single representative ❶ we utilize the trained prefix tree of x86 and x64 ground truth assemblies of approxis. In a nutshell, the trained model returns a mnemonic and a corresponding instruction length.

The integration of an approximate disassembler empowers the mrsh-mem tool to not only map sequences of bytes to integerized mnemonics, but also provides the opportunity to differ between code and data. The authors proposed the additional extraction of bi-gram frequencies to support the decoding process. The concept of code detection relies on those extracted frequencies. Besides 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 is 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. A depicted default value of $b = 64$ should lower the impact of smaller fragmentations. As already mentioned, within the buffer **C** the values of confidence are stored for two subsequent mappings (bi-grams). 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 Figure 8.2, all elements are considered as code offsets, as the values range between 8 and 15.

In the course of mrsh-mem, chunks with at least 30 % of the mappings considered to be code offsets are filtered, where code offsets are 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*.

8.2.1 *Pre-Evaluation*

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

1. **Selection of chunks.** 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.

2. **Multi-layered extraction.** We not only extract the mapped buffer of mnemonics but 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 weigh 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 8.1. 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.

$$sim_{pre} = min\left(\frac{\left(\sum_{i=1}^{y} f_c(b_i)\right) \cdot 1.5 + \sum_{i=1}^{y} f_c(m_i)}{z}, 100\right),$$

$$\text{where } f_c(x) = \begin{cases} 1, & \text{if } \tau_{min} \leq x \leq \tau_{max} \\ 0, & \text{else} \end{cases} \text{ and}$$

$$\forall x, y \in \mathbb{R}\left(\min(x,y) = \frac{x+y-|x-y|}{2}\right). \quad (8.1)$$

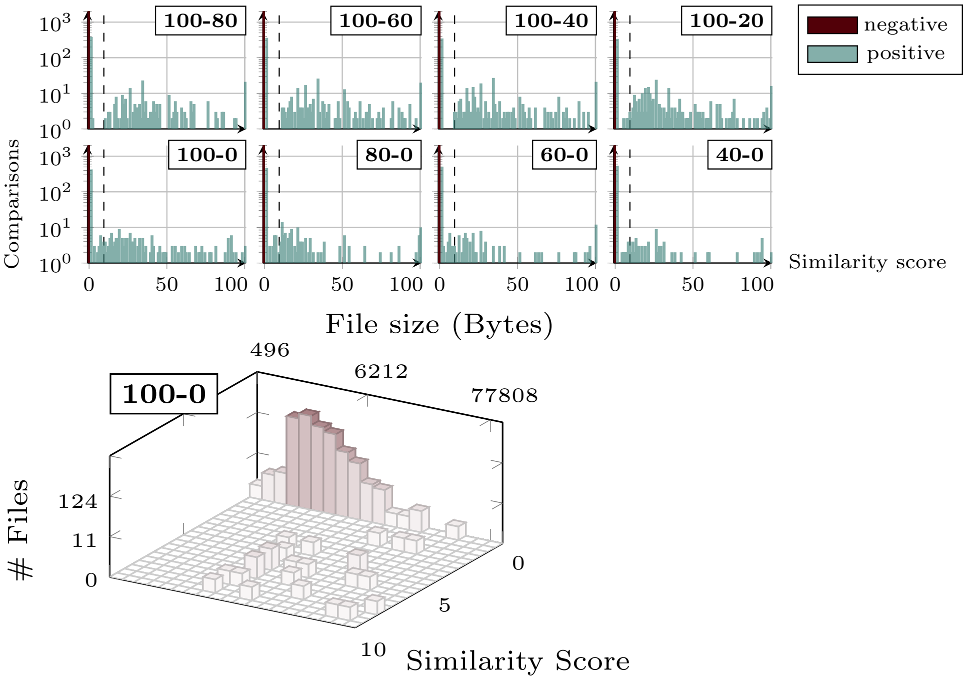


FIGURE 8.3: Scenario I. Library identification 1. Object-to-program - whole object file. (Labels specify depicted values of τ_{max}-τ_{min})

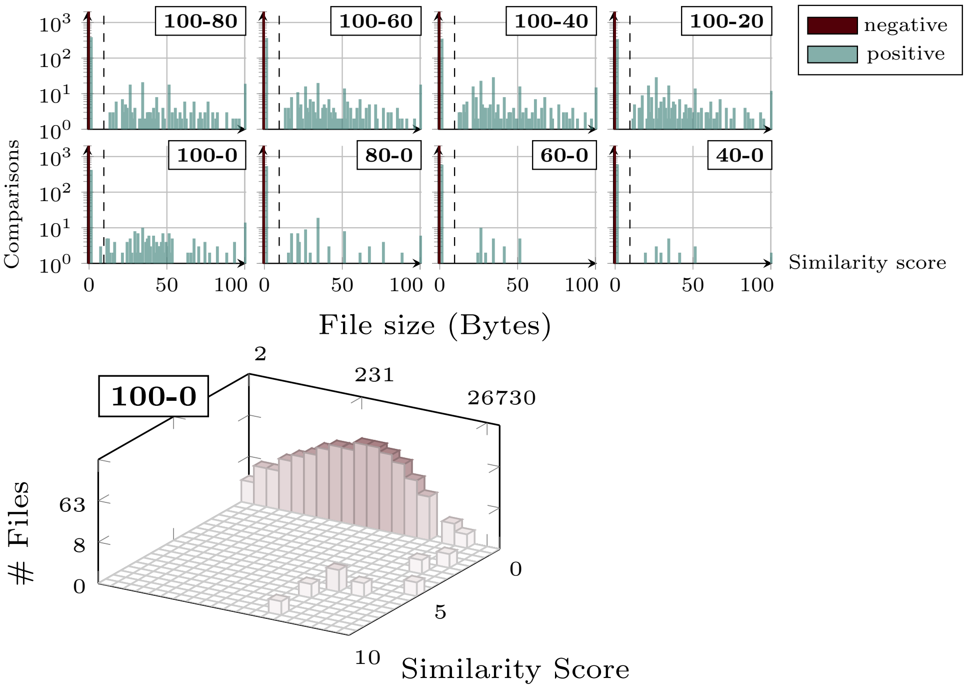


FIGURE 8.4: Scenario I. Library Identification 1. Object-to-program - text section. (Labels specify depicted values of τ_{max}-τ_{min})

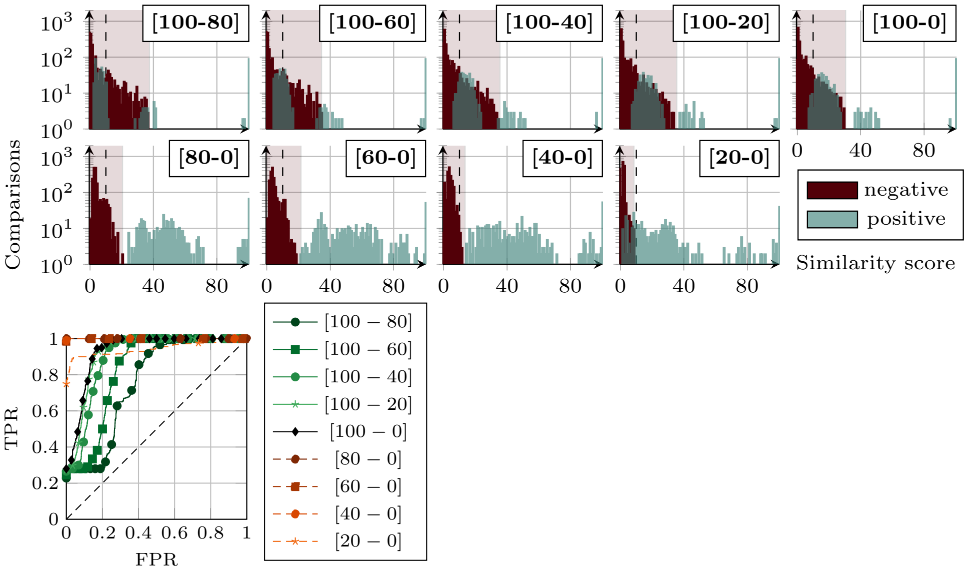


FIGURE 8.5: Scenario II. Re-Compilation 1. Optimization flags. Overview of score distributions for apx-bin with varying feature selection. (Labels specify depicted values of τ_{max} and τ_{min})

LIBRARY IDENTIFICATION. We inspect the influence of feature selection for the *Object-to-Program Comparison* of Scenario 1. In Figure 8.3 the results of matching a whole object file against a library are presented. As can 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 Figure 8.4 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 illustrate 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 Figure 8.3 and Figure 8.4 similarity scores of the non-matching libraries (negative) are always near zero. However, a remarkable amount of positive cases constantly yielded values near 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 illustrate 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 findings discussed where the extraction of constant data fragments out of the .rotdata section shows better results in the case of optimizations. As can be seen in Figure 8.5, the extraction of code fragments (i.e., $\tau_{max} = 100$, $\tau_{min} \geq 80$) showed ambiguous scores for positive and negative classes. In contrast, 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 can be seen in Figure 8.6, all parametrizations show a good performance with low similarities for 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 %.

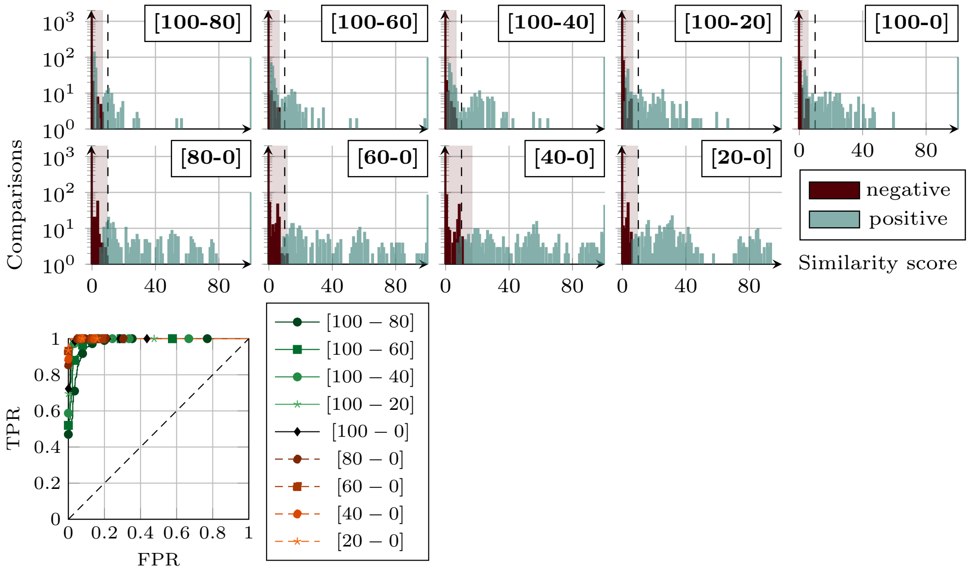


FIGURE 8.6: Scenario II. Re-Compilation 2. Compiler variation. Overview of score distributions for apx-bin with varying feature selection. (Labels specify depicted values of τ_{max} and τ_{min})

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.

8.2.2 *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 2.1.2, 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 actual 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 8.2). 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.

$$
\begin{aligned}
f_d(x) &= \begin{cases} 1, & \text{if } 0 \leq x \leq \tau_{min} \\ 0, & \text{else} \end{cases} \\
f_c(x) &= \begin{cases} 1, & \text{if } 100 \geq x \geq \tau_{max} \\ 0, & \text{else} \end{cases}
\end{aligned}
\tag{8.2}
$$

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 $\tau_{min} \leq c_i \leq \tau_{max}$).

$$
\begin{aligned}
sim_{bm} &= \frac{\gamma_d + \gamma_c}{2} \text{ ,where} \\
\gamma_d &= min\left(\frac{\left(\sum_{i=0}^{y-1} f_d(b_i) \cdot 2 + \sum_{i=0}^{z-1} f_d(m_i)\right) \cdot 1.5}{2 \cdot \sum_{i=0}^{n} f_d(c_i)}, 0.99 \right), \\
\gamma_c &= min\left(\frac{\sum_{i=0}^{y-1} f_c(b_i) \cdot 2 + \sum_{i=0}^{z-1} f_c(m_i)}{2 \cdot \sum_{i=0}^{n-1} f_c(c_i)}, 0.99 \right).
\end{aligned}
\tag{8.3}
$$

8.3 EVALUATION

We evaluate apx-bin in the course of the three introduced scenarios and by comparing them to the results from Pagani, Dell'Amico, and Balzarotti (2018). In Figure 8.7 the bimodal distribution of scores for several scenarios are shown. As can be seen,

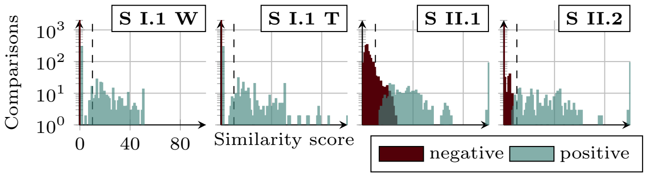


FIGURE 8.7: Scenario I and II: Overview of score distributions for apx-bin. (S: scenario; W: whole library; T: .text section).

	.o		.text		
Alg.	**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**

TABLE 8.1: Scenario I. Object-to-Program comparison 1. True and false positive rate compared to results Pagani, Dell'Amico, and Balzarotti (2018).

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.

8.3.1 *Library Identification*

Our prototype outperforms all of the other schemes in the course of identifying libraries in different object files or text sections. As can be seen in Table 8.1, 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 can be seen in Figure 8.8 apx-bin still performs better than most of the competing schemes on average.

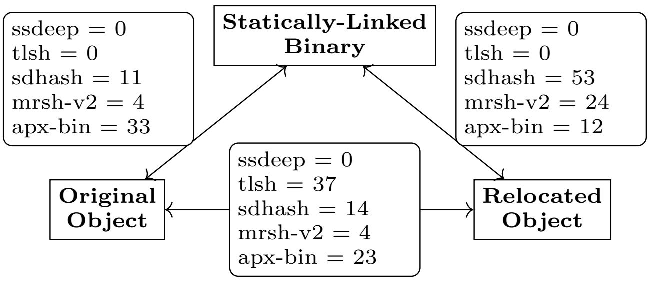


FIGURE 8.8: Scenario I. Impact of relocation 2. Average values of similarity for apx-bin compared to results Pagani, Dell'Amico, and Balzarotti (2018)

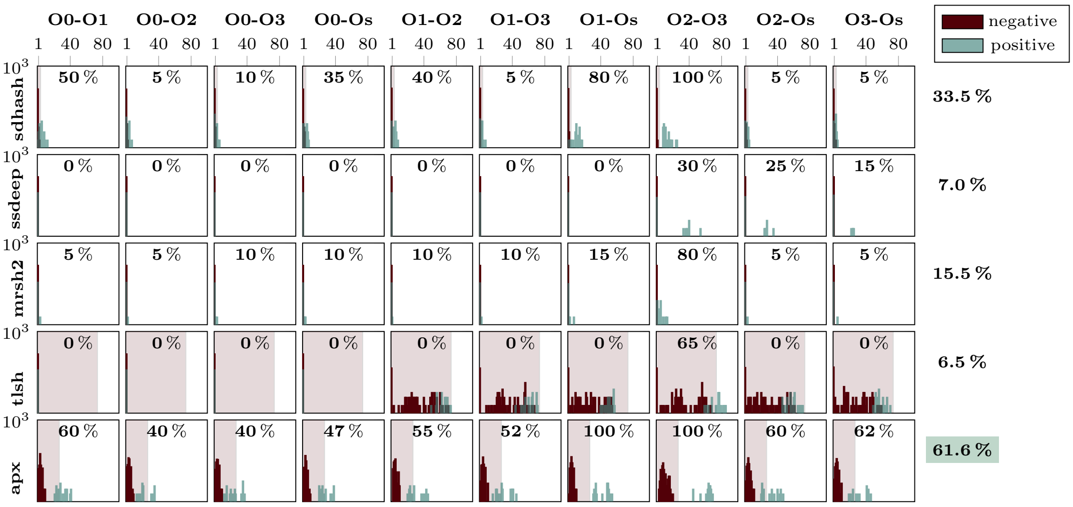


FIGURE 8.9: Scenario II. Re-Compilation 1. Optimization flags: competitive evaluation of apx-bin. Plots visualize the similarity between different optimization levels and for different schemes. Comparisons with the same configuration (i.e., same flags) are not shown. The value of percentage denotes the 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).

8.3.2 *Re-Compilation*

In Figure 8.9 and Figure 8.10 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 show 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.

As can be seen in Figure 8.9, 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 (Figure 8.10), apx can compete with the leading scores of mrsh-v2. In summary, the current model of apx-bin stabilizes the score values across different settings.

8.3.3 *Program Similarity*

The analysis of program similarity consists of small assembly differences and the random 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 instructions, the scores dropped after 10 swaps below 40 % (i.e., ssdeep, mrsh-v2 and sdhash). The score of tlsh stays almost close to 100 % and clearly outperforms competing schemes by its underlying concept.

In Figure 8.11 we show 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

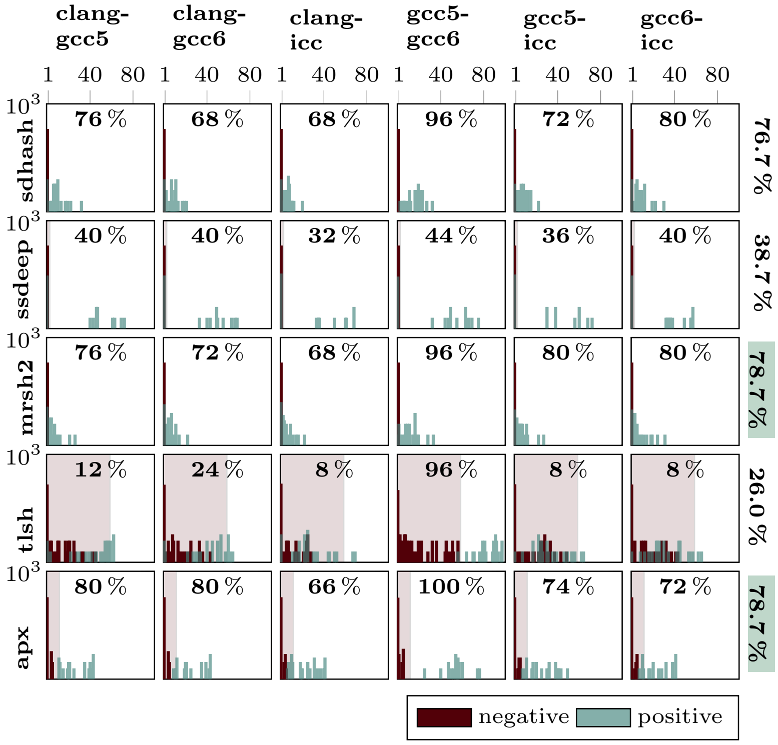


FIGURE 8.10: Scenario II. Re-Compilation 2. Different compilers: competitive evaluation of apx-bin. (y axis denotes the number of matches from 0 to 1000; x axis denotes the similarity score)

plot clearly visualizes the strength of the proposed score-model by additionally weighting the extracted chunks of data. The influence of instruction swapping modification or the insertion of NOP instructions impacts 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 8.2, apx-bin again shows stable results for all three cases. As mentioned by (Pagani, Dell'Amico, and Balzarotti, 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.

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 8.2: Scenario III. Minor source code modifications. Overview of ranging similarities.

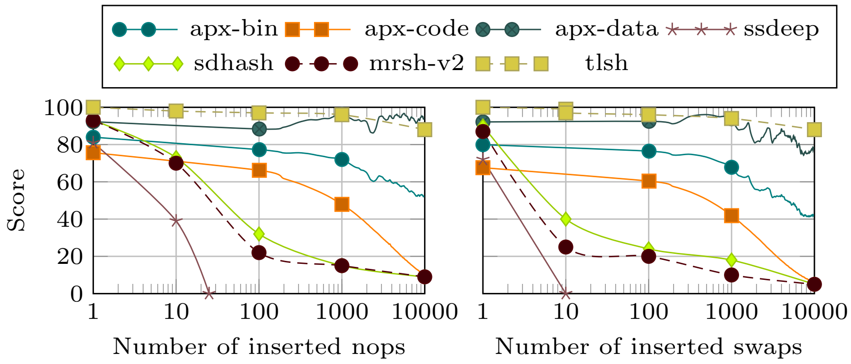


FIGURE 8.11: Scenario III. Small assembly differences 1. Evaluation after the insertion of random NOPs and the swapping of instructions: apx-bin compared to results from Pagani, Dell'Amico, and Balzarotti (2018).

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

TABLE 8.3: Scenario III. Source code modifications on malware 3. Modifications applied on the Mirai (M) and Grum (G) family. Overview of similarity scores.

8.3.4 *Extended Scenarios*

Inspecting the scenario of inserted assembly instructions previously proposed by Pagani, Dell'Amico, and Balzarotti (2018), we further extended existing approaches by an additional set of test binaries. Considering the increasing amount of NOPS inserted into a binary, someone could argue that tlsh is very resistance against this type of obfuscation, as the current test binaries only considers no-operations represented by the same byte-sequence. Thus, only a small set of specific byte-patterns gets additionally injected into a new binary. Especially, approaches like tlsh should be very resistant against those modifications. In contrast, inserting a large amount of similar byte-patterns has a considerably large impact on approaches based on CTPH (see Figure 8.11 - NOP insertion example).

The intel x86-instruction set offers a variety of different multi-byte-NOP instructions supported by the 64 bit architecture and partially supported by the 32 bit architecture (see Table 8.4). They are mainly used to implement memory alignment. Extracted from the *Intel Architectures Software Developer's Manual*[1] a (multi-byte) NOP does not impact the machine context and does not alter the content of a register (except the EIP register) or issue a memory operation. A no-operation could occupy one to nine bytes (see Table 8.4). Considering the insertion of multi-byte NOPs as a fairly simple technique of binary obfuscation, it is good to underline the susceptibility of different Approximate Matching techniques.

To extend the existing set of binaries, we utilized the Obfuscator-LLVM Project (Junod et al., 2015). The different obfuscation passes are also utilized by malware authors, even if most of the current techniques could be reversed by the utilization

[1] https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html (last access 2021-08-01).

Length	Assembly	Byte Sequence
2 bytes	`66 NOP`	`66 90H`
3 bytes	`NOP DWORD ptr [EAX]`	`0F 1F 00H`
4 bytes	`NOP DWORD ptr [EAX + 00H]`	`0F 1F 40 00H`
5 bytes	`NOP DWORD ptr [EAX + EAX*1 + 00H]`	`0F 1F 44 00 00H`
6 bytes	`66 NOP DWORD ptr [EAX + EAX*1 + 00H]`	`66 0F 1F 44 00 00H`
7 bytes	`NOP DWORD ptr [EAX + 00000000H]`	`0F 1F 80 00 00 00 00H`
8 bytes	`NOP DWORD ptr [EAX + EAX*1 + 00000000H]`	`0F 1F 84 00 00 00 00 00H`
9 bytes	`66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]`	`66 0F 1F 84 00 00 00 00 00H`

TABLE 8.4: Currently recommended multi-byte sequence of NOP instructions.

Argument	Description
noop	Insert a no-operation instruction.
noop_prob	Probability that a NOP inserted after an ordinary instruction (default: 50%).
noop_loop	Maximum amount of no-operation within a basic-block (default: 10).
sub	Swap simple instructions.
sub_loop	Amount of iterations for the sub runs per function (default: 1).
fla	Perform Control flow flattening.
bcf	Insert Bogus control flow.
bcf_prob	Probability that a bcf gets inserted into a basic block (default: 30%).
bcf_loop	Amount of changes performed for a functions (default: 1).
split	Split basic blocks.
split_num	Amount of splits per basic block (default: 2).
strenc	Encrypt strings.
jumpcond	Replace unconditional jumps with conditional jumps.
jumpcond_prob	Probability that an unconditional jump will be replaced (default: 50%).
dead	Insert dead code.
dead_prob	Probability that deadcode inserted at the end of a basic block (default: 50%).

TABLE 8.5: Overview of different modification arguments for the obfuscated-binary generation tool.

of advanced static analysis[2]. The modular structure of LLVM allows the realization of modifications and transformations in the frontend or the backend of the system: for example, to realize compiler-assisted code randomization (CCR) in the frontend (Koo et al., 2018), implement anti-disassembler techniques (Jämthagen, Lantz, and Hell, 2013) or insert machine-specific assembly instructions via the backend.

MODIFICATIONS. We extended the current backend for x86 and implemented interfaces for the generation process being controllable by a parametrized configuration file. We re-used, utilized, and added different approaches to generate the additional evaluation binaries. The current implementation allows control of the execution of different and combinable modification passes. As can be seen in Table 8.5, the recently introduced modifications consist of eight different categories. We re-used the following existing approaches: NOP Insertion, Instruction Swapping, Replacement of Simple Instructions, Control Flow Flattening, Bogus Control Flow, Splitting of Basic Blocks, and the Encryption of Strings[3]. We extended and added the following approaches: Multi-Byte NOP Insertion, Insertion of Dead-Code, and Replacement of Non-Conditional Jumps with Conditional Jumps. To implement the Multi-Byte NOP support, we had to adapt the OLLVM backend. For the implementation, we used LLVM 7.0.0 and the current available sources of the Obfuscator-LLVM project on GitHub.

[2] https://www.hex-rays.com/blog/hex-rays-microcode-api-vs-obfuscating-compiler/ (last access 2021-08-01).

[3] https://github.com/yazhiwang/ollvm-tll (last access 2021-08-01).

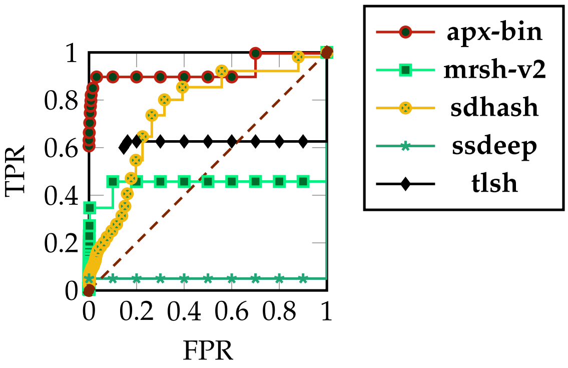


FIGURE 8.12: ROC (receiver operating characteristic) visualization of the comparison results of all modified binaries with the own class binaries (same application modified and unmodified) and comparison results with other class binaries (modified binaries of other class).

EXTENDED EVALUATION RESULTS. As can be seen in the Receiver Operating Characteristic (ROC) curve displayed in Figure 8.12, comparing binaries build with different obfuscation passes in use, the overall performance of apx-bin showed very good and stable results compared to the four competing schemes. Sdhash again proved its robustness, even in the light of major differences of two source-related binaries. Tlsh showed relatively stable results, however, it suffers from several high false positives. Classical CTPH-based approaches again suffer from the different types of modifications on a byte-level. Especially, ssdeep underlines its vulnerability in the case of those modifications.

In Figure 8.13, a detailed overview of the extended set of evaluation binaries is given. Each line represents a different obfuscation pass which has been previously applied on the evaluation binaries. The first five columns represent the five different schemes evaluated against the 32bit evaluation binaries and the next five columns represent the five different schemes evaluated against the 64bit evaluation binaries. The displayed bimodal distributions of the scores per columns underline the significant differences in terms of the different schemes. Again, we display the True Positive Rate (TPR) per set of binaries by accepting zero False Positive (FP) matches. Thus, per cell we yield the TPR for a specific set of evaluation binaries by depicting the considered threshold via the highest FP for the current class of evaluation binaries (highest occurring FP in current cell). In addition, we yield the TPR by depicting the highest FP as threshold over all classes of evaluation binaries (highest occurring FP in the current column). The results of tlsh in the case of multi-byte NOP insertions for 64bit binaries backed our assumptions: The approach also suffers from the high amount of additions, which now, in the light of the additions of multiple different byte-sequences, could not be damped by the scheme.

8.4 CONCLUSION

In this chapter 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, a derivative of mrsh-v2, we could outperform several of the existing approaches in different scenarios. Our adaptations mainly rely on mapping a present sequence of bytes into an integerized representation, utilizing a trained prefix-tree. The integrated feature of discriminating code

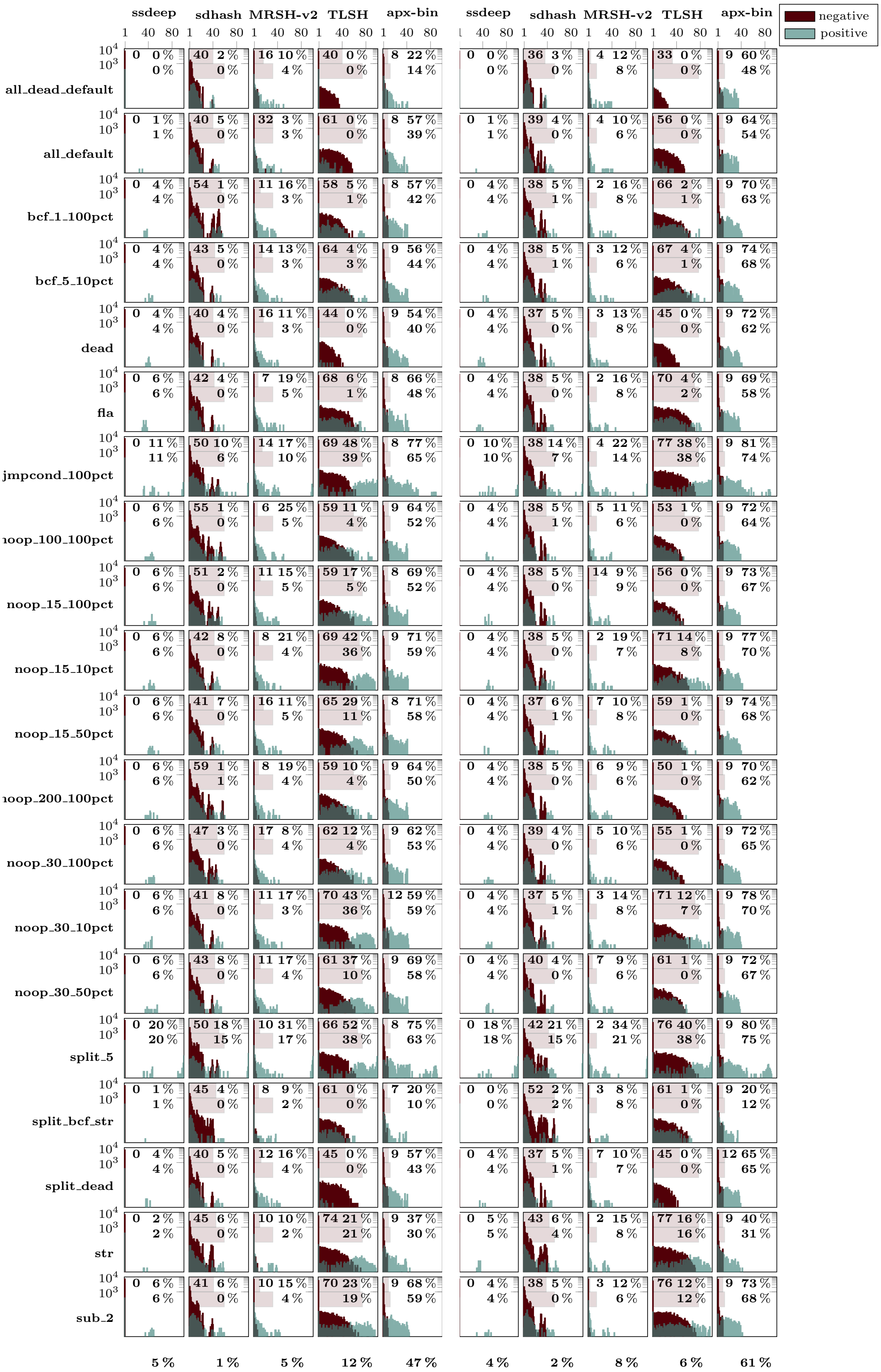
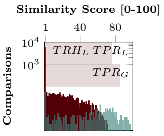


Figure 8.13: Results per modification of 32bit (left) and 64bit (right) binaries. TRH_L: Highest FP for a specific modification and scheme (highest FP in cell). TPR_L: TPR with zero FPs for modification (TPR for cell and highest FP in cell). TPR_G: TPR with zero FPs for all modifications and specific scheme (TPR for current cell and highest FP of whole scheme column).



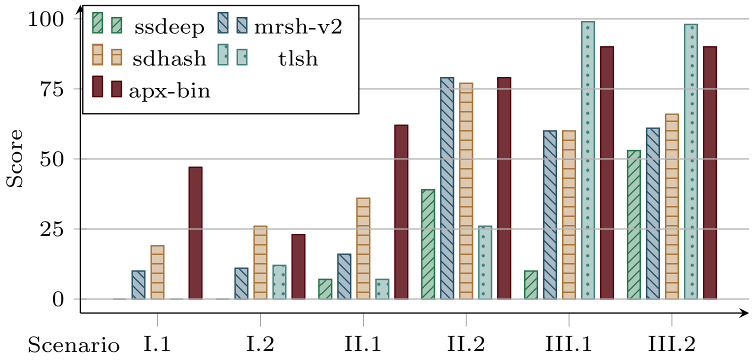


FIGURE 8.14: Overview of capabilities of apx-bin compared to existing techniques in the different scenarios. Scenarios with multiple scores have been averaged.

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 enables 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 Figure 8.14).

Besides 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 current scheme should yield a compact digest, e.g., by the utilization of multiple Bloom filters. The comprehensive set of scenarios proposed by Pagani, Dell'Amico, and Balzarotti (2018) are a valuable contribution and should be further extended and formalized. By the introduction of additional evaluation scenarios in Chapter 8.3.4 we further underlined this need. An uniform evaluation platform with widespread scenarios within an adversarial environment is still required (Oliver, Forman, and Cheng, 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 whether 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 *byte-wise* Approximate Matching.

9

CONCLUSION

This thesis addresses the applicability and transferability of concepts of Approximate Matching to the domain of memory carving and artefact extraction. More formally, we addressed the question of whether it is possible to strengthen main memory analysis by successfully transferring concepts of Approximate Matching to this domain. Besides different practical contributions, this thesis overall answered the following research questions:

RQ1: *Which memory analysis methods have been proposed and what are their main properties in terms of type, scope, and context of application?*

For a better understanding of the application domain, a general overview of the field of memory forensics was first given by categorizing more than 140 publications. The given *systematization of knowledge* underlines the ambitious efforts to establish and strengthen techniques of memory forensics in the field of digital forensics. Whereas structured analysis will highly likely further dominate the field, robust techniques of memory carving and unstructured analysis will still be required and furthermore be integrated in the processing pipelines of structured interpretation. Less research considers the application of Approximate Matching as a carving component. We identified relevant and related research, highlighting several pitfalls and challenges to be considered in own research.

RQ2: *What are applicable and interfaceable approaches for differentiating code and data applied on a considerably large and unstructured bulk of data?*

RQ3: *Under which premise could we interface or extend Approximate Matching as a Memory Carving technique? Which properties need to be respected and should be considered in further implementations and parametrization?*

As the application of bytewise Approximate Matching concepts obviously suffers from the alterability of artefacts stored in memory, we first addressed the problem of efficiently carving code and data fragments with an additional step of normalization. The approach could also be used to distinguish between architectures, compilers, and compiler versions. The proposed technique called *approxis* and its integrability into existing schemes has been further discussed by the introduction of a new scheme called *mrsh-mem*, a derivation of the mrsh family. We showed the feasibility of our approach by comparing a memory dump against code fragments gained from different resources. Our current prototype achieves a good computational performance, without the usage of any parallelization.

RQ4: *What are suitable candidates for storing extracted artefacts, and how is it possible to solve important aspects like common-block-handling?*

Discussing the adaptation of existing schemes also led to the problem of storing large amounts of artefacts and the inspection of suitable candidates for storing a large corpus of hash-based artefacts with additional features like common-block-handling. Often denoted as Database Lookup Problem (DLP), we tackled the decision by reassessing three different candidates as possible chunk databases. Moreover, we introduced several extensions to the considered storage systems, especially in terms of the recently introduced hierarchical Bloom filter trees (HBFT). Those required extensions enabled us to discuss and compare the strongly differing approaches with each other. We finally proposed the utilization of a modified *flat hash map*.

RQ5: *Is it possible to improve or replace existing contextual trigger functions of common Approximate Matching with better suited techniques?*

We addressed the possible replacement of chunk extraction methods often used by different CTPH schemes (i.e., PRF-based extraction) for the specialized task of carving code-related fragments. We therefore inspected in detail two signature-based concepts which have been recently proposed in the field of binary analysis. We discussed their applicability in our context and finally evaluated the two strongly modified concepts in detail, considering the limited scope of preprocessing. Finally, we proposed a simplified *weighted-prefix tree* (WPT) as replacement of the state-of-the-art PRF-based extraction functions.

RQ6: *Can we transfer the newly introduced concepts to the field of binary matching?*

We discussed additionally proposed scheme adaptations in the field of binary matching and inspected the capabilities of a multi-layered feature extraction phase. Introduced and built upon our previous findings, we propose a new scheme called *apx-bin*, considered and evaluated for the task of binary matching. We evaluated our scheme against several other Approximate Matching schemes and utilized different evaluation scenarios recently proposed by other researchers. Our scheme shows competing results across all considered scenarios. We additionally introduced a new set of evaluation binaries, attacking several weaknesses of the considered schemes.

Summarized and illustrated in Figure 9.1, several contributions have been made: We introduced a new branch of SPHF functions. The introduction of a new preprocessor *approxis* finally led to the creation of a new scheme *mrsh-mem*. In addition to its first version, relying on the original pseudo-random function (PRF) responsible

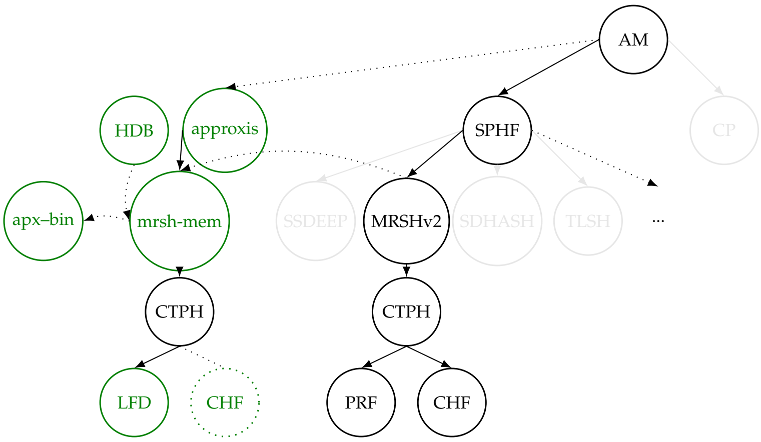


FIGURE 9.1: Overview of research path and contributions.

for the chunk boundary definition, we introduced a proper linear function detection approach (LFD) as a replacement in a subsequent step. The new additions led to a new scheme *apx-bin*, usable for the task of binary matching. In addition, we made several additions to existing hash databases (HDB), especially in terms of the newly introduced hierarchical bloom filter trees (HBFT).

BIBLIOGRAPHY

Adelstein, Frank (2006). “Live forensics: diagnosing your system without killing it first”. In: *Communications of the ACM* 49.2, pp. 63–66.

Al-Sharif, Ziad A, Hasan Bagci, Aseel Asad, et al. (2018). “Towards the memory forensics of ms word documents”. In: *Information Technology-New Generations*. Springer, pp. 179–185.

Ali-Gombe, Aisha et al. (2019). “DroidScraper: A Tool for Android In-Memory Object Recovery and Reconstruction”. In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pp. 547–559.

Aljaedi, Amer et al. (2011). “Comparative analysis of volatile memory forensics: live response vs. memory imaging”. In: *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*. IEEE, pp. 1253–1258.

Andriesse, Dennis, Asia Slowinska, and Herbert Bos (2017). “Compiler-Agnostic Function Detection in Binaries”. In: *IEEE European Symposium on Security and Privacy*.

Andriesse, Dennis et al. (2016). “An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries”. In: *USENIX Security Symposium*.

Arasteh, Ali Reza and Mourad Debbabi (2007). “Forensic memory analysis: From stack and code to execution history”. In: *digital investigation* 4, pp. 114–125.

Astebol, Knut Petter (2012). “mvHash: a new approach for fuzzy hashing”. In:

Aumaitre, Damien (2009). “A little journey inside Windows memory”. In: *Journal in computer virology* 5.2, pp. 105–117.

Azab, Ahmad et al. (2014). “Mining malware to detect variants”. In: *2014 Fifth Cybercrime and Trustworthy Computing Conference*. IEEE, pp. 44–53.

Baier, Harald and Frank Breitinger (2011). “Security aspects of piecewise hashing in computer forensics”. In: *2011 Sixth International Conference on IT Security Incident Management and IT Forensics*. IEEE, pp. 21–36.

Balzarotti, Davide, Roberto Di Pietro, and Antonio Villani (2015). “The impact of GPU-assisted malware on memory forensics: A case study”. In: *Digital Investigation* 14, S16–S24.

Bao, Tiffany et al. (2014). “Byteweight: Learning to recognize functions in binary code”. In: USENIX.

Barabosch, Thomas et al. (2017). “Quincy: Detecting host-based code injection attacks in memory dumps”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, pp. 209–229.

Barmpatsalou, Konstantia et al. (2018). “Current and future trends in mobile device forensics: A survey”. In: *ACM Computing Surveys (CSUR)* 51.3, pp. 1–31.

Bauer, Johannes, Michael Gruhn, and Felix C Freiling (2016). “Lest we forget: Cold-boot attacks on scrambled DDR3 memory”. In: *Digital Investigation* 16, S65–S74.

Becher, Michael, Maximillian Dornseif, and Christian N Klein (2005). “FireWire: all your memory are belong to us”. In: *Proceedings of CanSecWest*, p. 67.

Beverly, Robert, Simson Garfinkel, and Greg Cardwell (2011). "Forensic carving of network packets and associated data structures". In: *digital investigation* 8, S78–S89.

Bhatia, Rohit et al. (2018). "Tipped Off by Your Memory Allocator: Device-Wide User Activity Sequencing from Android Memory Images." In: *NDSS*.

Bhatt, Manish and Irfan Ahmed (2018). "Leveraging relocations in ELF-binaries for Linux kernel version identification". In: *Digital Investigation* 26, S12–S20.

Bilar, Daniel (2006). "Statistical structures: Fingerprinting malware for classification and analysis". In: *Proceedings of Black Hat Federal 2006*.

Block, Frank and Andreas Dewald (2017). "Linux memory forensics: Dissecting the user space process heap". In: *Digital Investigation* 22, S66–S75.

— (2019). "Windows Memory Forensics: Detecting (Un) Intentionally Hidden Injected Code by Examining Page Table Entries". In: *Digital Investigation* 29, S3–S12.

Bloom, Burton H. (1970). "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Communications of the ACM* 13, pp. 422–426.

Boileau, Adam (2006). "Hit by a bus: Physical access attacks with Firewire". In: *Presentation, Ruxcon* 3.

Breitinger, Frank (June 1, 2014). "On the utility of bytewise approximate matching in computer science with a special focus on digital forensics investigations". PhD thesis. Technical University Darmstadt. published.

Breitinger, Frank and Ibrahim Baggili (2014). "File detection on network traffic using approximate matching". In:

Breitinger, Frank and Harald Baier (2012a). "Properties of a similarity preserving hash function and their realization in sdhash". In: *2012 Information Security for South Africa*. IEEE, pp. 1–8.

— (2012b). "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, Frank, Harald Baier, and Jesse Beckingham (2012). "Security and implementation analysis of the similarity digest sdhash". In: *First international baltic conference on network security & forensics (nesefo)*.

Breitinger, Frank, Harald Baier, and Douglas White (2014). "On the database lookup problem of approximate matching". In: *Digital Investigation* 11, S1–S9.

Breitinger, Frank, Christian Rathgeb, and Harald Baier (2014). "An efficient similarity digests database lookup-A logarithmic divide & conquer approach". In: *The Journal of Digital Forensics, Security and Law: JDFSL* 9.2, p. 155.

Breitinger, Frank and Vassil Roussev (2014). "Automated evaluation of approximate matching algorithms on real data". In: *Digital Investigation* 11, S10–S17.

Breitinger, Frank, Georgios Stivaktakis, and Harald Baier (2013). "FRASH: A framework to test algorithms of similarity hashing". In: *Digital Investigation* 10, S50–S58.

Breitinger, Frank, Georgios Stivaktakis, and Vassil Roussev (2014). "Evaluating detection error trade-offs for bytewise approximate matching algorithms". In: *Digital Investigation* 11.2, pp. 81–89.

Breitinger, Frank et al. (2013). "mvHash-B - A New Approach for Similarity Preserving Hashing". In: *2013 Seventh International Conference on IT Security Incident Management and IT Forensics*, pp. 33–44.

Breitinger, Frank et al. (2014). "Approximate matching: definition and terminology". In: *NIST Special Publication* 800, p. 168.

Carrier, Brian D and Joe Grand (2004). "A hardware-based memory acquisition procedure for digital investigations". In: *Digital Investigation* 1.1, pp. 50–60.

Case, Andrew (2011). "De-anonymizing live CDs through physical memory analysis". In: *Blackhat DC Security Conference, Washington DC*.

Case, Andrew, Lodovico Marziale, and Golden G Richard III (2010). "Dynamic recreation of kernel data structures for live forensics". In: *Digital Investigation* 7, S32–S40.

Case, Andrew and Golden G Richard (2017). "Memory forensics: The path forward". In: *Digital Investigation* 20, pp. 23–33.

Case, Andrew and Golden G Richard III (2015). "Advancing Mac OS X rootkit detection". In: *Digital Investigation* 14, S25–S33.

— (2016). "Detecting objective-C malware through memory forensics". In: *Digital Investigation* 18, S3–S10.

Case, Andrew et al. (2008). "FACE: Automated digital evidence discovery and correlation". In: *digital investigation* 5, S65–S75.

Case, Andrew et al. (2010). "Treasure and tragedy in kmem_cache mining for live forensics investigation". In: *digital investigation* 7, S41–S47.

Case, Andrew et al. (2017). "Gaslight: A comprehensive fuzzing architecture for memory forensics frameworks". In: *Digital Investigation* 22, S86–S93.

Case, Andrew et al. (2019). "HookTracer: A System for Automated and Accessible API Hooks Analysis". In: *Digital Investigation* 29, S104–S112.

Case, Andrew et al. (2020). "Memory Analysis of macOS Page Queues". In: *Digital Investigation*.

Casey, Peter et al. (2019). "Inception: Virtual Space in Memory Space in Real Space–Memory Forensics of Immersive Virtual Reality with the HTC Vive". In: *Digital Investigation* 29, S13–S21.

Celis, Pedro, Per-Ake Larson, and J Ian Munro (1985). "Robin hood hashing". In: *Foundations of Computer Science, 1985., 26th Annual Symposium on*. IEEE, pp. 281–288.

Chan, E et al. (2009). "A framework for volatile memory forensics". In: *Proceedings of the16th ACM conference on computer and communications security*.

Chu, Howard (2011). "MDB: A memory-mapped database and backend for OpenLDAP". In: *Proceedings of the 3rd International Conference on LDAP, Heidelberg, Germany*, p. 35.

Cohen, MI (2008). "PyFlag–An advanced network forensic framework". In: *Digital investigation* 5, S112–S120.

Cohen, Michael (2017). "Scanning memory with Yara". In: *Digital Investigation* 20, pp. 34–43.

Cohen, Michael I (2015). "Characterization of the windows kernel version variability for accurate memory analysis". In: *Digital Investigation* 12, S38–S49.

Cozzie, Anthony et al. (2008). "Digging for Data Structures." In: *OSDI*. Vol. 8, pp. 255–266.

De Nicolao, Pietro et al. (2018). "ELISA: ELiciting ISA of Raw Binaries for Fine-Grained Code and Data Separation". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, pp. 351–371.

Dolan, Stephen (2013). "mov is Turing-complete". In: *Cl. Cam. Ac. Uk*, pp. 1–4.

Dolan-Gavitt, Brendan (2007). "The VAD tree: A process-eye view of physical memory". In: *digital investigation* 4, pp. 62–64.

— (2008). "Forensic analysis of the Windows registry in memory". In: *digital investigation* 5, S26–S32.

Dolan-Gavitt, Brendan et al. (2009). “Robust signatures for kernel data structures”. In: *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 566–577.

Dolan-Gavitt, Brendan et al. (2011). “Virtuoso: Narrowing the semantic gap in virtual machine introspection”. In: *2011 IEEE symposium on security and privacy*. IEEE, pp. 297–312.

Eagle, Chris (2008). *The IDA Pro Book: The Unofficial Guide to the World’s Most Popular Disassembler*. San Francisco, CA, USA: No Starch Press. ISBN: 1593271786, 9781593271787.

Enck, William et al. (2008). “Defending against attacks on main memory persistence”. In: *2008 Annual Computer Security Applications Conference (ACSAC)*. IEEE, pp. 65–74.

Eschweiler, Sebastian and Elmar Gerhards-Padilla (2012). “Towards Sound Forensic Acquisition of Volatile Data”. In: *Future Security Research Conference*. Springer, pp. 289–298.

Fan, Bin et al. (2014). “Cuckoo filter: Practically better than bloom”. In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, pp. 75–88.

Foster, Kristina (2012). *Using distinct sectors in media sampling and full media analysis to detect presence of documents from a corpus*. Tech. rep. NAVAL POSTGRADUATE SCHOOL MONTEREY CA.

Fowler, Glenn et al. (2011). “The FNV non-cryptographic hash algorithm”. In: *IETF-draft*.

French, David and William Casey (2012). “Fuzzy Hashing Techniques in Applied Malware Analysis”. In: *Results of SEI Line-Funded Exploratory New Starts Projects*, p. 2.

Garcia, Gabriela Limon (2007). “Forensic physical memory analysis: an overview of tools and techniques”. In: *TKK T-110.5290 Seminar on Network Security*. Vol. 207. Citeseer, pp. 305–320.

Garfinkel, Simson L and Michael McCarrin (2015). “Hash-based carving: Searching media for complete files and file fragments with sector hashing and hashdb”. In: *Digital Investigation* 14, S95–S105.

Gers, Felix A, Jürgen Schmidhuber, and Fred Cummins (1999). “Learning to forget: Continual prediction with LSTM”. In:

Graziano, Mariano, Andrea Lanzi, and Davide Balzarotti (2013). “Hypervisor memory forensics”. In: *International Workshop on Recent Advances in Intrusion Detection*. Springer, pp. 21–40.

Gruhn, Michael (2015). “Windows nt pagefile. sys virtual memory analysis”. In: *2015 Ninth International Conference on IT Security Incident Management & IT Forensics*. IEEE, pp. 3–18.

Gruhn, Michael and Felix C Freiling (2016). “Evaluating atomicity, and integrity of correct memory acquisition methods”. In: *Digital Investigation* 16, S1–S10.

Gruhn, Michael and Tilo Müller (2013). “On the practicability of cold boot attacks”. In: *2013 International Conference on Availability, Reliability and Security*. IEEE, pp. 390–397.

Gu, Yufei et al. (2012). “Os-sommelier: Memory-only operating system fingerprinting in the cloud”. In: *Proceedings of the Third ACM Symposium on Cloud Computing*, pp. 1–13.

Guilfanov, Ilfak (2012). *IDA fast library identification and recognition technology (FLIRT Technology): In-depth*.

Gupta, Vikas and Frank Breitinger (2015). "How cuckoo filter can improve existing approximate matching techniques". In: *International conference on digital forensics and cyber crime*. Springer, pp. 39–52.

Halderman, J Alex et al. (2009). "Lest we remember: cold-boot attacks on encryption keys". In: *Communications of the ACM* 52.5, pp. 91–98.

Hargreaves, Christopher and Howard Chivers (2008). "Recovery of encryption keys from memory using a linear scan". In: *2008 Third International Conference on Availability, Reliability and Security*. IEEE, pp. 1369–1376.

Harichandran, Vikram S, Frank Breitinger, and Ibrahim Baggili (2016). "Bytewise approximate matching: the good, the bad, and the unknown". In: *Journal of Digital Forensics, Security and Law* 11.2, p. 4.

Haruyama, Takahiro and Hiroshi Suzuki (2012). "One-byte modification for breaking memory forensic analysis". In: *Black Hat Europe*.

Hejazi, Seyed Mahmood, Chamseddine Talhi, and Mourad Debbabi (2009). "Extraction of forensically sensitive information from windows physical memory". In: *digital investigation* 6, S121–S131.

Heninger, Nadia and Hovav Shacham (2009). "Reconstructing RSA private keys from random key bits". In: *Annual International Cryptology Conference*. Springer, pp. 1–17.

Heriyanto, Andri, Craig Valli, and Peter Hannay (2015). "Comparison of Live Response, Linux Memory Extractor (LiME) and Mem tool for acquiring android's volatile memory in the malware incident". In:

Hilgers, Christian et al. (2014). "Post-mortem memory analysis of cold-booted android devices". In: *2014 Eighth International Conference on IT Security Incident Management & IT Forensics*. IEEE, pp. 62–75.

Hinton, Geoffrey E et al. (2012). "Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv preprint arXiv:1207.0580*.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: *Neural computation* 9.8, pp. 1735–1780.

Huebner, Ewa et al. (2007). "Persistent systems techniques in forensic acquisition of memory". In: *Digital Investigation* 4.3-4, pp. 129–137.

Inoue, Hajime, Frank Adelstein, and Robert A Joyce (2011). "Visualization in testing a volatile memory forensic tool". In: *Digital Investigation* 8, S42–S51.

Iyer, R Padmavathi et al. (2017). "Email spoofing detection using volatile memory forensics". In: *2017 IEEE Conference on Communications and Network Security (CNS)*. IEEE, pp. 619–625.

Jämthagen, Christopher, Patrik Lantz, and Martin Hell (2013). "A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries". In: *2013 Workshop on Anti-malware Testing Research*. IEEE, pp. 1–9.

Jin, Wesley et al. (2012). "Binary function clustering using semantic hashes". In: *Machine Learning and Applications (ICMLA), 2012 11th International Conference on*. Vol. 1. IEEE, pp. 386–391.

Junod, Pascal et al. (2015). "Obfuscator-LLVM–software protection for the masses". In: *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE, pp. 3–9.

Kazim, Abdullah et al. (2019). "Memory forensics: Recovering chat messages and encryption master key". In: *2019 10th International Conference on Information and Communication Systems (ICICS)*. IEEE, pp. 58–64.

Kiley, Matthew, Shira Dankner, and Marcus Rogers (2008). "Forensic analysis of volatile instant messaging". In: *IFIP International Conference on Digital Forensics*. Springer, pp. 129–138.

Koo, Hyungjoon et al. (2018). "Compiler-assisted code randomization". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, pp. 461–477.

Kornblum, Jesse (2006). "Identifying almost identical files using context triggered piecewise hashing". In: *Digital investigation* 3, pp. 91–97.

Kornblum, Jesse D (2007). "Using every part of the buffalo in Windows memory analysis". In: *Digital Investigation* 4.1, pp. 24–29.

Kornblum, Jesse D and CFIA ManTech (2006). "Exploiting the rootkit paradox with windows memory analysis". In: *International Journal of Digital Evidence* 5.1, pp. 1–5.

Lapso, Joshua A, Gilbert L Peterson, and James S Okolica (2017). "Whitelisting system state in windows forensic memory visualizations". In: *Digital Investigation* 20, pp. 2–15.

Latzo, Tobias, Julian Brost, and Felix Freiling (2020). "BMCLeech: Introducing Stealthy Memory Forensics to BMC". In: *Forensic Science International: Digital Investigation* 32, p. 300919.

Latzo, Tobias, Ralph Palutke, and Felix Freiling (2019). "A universal taxonomy and survey of forensic memory acquisition techniques". In: *Digital Investigation* 28, pp. 56–69.

Lee, Kyoungho et al. (2016). "Robust bootstrapping memory analysis against anti-forensics". In: *Digital Investigation* 18, S23–S32.

Lewis, Nathan et al. (2018). "Memory forensics and the Windows Subsystem for Linux". In: *Digital Investigation* 26, S3–S11.

Li, Yuping et al. (2015). "Experimental study of fuzzy hashing in malware clustering analysis". In: *8th Workshop on Cyber Security Experimentation and Test ({CSET} 15)*.

Libster, Eugene and Jesse D Kornblum (2008). "A proposal for an integrated memory acquisition mechanism". In: *ACM SIGOPS Operating Systems Review* 42.3, pp. 14–20.

Liebler, Lorenz and Harald Baier (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*. Springer, pp. 158–172.

— (2019). "Towards exact and inexact approximate matching of executable binaries". In: *Digital Investigation* 28, S12–S21.

Liebler, Lorenz and Frank Breitinger (2018). "mrsh-mem: Approximate matching on raw memory dumps". In: *International Conference on IT Security Incident Management and IT Forensics*. IEEE, pp. 47–64.

Ligh, Michael et al. (2010). *Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code*. Wiley Publishing.

Ligh, Michael Hale et al. (2014). *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons.

Lillis, David, Frank Breitinger, and Mark Scanlon (2017). "Expediting mrsh-v2 approximate matching with hierarchical bloom filter trees". In: *International Conference on Digital Forensics and Cyber Crime*. Springer, pp. 144–157.

Lin, Zhiqiang et al. (2011). "SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures." In: *Ndss*.

Lindenlauf, Simon, Hans Höfken, and Marko Schuba (2015). "Cold boot attacks on DDR2 and DDR3 SDRAM". In: *2015 10th International Conference on Availability, Reliability and Security*. IEEE, pp. 287–292.

Lipton, Zachary C, John Berkowitz, and Charles Elkan (2015). "A critical review of recurrent neural networks for sequence learning". In: *arXiv preprint arXiv:1506.00019*.

Maartmann-Moe, Carsten, Steffen E Thorkildsen, and André Årnes (2009). "The persistence of memory: Forensic identification and extraction of cryptographic keys". In: *digital investigation* 6, S132–S140.

McGregor, Patrick et al. (2008). "Braving the cold: New methods for preventing cold boot attacks on encryption keys". In: *Black Hat Security Conference*.

Müller, Tilo, Andreas Dewald, and Felix C Freiling (2010). "AESSE: a cold-boot resistant implementation of AES". In: *Proceedings of the Third European Workshop on System Security*, pp. 42–47.

Müller, Tilo, Felix C Freiling, and Andreas Dewald (2011). "TRESOR Runs Encryption Securely Outside RAM." In: *USENIX Security Symposium*. Vol. 17.

Müller, Tilo and Michael Spreitzenbarth (2013). "Frost". In: *International Conference on Applied Cryptography and Network Security*. Springer, pp. 373–388.

Müller, Tilo, Benjamin Taubmann, and Felix C Freiling (2012). "Trevisor". In: *International Conference on Applied Cryptography and Network Security*. Springer, pp. 66–83.

Okolica, James and Gilbert Peterson (2010a). "A compiled memory analysis tool". In: *IFIP International Conference on Digital Forensics*. Springer, pp. 195–204.

Okolica, James and Gilbert L Peterson (2010b). "Windows operating systems agnostic memory analysis". In: *Digital investigation* 7, S48–S56.

— (2011a). "Extracting the windows clipboard from physical memory". In: *digital investigation* 8, S118–S124.

Okolica, James S and Gilbert L Peterson (2011b). "Windows driver memory analysis: A reverse engineering methodology". In: *computers & security* 30.8, pp. 770–779.

Olajide, Funminiyi et al. (2012). "Identifying and finding forensic evidence on windows application". In: *Journal of Internet Technology and Secured Transactions, ISSN*, pp. 2046–3723.

Oliver, Jonathan, Chun Cheng, and Yanggui Chen (2013). "TLSH a locality sensitive hash". In: pp. 7–13.

Oliver, Jonathan, Scott Forman, and Chun Cheng (2014). "Using randomization to attack similarity digests". In: *International Conference on Applications and Techniques in Information Security*. Springer, pp. 199–210.

Otsuki, Yuto et al. (2018). "Building stack traces from memory dump of Windows x64". In: *Digital Investigation* 24, S101–S110.

Pagani, Fabio and Davide Balzarotti (2019). "Back to the whiteboard: a principled approach for the assessment and design of memory forensic techniques". In: *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 1751–1768.

Pagani, Fabio, Matteo Dell'Amico, and Davide Balzarotti (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.

Pagani, Fabio, Oleksii Fedorov, and Davide Balzarotti (2019). "Introducing the temporal dimension to memory forensics". In: *ACM Transactions on Privacy and Security (TOPS)* 22.2, pp. 1–21.

Palutke, Ralph and Felix Freiling (2018). "Styx: Countering robust memory acquisition". In: *Digital Investigation* 24, S18–S28.

Palutke, Ralph et al. (2020). "Hiding Process Memory via Anti-Forensic Techniques". In: *Digital Investigation*.

Petroni Jr, Nick L et al. (2006). "FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory". In: *Digital Investigation* 3.4, pp. 197–210.

Potchik, Brian (2017). *Architecture Agnostic Function Detection in Binaries*. URL: https://binary.ninja/2017/11/06/architecture-agnostic-function-detection-in-binaries.html.

Prakash, Aravind et al. (2014). "On the Trustworthiness of Memory Analysis—An Empirical Study from the Perspective of Binary Execution". In: *IEEE Transactions on Dependable and Secure Computing* 12.5, pp. 557–570.

Pridgen, Adam, Simson Garfinkel, and Dan S Wallach (2017). "Picking up the trash: Exploiting generational GC for memory analysis". In: *Digital Investigation* 20, S20–S28.

Rabaiotti, Joseph R and Christopher James Hargreaves (2010). "Using a software exploit to image RAM on an embedded system". In: *Digital Investigation* 6.3-4, pp. 95–103.

Reina, Alessandro et al. (2012). "When hardware meets software: A bulletproof solution to forensic memory acquisition". In: *Proceedings of the 28th annual computer security applications conference*, pp. 79–88.

Ren, Liwei (Mar. 2015). *A Theoretic Framework for Evaluating Similarity Digesting Tools*. visited on 2019-04-10. URL: https://www.dfrws.org/file/127/download?token=5YOUdHpY.

Richard III, Golden G and Andrew Case (2014). "In lieu of swap: Analyzing compressed RAM in Mac OS X and Linux". In: *Digital Investigation* 11, S3–S12.

Rodríguez, Ricardo J, Miguel Martín-Pérez, and Iñaki Abadía (2018). "A tool to compute approximation matching between windows processes". In: *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*. IEEE, pp. 1–6.

Roussev, Vassil (2009). "Building a better similarity trap with statistically improbable features". In: *2009 42nd Hawaii International Conference on System Sciences*. IEEE, pp. 1–10.

— (2010). "Data fingerprinting with similarity digests". In: *IFIP International Conference on Digital Forensics*. Springer, pp. 207–226.

— (2011). "An evaluation of forensic similarity hashes". In: *digital investigation* 8, S34–S41.

Roussev, Vassil, Irfan Ahmed, and Thomas Sires (2014). "Image-based kernel fingerprinting". In: *Digital Investigation* 11, S13–S21.

Roussev, Vassil, Golden G Richard III, and Lodovico Marziale (2007). "Multi-resolution similarity hashing". In: *digital investigation* 4, pp. 105–113.

Roussev, Vassil et al. (2006). "md5bloom: Forensic filesystem hashing revisited". In: *digital investigation* 3, pp. 82–90.

Ruff, Nicolas (2008). "Windows memory forensics". In: *Journal in Computer Virology* 4.2, pp. 83–100.

Rutkowska, Joanna (2007). "Beyond the CPU: Defeating hardware based RAM acquisition". In: *Proceedings of BlackHat DC* 2007.

Saltaformaggio, Brendan et al. (2015a). "GUITAR: Piecing together android app GUIs from memory images". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 120–132.

— (2015b). "Vcr: App-agnostic recovery of photographic evidence from android device memory images". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 146–157.

Saltaformaggio, Brendan et al. (2016). "Screen after previous screens: Spatial-temporal recreation of android app displays from memory images". In: *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pp. 1137–1151.

Saur, Karla and Julian B Grizzard (2010). "Locating × 86 paging structures in memory images". In: *digital investigation* 7.1-2, pp. 28–37.

Schatz, Bradley (2007). “BodySnatcher: Towards reliable volatile memory acquisition by software”. In: *digital investigation* 4, pp. 126–134.

Schneider, Janine, Julian Wolf, and Felix Freiling (2020). “Tampering with Digital Evidence is Hard: The Case of Main Memory Images”. In: *Forensic Science International: Digital Investigation* 32, p. 300924.

Schuster, Andreas (2006a). “Pool allocations as an information source in Windows memory forensics”. In: *IT-Incident Management & IT-Forensics-IMF 2006*.

— (2006b). “Searching for processes and threads in Microsoft Windows memory dumps”. In: *digital investigation* 3, pp. 10–16.

— (2008). “The impact of Microsoft Windows pool allocation strategies on memory forensics”. In: *digital investigation* 5, S58–S64.

Seitzer, Maximilian, Michael Gruhn, and Tilo Müller (2015). “A bytecode interpreter for secure program execution in untrusted main memory”. In: *European Symposium on Research in Computer Security*. Springer, pp. 376–395.

Shah, Abhishek (2010). *Approximate disassembly using dynamic programming. Master's report, Department of Computer Science, San Jose State University*.

Shin, Eui Chul Richard, Dawn Song, and Reza Moazzezi (2015). “Recognizing Functions in Binaries with Neural Networks.” In: *USENIX Security Symposium*, pp. 611–626.

Simmons, Patrick (2011). “Security through amnesia: a software-based solution to the cold boot attack on disk encryption”. In: *Proceedings of the 27th Annual Computer Security Applications Conference*, pp. 73–82.

Simon, Matthew and Jill Slay (2009). “Enhancement of forensic computing investigations through memory forensic techniques”. In: *2009 International Conference on Availability, Reliability and Security*. IEEE, pp. 995–1000.

— (2010). “Recovery of skype application activity data from physical memory”. In: *2010 International Conference on Availability, Reliability and Security*. IEEE, pp. 283–288.

Solomon, Jason et al. (2007). “User data persistence in physical memory”. In: *Digital Investigation* 4.2, pp. 68–72.

Stadlinger, Johannes, Andreas Dewald, and Frank Block (2018). “Linux Memory Forensics: Expanding Rekall for Userland Investigation”. In: *2018 11th International Conference on IT Security Incident Management & IT Forensics (IMF)*. IEEE, pp. 27–46.

Stevens, Richard M and Eoghan Casey (2010). “Extracting Windows command line details from physical memory”. In: *digital investigation* 7, S57–S63.

Stüttgen, Johannes and Michael Cohen (2013). “Anti-forensic resilient memory acquisition”. In: *Digital investigation* 10, S105–S115.

— (2014). “Robust Linux memory acquisition with minimal target impact”. In: *Digital Investigation* 11, S112–S119.

Stüttgen, Johannes, Stefan Vömel, and Michael Denzel (2015). “Acquisition and analysis of compromised firmware using memory forensics”. In: *Digital Investigation* 12, S50–S60.

Suiche, Matthieu (2008). “Windows hibernation file for fun 'n'profit”. In: *Black hat* 2008.

Suma, GS, S Dija, and KL Thomas (2014). “A novel methodology for windows 7× 64 memory forensics”. In: *2014 IEEE International Conference on Computational Intelligence and Computing Research*. IEEE, pp. 1–6.

Sun, He et al. (2014). “Trustdump: Reliable memory acquisition on smartphones”. In: *European Symposium on Research in Computer Security*. Springer, pp. 202–218.

Sutherland, Iain et al. (2008). "Acquiring volatile operating system data tools and techniques". In: *ACM SIGOPS Operating Systems Review* 42.3, pp. 65–73.

Sylve, Joe et al. (2012). "Acquisition and analysis of volatile memory from android devices". In: *Digital Investigation* 8.3-4, pp. 175–184.

Sylve, Joe T, Vico Marziale, and Golden G Richard III (2016). "Pool tag quick scanning for windows memory analysis". In: *Digital Investigation* 16, S25–S32.

— (2017). "Modern windows hibernation file analysis". In: *Digital Investigation* 20, pp. 16–22.

Taubmann, Benjamin et al. (2015). "A lightweight framework for cold boot based forensics on mobile devices". In: *2015 10th International Conference on Availability, Reliability and Security*. IEEE, pp. 120–128.

Thing, Vrizlynn LL and Zheng-Leong Chua (2013). "Smartphone Volatile Memory Acquisition for Security Analysis and Forensics Investigation". In: *IFIP International Information Security Conference*. Springer, pp. 217–230.

Thing, Vrizlynn LL, Kian-Yong Ng, and Ee-Chien Chang (2010). "Live memory forensics of mobile phones". In: *digital investigation* 7, S74–S82.

Thomas, Tyler et al. (2020). "Memory FORESHADOW: Memory FOREnSics of HArDware cryptOcurrency Wallets–A Tool and Visualizaton Framework". In: *Digital Investigation*.

Tridgell, Andrew (2002). *Spamsum readme*. URL: https://www.samba.org/ftp/unpacked/junkcode/spamsum/README.

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

Uroz, Daniel and Ricardo J Rodríguez (2020). "On Challenges in Verifying Trusted Executable Files in Memory Forensics". In: *Forensic Science International: Digital Investigation* 32, p. 300917.

Van Baar, RB, Wouter Alink, and AR Van Ballegooij (2008). "Forensic memory analysis: Files mapped in memory". In: *digital investigation* 5, S52–S57.

Vidas, Timothy (2007). "The acquisition and analysis of random access memory". In: *Journal of Digital Forensic Practice* 1.4, pp. 315–323.

— (2010). "Volatile memory acquisition via warm boot memory survivability". In: *2010 43rd Hawaii International Conference on System Sciences*. IEEE, pp. 1–6.

Vömel, Stefan and Felix C Freiling (2011). "A survey of main memory acquisition and analysis techniques for the windows operating system". In: *Digital Investigation* 8.1, pp. 3–22.

— (2012). "Correctness, atomicity, and integrity: defining criteria for forensically-sound memory acquisition". In: *Digital Investigation* 9.2, pp. 125–137.

Vömel, Stefan and Hermann Lenz (2013). "Visualizing indicators of Rootkit infections in memory forensics". In: *2013 Seventh International Conference on IT Security Incident Management and IT Forensics*. IEEE, pp. 122–139.

Vömel, Stefan and Johannes Stüttgen (2013). "An evaluation platform for forensic memory acquisition software". In: *Digital Investigation* 10, S30–S40.

Wächter, Philipp and Michael Gruhn (2015). "Practicability study of android volatile memory forensic research". In: *2015 IEEE international workshop on information forensics and security (WIFS)*. IEEE, pp. 1–6.

Walters, A, Blake Matheny, and Doug White (2008). "Using hashing to improve volatile memory forensic analysis". In: *American Acadaemy of forensic sciences annual meeting*.

Walters, Aaron and Nick L Petroni (2007). "Volatools: Integrating volatile memory into the digital investigation process". In: *Black Hat DC* 2007, pp. 1–18.

Walters, Aron (2006). "FATKit: Detecting malicious library injection and upping the "anti"". In: *July* 4, p. 4.

Wartell, Richard et al. (2011). "Differentiating code from data in x86 binaries". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, pp. 522–536.

White, Andrew, Bradley Schatz, and Ernest Foo (2012). "Surveying the user space through user allocations". In: *Digital Investigation* 9, S3–S12.

— (2013). "Integrity verification of user space code". In: *Digital Investigation* 10, S59–S68.

Witherden, Freddie (2010). "Memory forensics over the ieee 1394 interface". In: 3, p. 2017. URL: https://freddie.witherden.org/pages/ieee-1394-forensics.pdf.

Yang, Seung Jei et al. (2017). "Live acquisition of main memory data from Android smartphones and smartwatches". In: *Digital Investigation* 23, pp. 50–62.

Yitbarek, Salessawi Ferede et al. (2017). "Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors". In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, pp. 313–324.

Young, Joel et al. (2012). "Distinct sector hashes for target file detection". In: *Computer* 45.12, pp. 28–35.

Zhang, Ruichao, Lianhai Wang, and Shuhui Zhang (2009). "Windows memory analysis based on kpcr". In: *2009 Fifth international conference on information assurance and security*. Vol. 2. IEEE, pp. 677–680.

Zhang, Shuhui et al. (2010). "Exploratory study on memory analysis of Windows 7 operating system". In: *2010 3rd International conference on advanced computer theory and engineering (ICACTE)*. Vol. 6. IEEE, pp. V6–373.

Zhao, Qian and Tianjie Cao (2009). "Collecting Sensitive Information from Windows Physical Memory." In: *JCP* 4.1, pp. 3–10.

Zheng, Jiamin et al. (2017). "An anti-forensics method against memory acquiring for Android devices". In: *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*. Vol. 1. IEEE, pp. 214–218.

Zwanger, Viviane, Elmar Gerhards-Padilla, and Michael Meier (2014). "Codescanner: Detecting (Hidden) x86/x64 code in arbitrary files". In: *Malicious and Unwanted Software: The Americas (MALWARE), 2014 9th International Conference on*. IEEE, pp. 118–127.