A Study of Malware Detection Techniques for HPC Application Recognition

Bachelor Thesis

University of Basel
Faculty of Science
Department of Mathematics and Computer Science
High Performance Computing Group

Advisor and Examiner: Dr. Prof. Florina M. Ciorba
Supervisor: Thomas Jakobsche

Author: Edi Zeqiri
Email: edi.zeqiri@stud.unibas.ch

February 2, 2024

Abstract

This thesis investigates how well fuzzy hashing methods work in High Performance Computing (HPC) systems to improve malware detection. In light of growing concerns about the security of these critical infrastructures, traditional Cyber Security measures are insufficient, requiring the development of novel defenses against highly skilled attackers. By integrating fuzzy hashing techniques like TLSH, ssdeep ,"Strings" and Machoke, we propose a novel application tailored for the unique challenges posed by HPC systems. We illustrate the flexibility and potential gains these approaches provide for detecting HPC binaries as malware through a series of empirical assessments, including the creation of a proof-of-concept framework. Our results demonstrate significant progress in static binary detection, adding to the field of Cyber Security by offering a more sophisticated view of malware classification in High Performance Computing environments. With TLSH, we can classify more than 80% of HPC binaries with high confidence and reach F1 scores as high as 0.95. This work closes a significant research gap and establishes the foundation for further studies into the application and optimization of fuzzy hashing to improve the security of HPC systems.

Contents

Acknowledgments

I would like to extend my heartfelt thanks to Prof. Dr. Florina M. Ciorba for her guidance during my Bachelor's thesis with the HPC group. Her insightful feedback and valuable advice were crucial in improving my scientific mindset and the correctness of this thesis. I also appreciate Thomas Jakobsche's exceptional mentorship. His consistent support and enriching interactions significantly contributed to my growth. A special acknowledgment to all members of the HPC group for making my Bachelor's thesis journey interesting. I learned a lot from all the HPC members and the group meetings overall. Immense gratitude to my family and friends for their unwavering support and love.

Chapter 1

Introduction

In a time where High-Performance Computing (HPC) systems play an important role in advancing scientific and industrial research, ensuring their security is of utmost importance. This thesis delves at the field of malware detection specifically within the context of recognizing HPC applications. The increasing intricacy and size of HPC settings provide distinctive difficulties and weaknesses, making them attractive targets for unscrupulous individuals. Conventional Cyber Security solutions often prove inadequate in dealing with the complex threats presented in these settings.

This work aims to overcome this gap by analyzing current malware detection approaches and their application to HPC systems. The primary objective is to assess the adaptability and improvement potential of these strategies in identifying High-Performance Computing (HPC) binaries or applications as if they were malware. This thesis examines several detection techniques, but lays the primary focus on the application of hash analysis as a tool for static detection of malware.

Although there are binary/malware detection frameworks out there, they are mostly oriented toward dynamic analysis as opposed to static analysis. Examining a program's activity while it's operating is known as dynamic analysis, and it may be more successful in discovering certain kinds of malware. However, depending just on dynamic analysis might be unfeasible for HPC systems, where real-time speed and efficiency are critical. These frameworks can fail to detect possible risks in HPC binaries quickly and accurately enough to not affect system performance.

Currently, there is a huge lack in research explicitly addressing malware detection in High-Performance Computing (HPC) systems. This thesis tackles this gap by presenting a new approach: the use of fuzzy hashing algorithms for HPC binaries and application detection. Fuzzy hashing, largely recognized in digital forensics for finding similar although not identical data, presents a potential way for identifying versions of HPC binaries. This use of fuzzy hashing in the context of HPC is innovative, establishing this thesis as a pioneering effort in examining its efficacy. By adapting fuzzy hashing to the particular issues offered by HPC systems, this study intends to advance the subject of Cyber Security in a path that still needs to be explored.

1.1 Motivation

Administrators of High-Performance Computing systems have a significant interest in knowing the activities occurring on these platforms. There is a need for clear insight into the operations, processes, and applications running on HPC systems. This understanding is vital for maximizing performance and ensuring that the systems are utilized for their intended scientific or computational goals. Monitoring who is utilizing HPC resources and for what purposes is another essential component. In contexts where HPC systems are shared resources, such as in research universities, it's necessary to maintain track of utilization trends. This assists in properly controlling the systems, ensuring that they are utilized responsibly and for the intended research or computing activities. Protection against malware and hostile actors is a major concern for HPC systems. Users and administrators are especially mindful of risks like hashcat, used for password cracking, or cryptocurrency miners that waste computational resources [10]. Ensuring these systems are protected against such exploitative malware is essential for sustaining the integrity and security of HPC environments.

The fundamental research question this thesis will address is: Can fuzzy hashing be successfully utilized to identify or classify supplied binaries in High-Performance Computing systems and obtain trustworthy intelligence? This investigation looks into the possibility of fuzzy hashing as a tool for Cyber Security in HPC systems, specifically focusing on its capacity to distinguish and classify different sorts of binaries, which might include both legitimate programs and possible malware.

1.2 Challenges

The challenge in utilizing fuzzy hashing to identify or classify binaries in HPC systems lies in their inherent complexity. Binaries in these systems may be compared to complicated drawings comprising nested sub-pictures in a recursive manner [14]. This makes effectively recognizing and classifying them via fuzzy hashing a non-trivial effort. Effectively interpreting these hierarchical structures is essential to successful detection and classification.

A key challenge in this study is the limitation imposed by privacy considerations. Often, there is an impossibility to directly access the binaries for analysis, resulting to a dependence on static analysis. This constraint implies that the study must traverse the problem of obtaining relevant and correct information from binary without the benefit of dynamic investigation, making the process more complicated and subtle.

Another problem is the necessity for rapid metadata collection and generating intelligent estimates based on this information. The act of acquiring information fast and correctly, and then utilizing this data to make educated predictions about the nature of the binary, is important. This speed and efficiency are critical in HPC systems, where delay or inaccuracy may lead to serious security risks or performance hindrances.

While conventional solutions for this particular topic have not been developed, there are current solutions that handle comparable challenges, but with different purposes. These technologies, mostly designed for different settings such as digital forensics or data analysis, involve approaches that might be equivalent to fuzzy hashing in recognizing and classifying data. However, their aims and application fields vary greatly from the emphasis of this study. This thesis studies the possibilities of adapting these current approaches to the particular environment and needs of HPC systems for detection and classification.

1.3 Goals

The main goal of this project is to develop a proof of concept framework particularly tailored to handle the issues of employing fuzzy hashing for recognizing and classifying binaries in HPC systems. The objective is to design a system that is flexible and successful in multiple HPC situations, giving a standardized way to dealing with the difficulties and privacy limitations involved with binary analysis in these systems. By doing this, the study will not only address the present problems but also give a useful resource to the larger HPC community, boosting Cyber Security measures across diverse HPC platforms.

The key to achieving our goals is carrying out fuzzy hashing correctly. This approach, notable for its potential to discover and classify comparable but not identical data sets, is seen as the basic answer to the issues highlighted. An important part of this approach is to guarantee that it is both quick and generic. The framework designed must be capable of swiftly processing and interpreting data, a need given the high-performance nature of HPC computers. Additionally, the solution must be general enough to be useful across a broad variety of HPC settings, independent of their individual setups or the sorts of applications they execute.

To further enhance and verify the solution, the study will include established guidelines and discoveries from the area of malware detection. This involves assessing current approaches, evaluating their usefulness, and changing them as appropriate for the context of HPC systems. The objective is to expand upon the present body of knowledge, customizing it to provide a more robust, efficient, and successful solution to malware detection in HPC systems utilizing fuzzy hashing.

1.4 Our Contribution

The objective is to provide techniques that offer a more thorough, nuanced comprehension of the data in addition to recognizing patterns and variances in binary. This might include merging several fuzzy hashing methods or integrating them with other analytical tools in order to increase the precision and effectiveness of malware classification and detection.

Research and advancement of these improved fuzzy hashing methods directly contributes to strengthening HPC system security procedures. This discovery may greatly aid in protecting HPC settings from complex malware attacks by offering a more efficient tool for binary identification and classification. The knowledge obtained from this research may lead to the development of new standards and best practices for HPC system security.

Besides security, HPC administrators can gain an introspective view into theirs systems and know which user is submitting which job for which purpose. This metadata gives insight into batch jobs and can categorize them into self created categories, like weather or physics simulation. This can prevent misuse of resources and

Entities in both academic and industrial sectors that rely on HPC systems for complex computations and data processing will have a vested interest in the success of this research. Enhanced security protocols may augment the dependability and authenticity of their output, guaranteeing that these formidable computational instruments continue to be secure and operational for important investigations and advancements.

Chapter 2

Background

2.1 Fuzzy Hashers

Fuzzy hashing is a technique used in Computer Science, particularly in Cyber Security, to identify files or data that are similar to each other. Fuzzy hashing accepts little changes in the data, allowing for the discovery of similarities, in contrast to classical hashing, which seeks to produce a unique identifier or hash for each separate piece of data. Conventional hashing is good at verifying exact matches but not very effective at identifying patterns or similarities because it generates whole new hashes for even the smallest changes to the material.
By generating comparable hashes for files that are similar but not identical, fuzzy hashing adds flexibility. This function is especially helpful for identifying malware variants in the field of Cyber Security. Malware frequently modifies itself slightly to avoid detection; yet, fuzzy hashing allows these small changes to be connected to known malware, improving detection.
Comparing two text passages can be used to illustrate the fuzzy hashing idea. Fuzzy hashing measures the degree of similarity between two texts, regardless of differences, whereas standard hashing determines whether two texts are exactly the same, producing a binary answer. This method works well for handling data that comes from the same source but has undergone minor changes or corruptions, allowing for a more sophisticated identification of the connections across datasets.

Note that in this thesis the words sample and binary are used interchangeably.

ssdeep - Context triggered piece wise hashing

Ssdeep is a fuzzy hasher which uses a method called context-triggered piecewise hashing [12] . With this method, a file is divided into many segments, each of which is bounded by the context of the data. This means that the hash is generated using the file's structure rather than its whole content. With this technique, ssdeep can produce a hash that accurately captures the general organization and content pattern of the file.

Ssdeep compares two files and produces a similarity score that indicates how similar the files are to one another. For instance, ssdeep can determine that two files are essentially identical even if they have variations, such as a few altered lines in a document or changes in a malware variants.

This score is especially helpful for detecting changed or updated malware, which are variants of the same file that have been modified.

The segments of ssdeep vary in length, unlike the fixed-size blocks used in standard hashing. Its capacity to identify similarities in files that have seen little modifications is largely dependent on this variability. After file segmentation, ssdeep creates a hash for each part. These hashes are shorter than standard cryptographic hashes and intended to capture the substance of each section. For this, ssdeep uses a rolling hash technique [12]. Because it can be calculated fast across a window of data that goes through the file, a rolling hash is efficient. The hash is updated according to incoming and outgoing data as the window progress.
To compare ssdeep hashes, the library uses its own compare function. This function measures the edit distance between two strings, s_1 and s_2, defined as the minimum number of operations required to transform s_1 into s_2. These operations can include changing, inserting, or deleting a single character. When comparing two ssdeep fuzzy hashes, the resulting score ranges from 0 to 100, with 0 indicating no match and 100 signifying an identical match. Typically, a score of 80 or above is considered to indicate a confident match.[13]

TLSH - Locality Sensitive Hashing

Trend Micro Locality Sensitive Hash (TLSH) is a fuzzy hasher developed by Trend Micro. TLSH is currently the defacto standard for malware detection. Since its release, the fuzzy hasher has been introduced into Virustotal and Malwarebazaar, two industry standard platforms for collecting file samples and classifying malware in the Cyber Security field.

The TLSH algorithm starts by processing the byte string through a sliding window of size 5 bytes. This window goes through the byte string and obtains triplets of bytes. The Pearson hash function [16] is used for analyzing each triplet, to reduce collision and improve the uniform distribution. As a result, the captures are the frequency of different triplet hash values by the count of increments of matching bucket counts. After the sliding window analysis, each triplet's frequency is represented as an array of bucket counts. TLSH generates three quartile points from this array. A statistical metric known as a quantile splits a collection of data into four equal pieces. In this case, the algorithm finds numbers such that 25% of the bucket counts are below or equal to the first quartile ($q1$), 50% are below or equal to the second quartile ($q2$), and 75% are below or equal to the third quartile ($q3$). The bucket counts' total distribution may be summed up using these quartiles.

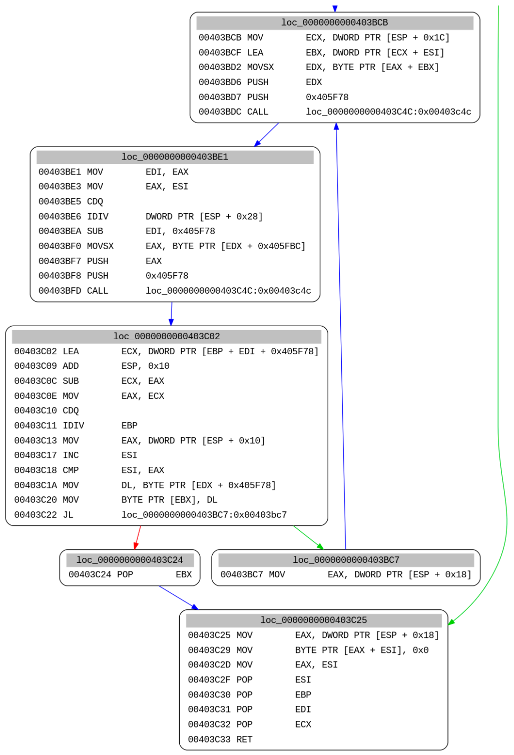
The body of the TLSH hash is created by examining each bucket's count relative to the quartile points. For each bucket, a 2-bit encoding gets created, based on which quartile range the count falls into. If the count is less than or equal to $q1$, then 00 is returned; if it's between $q1$ and $q2$, 01 is returned; if it's between $q2$ and $q3$, 10 is emitted; and if it's above $q3$, 11 is emitted. This process converts the bucket array into a binary string that reflects the statistical profile of the data.

Combining the binary string of the digest body and the hex format of the digest header, which includes quartile and checksum details, creates the final TLSH hash. This final hash accurately represents the statistics and content of the data.
The scoring of TLSH is non-conventional, compared to the other fuzzy hashers. A score of 0 indicates a perfect match, while higher scores indicate the dissimilarity of two TLSH hashes. This

feature gives the user the possibility to choose the false positive tolerance (risk tolerance) and self determine the scope of TLSH. [15]

Machoke - Call Flow Graph Hashing

Machoke, which shares its name with a Pokémon character, is a tool specifically created for identifying malware in binary files. The foundation of it is an idea known as a Call Flow Graph (CFG). A CFG is simply a map that illustrates the Assembly language level interactions and calls between various parts of a computer program, namely functions. Compared to higher-level programming languages, this level is more comprehensive and closer to machine code. The CFG is utilized by Machoke to log the progression of these function invocations. Machoke divides the code into sections known as fundamental blocks whenever a function calls another function inside the code. Then, in order to keep track of these call points, it gives each one a distinct number. Machoke uses the reverse engineering program Radare2 to analyze the binary file and comprehend its structure. The positions and specifics of these function calls inside each block are determined by Radare2 once it decodes the binary file. Originally, Machoke first creates a unique hash value for each code block using MD5 hashing in its first version, but in this thesis an alternative hashing technique called MurmurHash3 generates a 32-bit hash for every block instead. Then, a single lengthy string is created by combining these distinct hashes from every block. The final string is then what is called a Machoke hash. [8]



```
loc_0000000000403BCB
00403BCB MOV        ECX, DWORD PTR [ESP + 0x1C]
00403BCF LEA        EBX, DWORD PTR [ECX + ESI]
00403BD2 MOVSX      EDX, BYTE PTR [EAX + EBX]
00403BD6 PUSH       EDX
00403BD7 PUSH       0x405F78
00403BDC CALL       loc_0000000000403C4C:0x00403c4c
```

```
loc_0000000000403BE1
00403BE1 MOV        EDI, EAX
00403BE3 MOV        EAX, ESI
00403BE5 CDQ
00403BE6 IDIV       DWORD PTR [ESP + 0x28]
00403BEA SUB        EDI, 0x405F78
00403BF0 MOVSX      EAX, BYTE PTR [EDX + 0x405FBC]
00403BF7 PUSH       EAX
00403BF8 PUSH       0x405F78
00403BFD CALL       loc_0000000000403C4C:0x00403c4c
```

```
loc_0000000000403C02
00403C02 LEA        ECX, DWORD PTR [EBP + EDI + 0x405F78]
00403C09 ADD        ESP, 0x10
00403C0C SUB        ECX, EAX
00403C0E MOV        EAX, ECX
00403C10 CDQ
00403C11 IDIV       EBP
00403C13 MOV        EAX, DWORD PTR [ESP + 0x10]
00403C17 INC        ESI
00403C18 CMP        ESI, EAX
00403C1A MOV        DL, BYTE PTR [EDX + 0x405F78]
00403C20 MOV        BYTE PTR [EBX], DL
00403C22 JL         loc_0000000000403BC7:0x00403bc7
```

```
loc_0000000000403C24
00403C24 POP        EBX
```

```
loc_0000000000403BC7
00403BC7 MOV        EAX, DWORD PTR [ESP + 0x18]
```

```
loc_0000000000403C25
00403C25 MOV        EAX, DWORD PTR [ESP + 0x18]
00403C29 MOV        BYTE PTR [EAX + ESI], 0x0
00403C2D MOV        EAX, ESI
00403C2F POP        ESI
00403C30 POP        EBP
00403C31 POP        EDI
00403C32 POP        ECX
00403C33 RET
```

Figure 2.1: Example Call Flow Graph of a function and its call blocks [2]

Two obvious disadvantages can be spotted here. First, since every functions hashed flow get concatenated, the final output size is dependent on the number of functions and can be roughly estimated by the size of the binary. Because of this, comparing Machoke hashes scales with the size of the sample. The second disadvantage is the disassembling of the sample itself. Generating the hash has the same performance problem like comparing them. With bigger samples (assuming bigger means more functions), this fuzzy hasher scales very poorly.

With Jaccard distance calculation, a machoke hash gets treated as a string and the comparison returns a score from 0 to 1, where 0 is no match and 1 is a perfect match. Jaccard distance is defined as follows:

$$J(A,B) = \frac{\mid A \cap B \mid}{\mid A \cup B \mid}$$

Where A and B are sets of the tokenized Machoke hash. The token size was set to 4 characters per token.

Strings with TLSH

This fuzzy hasher is a hybrid construction between the UNIX strings command, which returns a list of strings contained in a binary and TLSH. However, using only the UNIX strings command still has the same problematic in performance like Machoke. To enhance this approach, the use of TLSH for hashing the output of the "strings" command gets incorporated. This change separates the performance of the comparison from the variable output size of the "strings" command by significantly reducing the output to the fixed size of a TLSH hash.

This fuzzy hasher differs from plain TLSH by only hashing the strings and not the whole binary.

Chapter 3

Related Work

A major gap in the field of High-Performance Computing (HPC) is the lack of established practices for recognizing and categorizing HPC binaries. This lack of focused strategies is an indication of a serious flaw in the way that existing cybersecurity techniques apply to HPC systems. Because HPC systems have different operating needs and features from traditional computing systems, they require specific methodologies for binary analysis and malware detection.

According to Tsujita et al. [19] HPC jobs can be classified by metadata analysis. They have shown that high CPU and memory utilization correlate with high I/O. Information about the electric power consumption can divide jobs in high I/O jobs or high computation jobs.

In 2018, Ates et al. [7] introduced "Taxonomist," a method for identifying applications on supercomputers by analyzing patterns in resource usage. By analyzing their activity, this machine learning-based method enables the categorization of programs and the identification of illicit behaviors, including malware. This study is relevant to the field of malware detection in high-performance computing (HPC) settings because it offers a framework for spotting unusual application behavior, which is an important factor in detecting possible malware in HPC binaries.

Budiardja et al. (2016) [9] explore XALT, a tool for monitoring library function utilization in high-performance computing (HPC) systems. It describes the transparent way in which XALT gathers job-level and link-time data, with particular attention to its function-tracking capability that allows programs to identify the precise library functions they use. This makes it possible to analyze program usage more precisely, which helps with security and optimization initiatives.

Jakobsche et al. (2021) [11] presents an Execution Fingerprint Dictionary (EFD) for recognizing HPC applications by capturing system metric patterns during execution. Inspired by Shazam, this approach achieves over 95% F-score in application identification using just one system statistic within the first two minutes of operation, greatly reducing the amount of data required. This method, which uses little data to achieve high application identification accuracy, provides a low-weight and effective mechanism for tracking and maybe identifying abnormalities or malware in HPC systems.

Peisert (2010) [17] investigates the idea of ”fingerprinting” HPC systems in order to recognize communication and computational characteristics, with a particular emphasis on dynamic analysis using MPI data monitoring. The goal of this technique is to distinguish between typical program use and any abuses like resource exploitation or illegal access. Peisert's research highlights the possibility of using communication patterns as markers of unusual activity, which is pertinent to my study. It implies that comparable tactics might improve malware detection in HPC binaries by spotting departures from normal application behaviors.

Chapter 4

Methodology

The execution relies on three consecutive phases. The first phase emphasizes the gathering of the data. Since this thesis tries to solve a current practical problem, it needs data from current actively used binaries. After the relevant labeled binaries have been retrieved, the fuzzy hashing has to start, where samples need to be hashed and labeled correctly to fit the database schema and be saved concurrently. The last phase relies on analysing the samples and return a prediction for each sample based on the application of the fuzzy hashers.
Thresholds are defined as limits set to each fuzzy hashers returned score. It is optimal to only accept results on which the fuzzy hasher is certain to have the false positive rate as low as positive. The goal of this methodology is, to empirically test which thresholds are acceptable as a match and from which threshold we want to return a "No Match" and not accept the result for a prediction.

4.1 Data Gathering

4.1.1 Malware Dataset

To compare the effectiveness of the fuzzy hashers, a labeled dataset is necessary. Meaning a dataset which already has each sample mapped to a family. This allows easier testing and validation of the predictions. The dataset was sourced from vx-underground, a key repository giving a wide number of malware samples. This platform is essential for Cyber Security research and malware analysis, providing access to real-world malware instances that contribute in the development of stronger security solutions. More than 5TB of data is available on the platform. Ranging from malware creation, detection to samples mapped to their families.

The process of obtaining data required using the vx-underground[5] website, where each malware sample is available for individual download. To accommodate the enormous volume of samples, web scraping techniques were deployed. This requires web scraping to explore the site, discover download paths for each sample, and then execute a series of GET calls for quick access. This was done through python and the requests[18] library, needed for the web requests. Safety was a primary concern in handling the samples, each of which was securely zipped with the password 'infected'. Before extraction, an exclusion was configured in the antivirus Microsoft Defender to bypass any automatic blocking or removal of the samples with the samples. The unzipping procedure was automated using 7z[1] paired with a PowerShell script, providing a safe and orderly extraction of the malware samples.

The malware dataset consists of 208646 samples and 527 families in which a family represents a malware family like AgentTesla and its samples are different versions of the same malware.

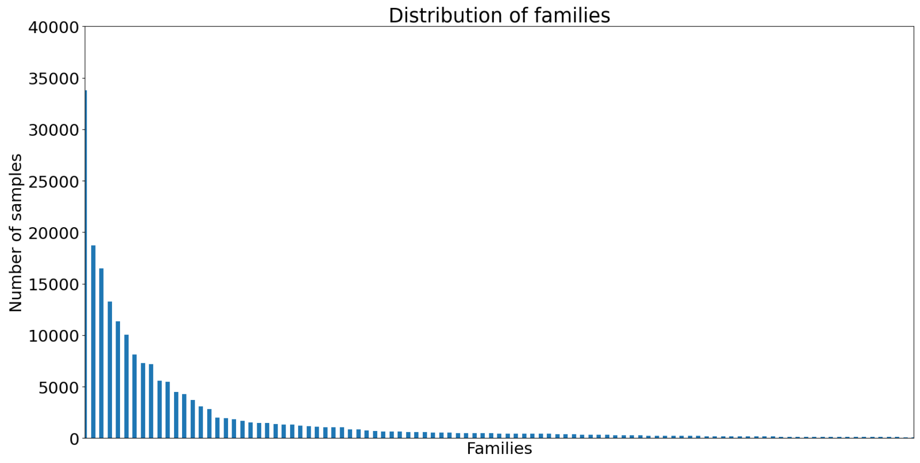


Figure 4.1: Distribution of malware families

4.1.2 Scicore

There were no pre-made datasets available for the purpose of this study. So, the research turned to using modules from the Scicore[4] High-Performance Computing system. These modules were chosen because they are used often and are marked as applications. They have executable files, which are key for this project. The main source for collecting data was the /scicore/apps/soft directory on Scicore. This directory was important because it had different applications, each in its own folder, like OpenMalaria[6]. These applications also had folders for different versions.

To collect the data, the main task was to pick out the main executable file (like 'OpenMalaria' in its application) from the 'bin' folders. It was important to only take these main files and not the other extra files that are usually found in these folders. A list was made to keep track of where these main files were located. Then, a Python script was used to gather all these files and organize them in a way that's useful for further study and analysis. The root folder is the dataset, the subfolders are the families and the files in the subfolders are samples of the respective family. This method made sure that the collected data was relevant and ready for the next steps in the research.

The Scicore dataset consists of 2066 samples and 334 families, in which families are referred to applications like OpenMalaria and samples are its subversion of previous releases or different compiler toolchains.

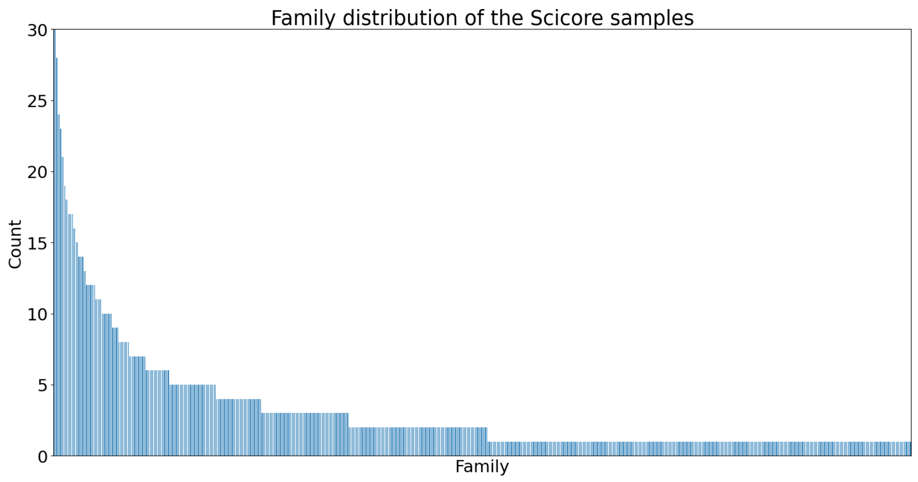


Figure 4.2: Distribution of samples per family in Scicore dataset

4.2 Fuzzy Hashing

For the experiments to work, the following framework was built as a semi automatic pipeline from the sample as a binary to the fuzzy hash into the database. The framewok is labeled in this way, because the user can provide arbitrary binaries in the folder structure pictured above and call the python file.

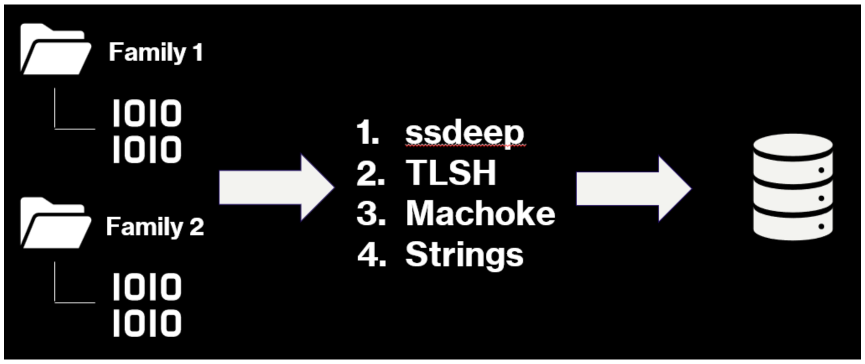


Figure 4.3: Enter Caption

The following tasks will be handled by the framework:

1. Connect to the database and create the tables.
2. Enumerate all the subfolders and treat them as a family.
3. For every sample, take all four fuzzy hashes and time their performance.

4. Combine the result including the SHA256 hash as the ID and send it to the database.

With this framework, arbitrary samples and families can be fuzzy hashed and saved into a database for further processing. In this case, Python was the language used to implement the functionalities and Jupyter notebooks for further processing. This software stack was purposefully chosen, because most of the fuzzy hashers have wrapper libraries built with python which made it the obvious choice.

The main methodology of testing is focused on the use of the ROC curve 4.3.1 and F1 score 4.3.2 as the primary comparison of the effectiveness. To reach those two parameters, which lead to the results, the data has to be prepared and the compared accordingly. Normalizing the data follows the usual process of removing "NaN"s and checking the consistency of the data. TLSH and ssdeep will always return a hash for a sample, given the size is greater than 32 bytes. TLSH will also return "TNULL" as a fuzzy hash for samples which do not fit the implementation. Important to mention is that Machoke will not work on non compiled samples, like python scripts, bash scripts or also compiled samples which do not meet the requirement for Radare2 to be reversed.

4.3 Data Analysis

To analyse the cleaned data, two important statistical tools will be used, the ROC curve and the F1 score. These tools will define how well the fuzzy hashers can predict the family of a sample, based on their metrics. This way we can statistically show the effectiveness of the fuzzy hashers on the datasets and understand if they are meaningful. Since we are only interested on the performance of the Scicore dataset, the malware dataset acts as a pseudo cross validation test set. Every metric the Scicore dataset returns we will validate on the malware dataset, since it is known [15] that TLSH and ssdeep are effective in recognizing malware on a dataset with 109 distinct binary malware files from three malware families.

4.3.1 Receiver Operating Characteristic (ROC) Curve

A Receiver Operating Characteristic (ROC) curve is a graphical representation to evaluate the performance of a classification model at various threshold settings. It plots two parameters:

1. **True Positive Rate (TPR)**: Also known as sensitivity, measures the proportion of actual positives correctly identified. It is calculated as:

$$TPR = \frac{TP}{TP + FN} \tag{4.1}$$

   where TP is the number of true positives and FN is the number of false negatives.

2. **False Positive Rate (FPR)**: Measures the proportion of actual negatives that are incorrectly identified as positives. It is calculated as:

$$FPR = \frac{FP}{FP + TN} \tag{4.2}$$

   where FP is the number of false positives and TN is the number of true negatives.

The ROC curve plots TPR against FPR at different threshold levels. A model with perfect prediction has a curve that goes straight up the y-axis and then along the x-axis. The area under the curve (AUC) quantifies the overall ability of the model to distinguish between the positive and negative classes. An AUC of 1 indicates perfect classification, while an AUC of 0.5 suggests no classification.

The variables are defined for this usecase as follows:

- **True Positive (TP)**: A sample from the dataset gets matched with its corresponding family
- **False Positive (FP)**: A sample from the dataset gets matched with a different family from the same dataset
- **True Negative (TN)**: A sample has no match if the corresponding family does not exist
- **False Negative (FN)**: A sample from the dataset has no match, but the corresponding family exists in the dataset

4.3.2 F1-Score

The F1-Score is a statistical metric used to evaluate the accuracy of a binary classification model, particularly in scenarios with imbalanced class distributions. It harmoniously combines precision and recall into a single measure, providing a balanced view of the model's performance. The F1-Score is crucial in situations where both false positives and false negatives carry significant importance. The formula for the F1-Score is:

$$F1 = 2 \times \frac{p \times r}{p + r} \tag{4.3}$$

For this specific use case, precision and recall are defined with respect to samples and families as follows:

- **Precision (p)**: The proportion of samples correctly identified as belonging to their respective families among all the samples identified as part of a certain family. It is calculated as:

  $$p = \frac{TP}{TP + FP} \tag{4.4}$$

  where TP represents the true positives (samples correctly matched with their family) and FP represents the false positives (samples incorrectly matched with a different family).

- **Recall (r)**: The proportion of samples correctly identified as belonging to their respective families among all the samples that actually belong to that family. It is calculated as:

  $$r = \frac{TP}{TP + FN} \tag{4.5}$$

  where TP represents the true positives and FN represents the false negatives (samples from a family not correctly identified).

4.3.3 Edi Curve

The last statistical tool used in this thesis is the Edi curve. This tool was created by the author of the paper from the necessity to gain an absolute view of the predictions made by the fuzzy hashers. The idea behind this is to measure, in absolutes,the precision based on thresholds while at the same time having a perspective of the total samples that are cut off.
With this addition, the curve represents a more practical perspective on the predictions, which the other two statistical tools do not. Without the knowledge of the amount of samples still being classified, a true positive rate is in this usecase irrelevant. For example, having a 99% true positive rate is great but if only 3 of 2000 are in the threshold range where this true positive rate applies than it is meaningless.

- **Total:** The amount of samples still being considered to have a match.

$$total = \frac{T}{S} \tag{4.6}$$

  Where S is the size of the dataset and T the amount of samples of the subset of S in the respective threshold.

Precision is taken from the F1 score subsection 4.3.2. Two points on the Edi curve are highlighted. First is the intersection of precision and recall from the F1 score and second the argmin of total and precision. This allows to define two thresholds in which a range can be interpreted. With this range, based on the requirements of the use case, a specific threshold can be set to allow either more samples to be predicted or have an exact prediction on a smaller dataset.

To summarize this tool, the Edi curve can be seen as the quantitative metric and the F1 score with precision and recall as the qualitative metric. With the Edi curve, absolute values can be analyzed, while the F1 score might help to interpret trends more accurately.

Chapter 5

Results

This chapter goes into the results of the methods. The plots were made with the assistance of the miniHPC [3] to run scheduled jobs. All of the statistics involving the malware dataset had to be created with the miniHPC, since $O(n^2)$ comparisons were needed (for the any to any comparison of the fuzzy hasher distance functions) with $n = 2 * 10^5$ which results in 1.99×10^{10} comparisons. To measure the performance and accuracy of the fuzzy hashers, an any to any comparison is necessary, meaning to compare every fuzzy hash, like TLSH for example, to all the others in the dataset and take the minimum difference score for TLSH and String and the maximum for ssdeep.
The scicore dataset was small enough that the computation could be split into first preparing all the combinations in a matrix, saving it therefore in RAM and only then to compute the difference of every entry in parallel. This process was faster than reading from the dataset one by one and decreased I/O.

After the preparation of the measures explained in chapter 4, the end result is a matrix with every sample from the original dataset, matched with its best predicted family and the corresponding score. A sample row looks like this:

ID	True Family	Predicted Family	TLSH 1	TLSH 2	Diff Score
8932331	Blackmoon	Blackmoon	T1004423DD114...	T1154423DD114...	1

Table 5.1: Example row of the comparison with TLSH

In the following plots, the goal is not only to show how good the fuzzy hashers can predict the family of a sample, but to also gain a view about their thresholds and which have to be chosen, for the best results. Therefore, in each incremental threshold step the dataset gets smaller, because the rows with a "Diff Score" of higher or lower than a threshold get removed temporarily. Setting a threshold is essential to reduce the false positive rate. If the fuzzy hasher always predicts a family, independent of the final score but given it always takes the minimum, for TLSH as an example, then it would also consider a match if the score was 3000 which is not acceptable, as can be seen in [15].

The source code of the results have been made available online: *https://github.com/edizeqiri/obstkorb*

5.1 Machoke

This fuzzy hasher will not be listed together with the other ones, because of its poor performance compared to the others. Less than 1% of the dataset had a match and even then the FP rate was over 60%. Thus, Machoke will be neglected from the comparison.
Since disassembling takes a significant amount of computing power, we could not fuzzy hash the complete malware dataset with Machoke. The scicore dataset could be completely hashed and a small subset of the malware dataset. The miniHPC could not be used, since we would have to load malware on the miniHPC, which belongs to the University of Basel and by their regulation is not allowed to have malware on it. Therefore, only a small subset of 300 samples could be hashed and tested. The small dataset size does not return accurate statistical results and will therefore not be shown. For the interested reader, figures for Machoke can be found in the Appendix A

5.2 ROC Curves

Figure 5.1-5.2 and Table 5.1-5.2 visualize the performance of the true positive rate against the false positive rate based on specific thresholds. For TLSH and Strings, the curve was generated with thresholds from $0 - 300$ and for ssdeep from $0 - 100$. The dotted line shows the accuracy of a random classifier.

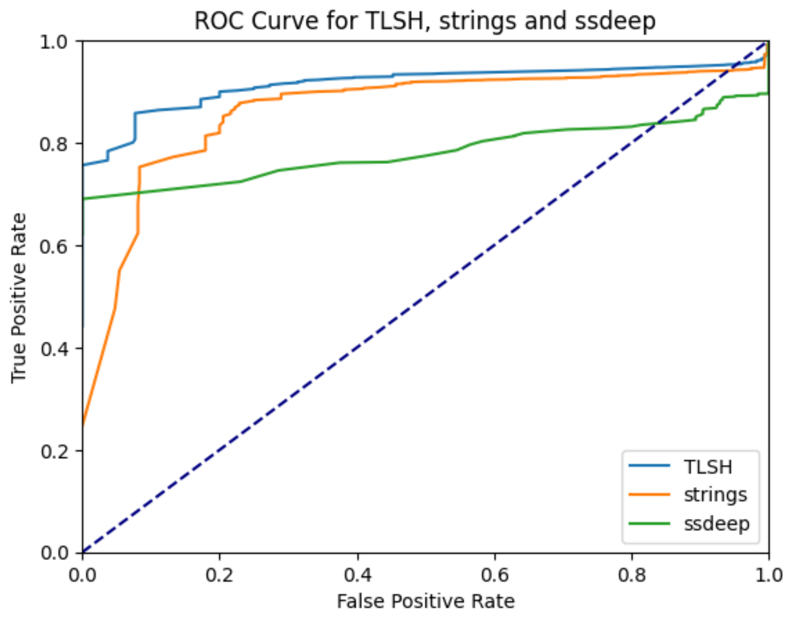


Figure 5.1: ROC Curve of the Scicore dataset

TLSH	Strings	SSDEEP
0.913	0.861	0.782

Table 5.2: Area under the curve for Figure 5.1

TLSH is leading this result, closely followed by ”Strings” and ssdeep making the 3rd place. TLSH and ”Strings” follow a similar pattern where the TP rate increases rapidly and then flatten out with a higher threshold. Ssdeep has a linear development, which shoes that increasing the threshold does not greatly benefit the TP rate.

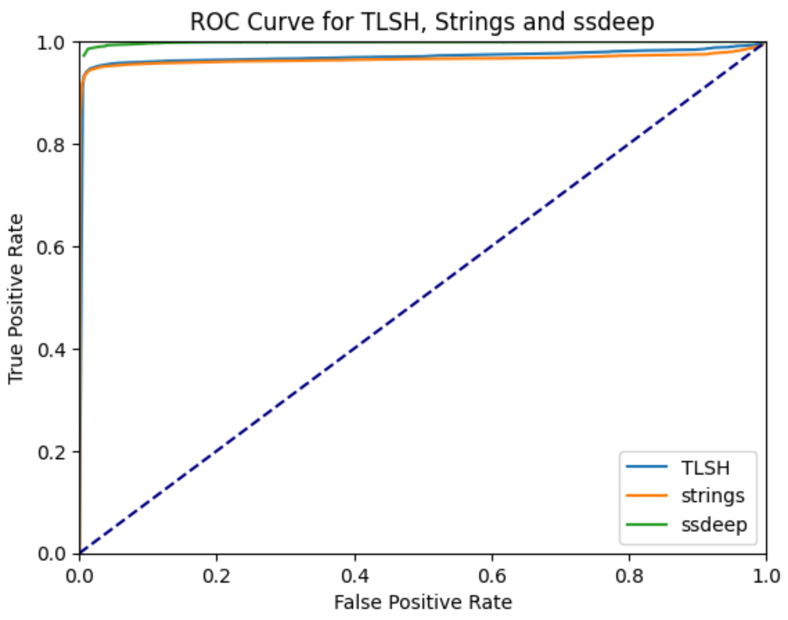


Figure 5.2: ROC Curve of the malware dataset

	TLSH	ssdeep	Strings
AUC	0.969	0.995	0.965

Table 5.3: Area under the curve for Figure 5.2

TLSH and ”Strings” are very close visually and their area under the curve could be rounded up to be the same. Ssdeep has a near perfect score visually and by the area under the curve, the malware and scicore dataset have noticeable differences. To get a better understanding of this result, other metrics have to be put into perspective. First, we will consider the file size a factor for the difference.

mean	1.71e+06
std	5.73e+06
min	2.74e+02
25%	1.08e+05
50%	3.13e+05
75%	1.09e+06
max	7.95e+07

Table 5.4: Table describing the file size for the malware dataset

mean	4.76e+06
std	2.30e+07
min	9.90e+01
25%	1.52e+04
50%	1.43e+05
75%	1.39e+06
max	3.53e+08

Table 5.5: Table describing the file size for the scicore dataset

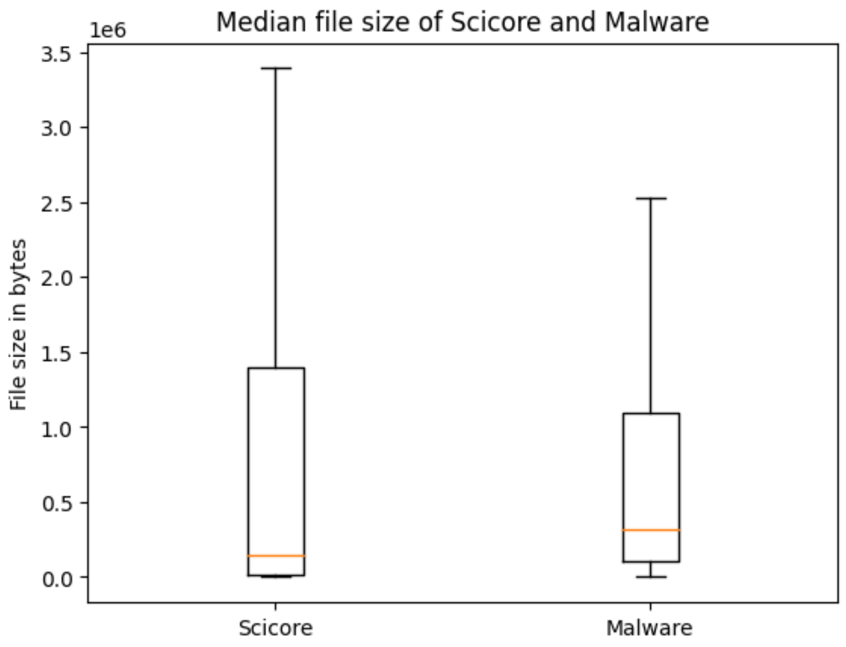


Figure 5.3: Median file size of the datasets

It is clearly visible by the tables Table 5.4,Table 5.5 and Figure 5.3 that the scicore dataset has bigger samples than the malware dataset. This can be a factor for the big difference in AOC from Figure 5.1 and Figure 5.2. Therefore, the Pearson correlation will show if the file size does correlate with the TP rate.

	Scicore	Malware
Pearson	-0.0411	-0.0264
p-value	0.458	0.592

Table 5.6: Pearson correlation of file size and TP rate for Scicore and malware dataset

Since a Pearson correlation coefficient of 0 indicates no correlation and the Pearson values are very close to 0, file size and TP rate do not correlate. The p-value is 0.46, which is quite high, reinforcing the conclusion that there is likely no significant linear correlation between the variables.

The last conclusion has to be that the datasets have different sizes. The malware dataset is bigger by a factor of 100. Reducing the dataset to the same size of the scicore dataset, while keeping the file size distribution leads to the following ROC curve.

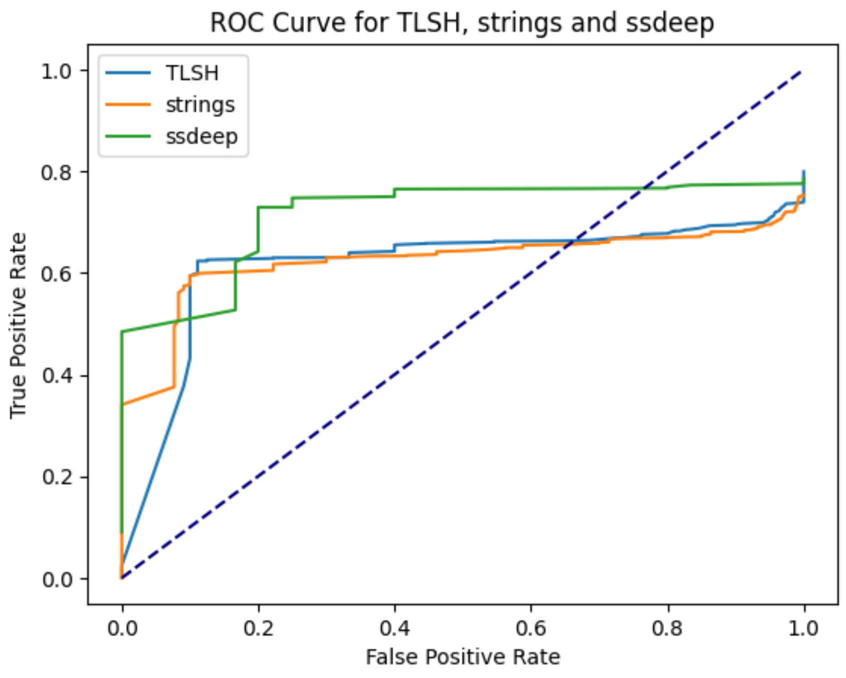


Figure 5.4: Reduced malware dataset to the size of the scicore dataset

	TLSH	ssdeep	Strings
Area under the Curve	0.616	0.715	0.623

Table 5.7: AUC for reduced malware dataset

This reduced malware dataset has now either 2 samples per family or 1% of the previous size of the family to match the size of the scicore dataset. It is clearly visible in the ROC curve itself

Figure 5.4 and in the Table 5.7, that the amount of samples drastically changes the results. TLSH and "Strings" both are below ssdeep, but all of the fuzzy hashers do not have a higher true positive rate of 80%. This trend is similar in all ROC curves in this thesis that, with increased dataset size, the true positive rate is increases.
This explains the tremendous difference in the ROC curves for the malware and Scicore datasets. Not only can we now explain the differences but also gained new insight into the behaviour of how fuzzy hashers behave based on the amount of data that is provided. In the following results, the reduced malware dataset will be added as a third dataset for comparison.

5.3 F1 Score

5.3.1 Malware

The following four graphs are the F1 scores with the corresponding precision and recall of the fuzzy hashers mapped on the thresholds. The Y-axis is the score and the X-axis is the threshold.

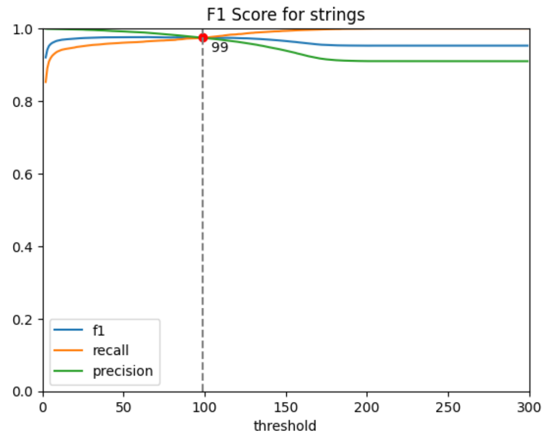


Figure 5.5: F1 score of Strings

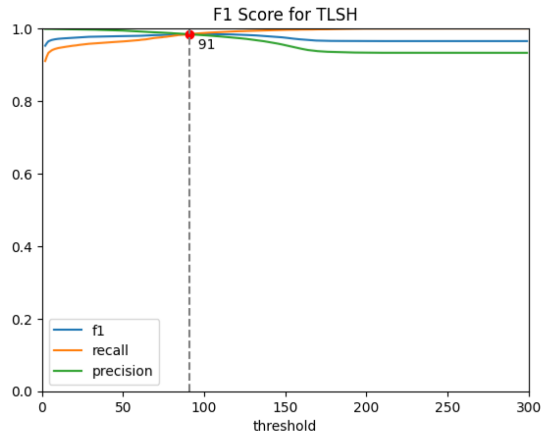


Figure 5.6: F1 score of TLSH

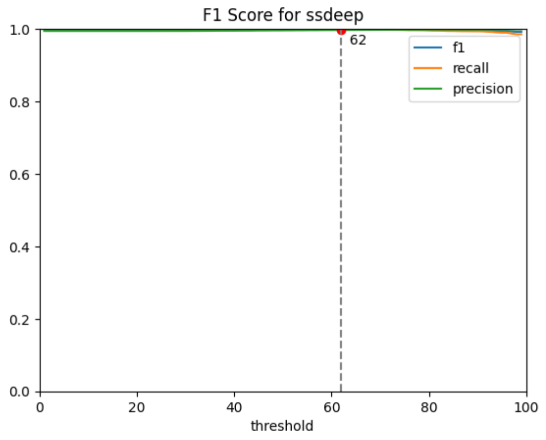


Figure 5.7: F1 score of ssdeep

For the malware dataset, TLSH precision decreases after a threshold of over 80-90 and recall does not increase significantly more. The same is true for ”Strings” but with a threshold of 90-100. For ssdeep, recall starts to decrease after 90-95 and precision stays constant.

5.3.2 Scicore

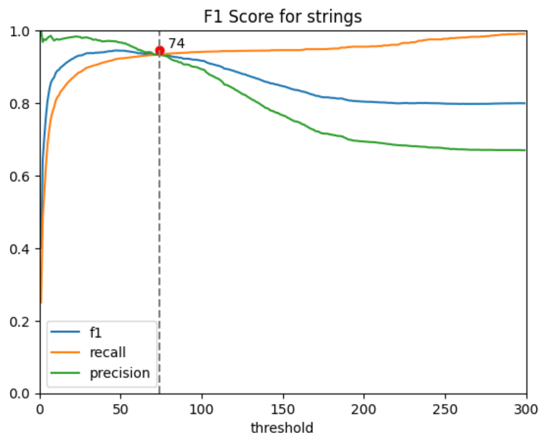


Figure 5.8: F1 score of Strings

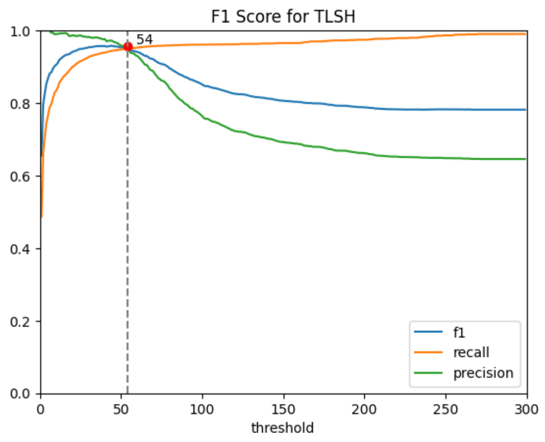


Figure 5.9: F1 score of TLSH

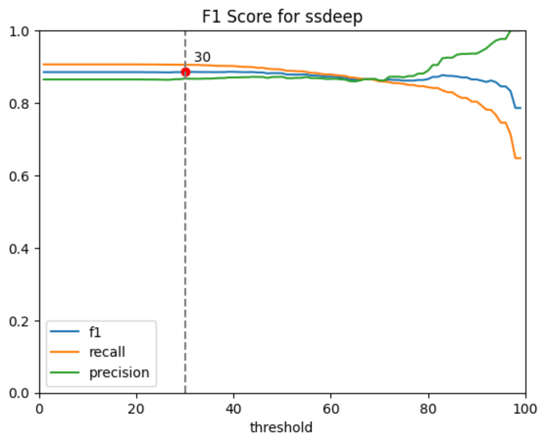


Figure 5.10: F1 score of ssdeep

On the Scicore dataset, the TLSH based fuzzy hashers behave similar, but ”Strings” is shifted to the right. Both TLSH plots decrease in every metric which is not expected. Ssdeep stays the same until a threshold of 70 and then recall decreases while precision increases. The highest F1 score of 0.95 is reached with TLSH.

5.3.3 Reduced Malware

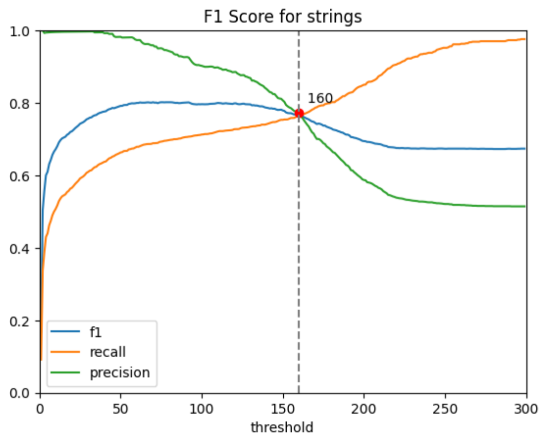


Figure 5.11: F1 score of Strings

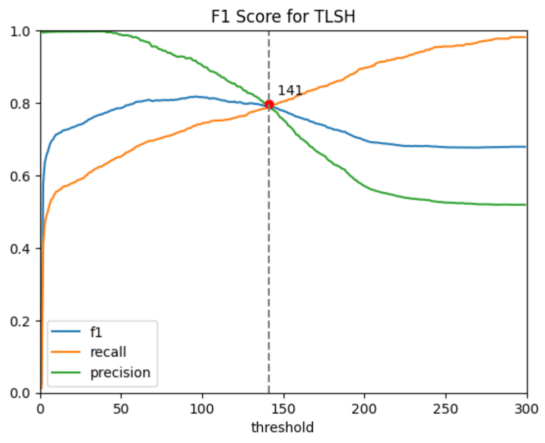


Figure 5.12: F1 score of TLSH

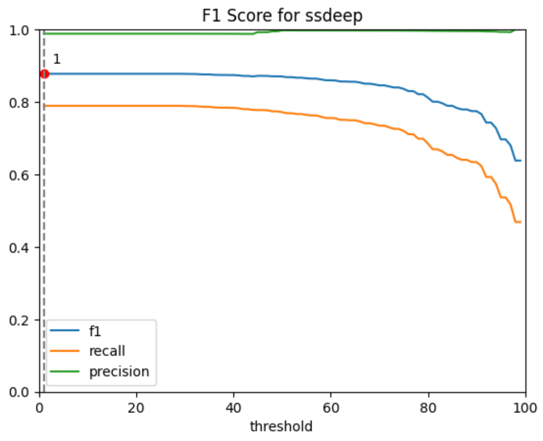


Figure 5.13: F1 score of ssdeep

5.4 Edi Curve

Two dotted lines are visible for each Edi curve. The grey dotted line is placed to indicate which threshold the F1 score has its peak. The blue line indicates the intersection of the precision graph and the total graph, where total stands for the total amount of samples still being considered after the threshold cut.

5.4.1 Malware

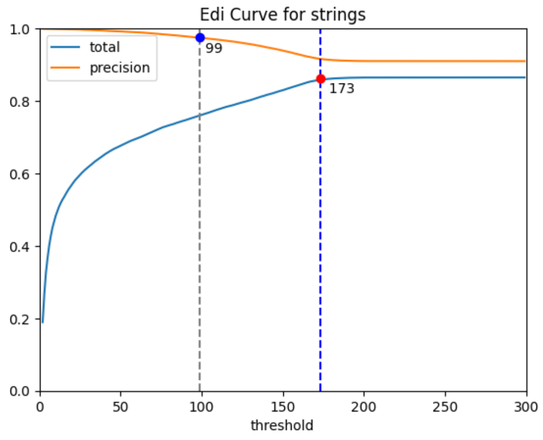


Figure 5.14: Edi curve for Strings

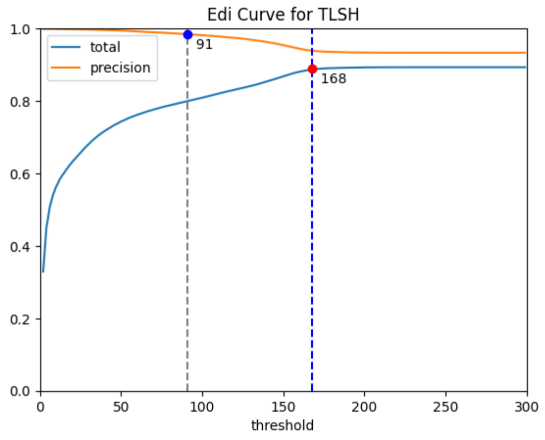


Figure 5.15: Edi curve for TLSH

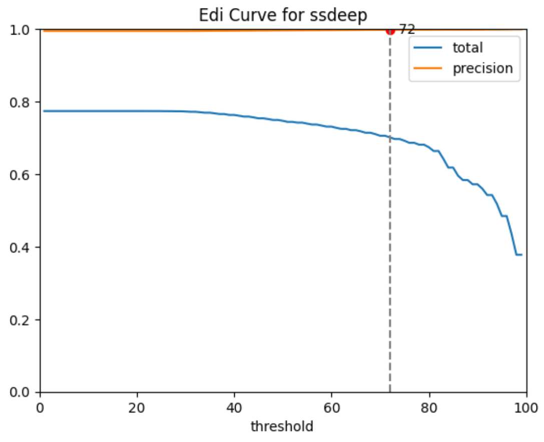


Figure 5.16: Edi curve for ssdeep

5.4.2 Scicore

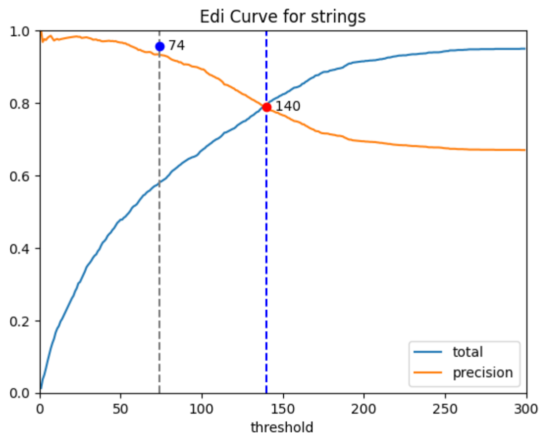


Figure 5.17: Edi curve for Strings

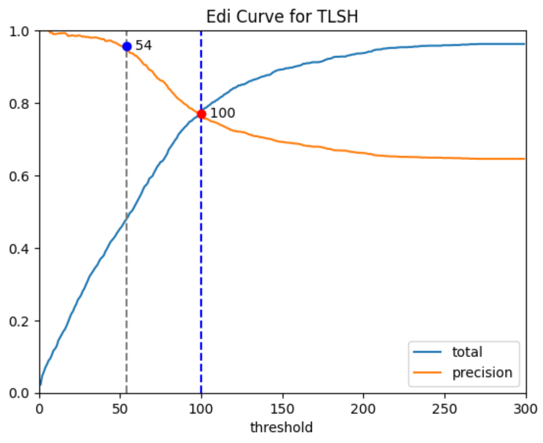


Figure 5.18: Edi curve for TLSH

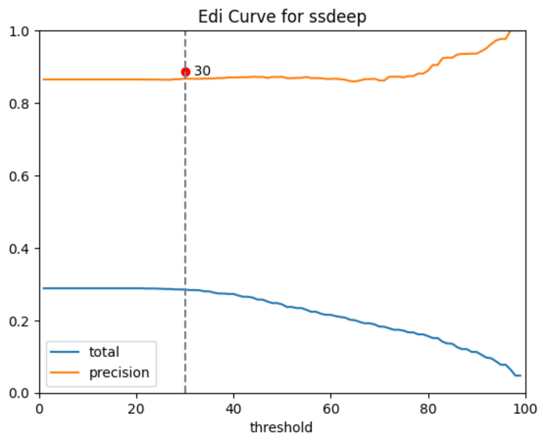


Figure 5.19: Edi curve for ssdeep

5.4.3 Reduced Malware

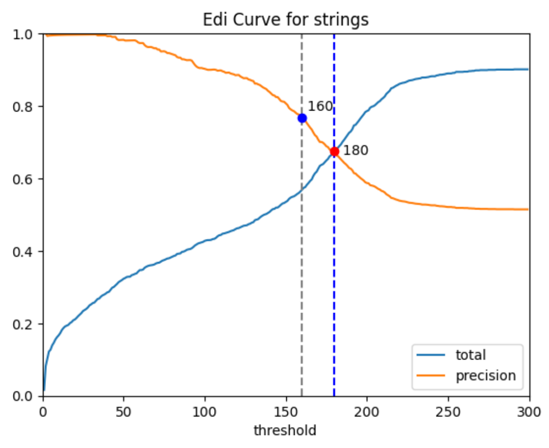


Figure 5.20: Edi curve for Strings

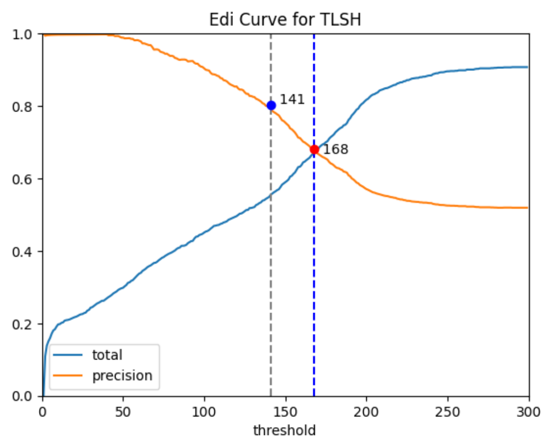


Figure 5.21: Edi curve for TLSH

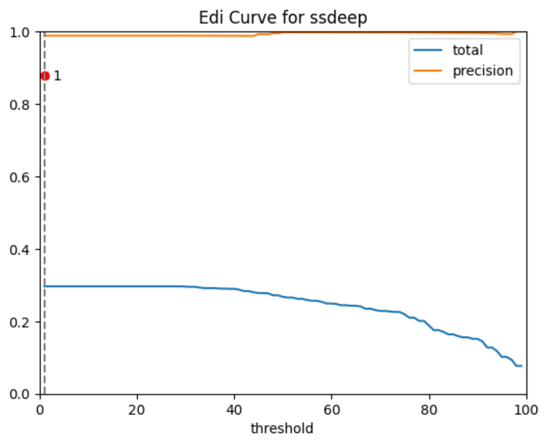


Figure 5.22: Edi curve for ssdeep

Chapter 6

Discussion

6.1 Malware vs Scicore

The vast difference in the results of the ROC curve can be explained by the size difference of the datasets. The Scicore dataset is smaller by a factor of 100 compared to the malware dataset. By doing the analysis with a reduced cut of the samples, the ROC curve of the malware dataset takes a huge hit. This implies another finding. The more samples are hashed in the database, the higher is the performance of the fuzzy hashers in accuracy. This can be rationally explained, because when trying to find comparisons, more similar items help identify the right type.
This is also reflected on the other curves where the malware dataset is shown. The F1 Score and Edi curve are near perfect in close to every metric (besides Machoke).

Since the database of a fuzzy hasher should always be extended, the performance increases with its use. This finding shows that starting with a sparse dataset, fuzzy hashers can not shine from the beginning. Strategies may have to be implemented like setting the threshold to much stricter range to accurately predict, if the dataset is sparse and then increase the threshold when the dataset size has increased.

Ssdeep does not lose its precision even with the reduced malware dataset. TLSH and "Strings" do have a lower precision in the reduced dataset even compared to the scicore dataset. This is an indication that ssdeep performs better on malware or generally on the malware dataset, but with the current results, it can not be fully explained. This has to be analyzed further.

6.2 Thresholds

The ROC curve and F1 Score give a good representation of the performance in each incremental step of the threshold. A very important information, which is not shown in these two statistical tools, is the actual amount of data still being hashed after the threshold cut. We can clearly see in the ROC curve that restricting thresholds has diminishing returns in respect to increasing TP and reducing FP. The missing information about the absolute amount of samples still being processed, is visible on the Edi curve and partially in the F1 score.

Depending on the use case and the requirements, the thresholds for TLSH and "Strings" can be set between 50-150, where a lower threshold means that predictions are more accurate but less samples will be predicted. For ssdeep a threshold between 30-70 can be set, where a lower threshold means that predictions are less accurate but more samples will be predicted. Two distinct use cases are set as context. First is the requirement to identify as many samples as possible and cover the broad scale of samples. The second requirement is being as exact as possible with the prediction and ignoring the amount of samples being predicted but tolerating only the highest TP rate possible.
For Scicore specifically, setting a lower threshold for TLSH will be beneficial, since TLSH still can have a prediction close to 80% of the dataset with an accuracy of close to 80% (see Figure 5.18). The threshold can then increased as the dataset grows. This approach would have the best practical benefit to correctly predict as many samples as possible. The same approach can be applied to the other fuzzy hashers.

6.3 Performance of Prediction

The effectiveness of fuzzy hashers greatly depends on how much data they have to work with. They do best when there's a lot of data available right from the start. However, they can still improve over time as more data gets added to their database, often from the hashers themselves when they find a good match. But, if there are fewer than 1,000 hashed samples, the fuzzy hashers' ability to correctly identify what group a sample belongs to drops significantly.
When it comes to large datasets, such as those used for studying malware, fuzzy hashers are really useful for identifying similar files. Choosing the right one is the challenging part. Based on the analysis of malware datasets through ROC curves, ssdeep stands out because it generally shows better results compared to others.
Despite ssdeep having a slight dip in its F1 score, which suggests a small drop in accuracy, it still scores higher overall compared to others like TLSH and "Strings," which might show higher peaks but are less consistent in their accuracy.
In terms of predicting a larger number of samples with accuracy, TLSH outperforms ssdeep, although ssdeep is more precise in its predictions. Specifically for malware datasets, ssdeep can confidently predict more than 32% of samples. But, for smaller datasets, its ability to classify confidently drops, showing that while ssdeep is precise, it covers a smaller portion of the dataset than TLSH and "Strings," which can confidently classify over 80% of the samples across various datasets.
Looking at specific datasets, ssdeep's accuracy in identifying true positives is lower compared to other hashers. This contrast becomes more apparent in different datasets where TLSH struggles. The size of the dataset often affects this outcome, affecting the balance between accuracy and the ability to identify correct matches. The unique way ssdeep handles unmatched samples, compared to TLSH which requires a user-defined threshold for the same, significantly influences how much of the dataset it can cover.
All the fuzzy hashers can reach an accuracy of over 80%, but they differ in how much of the dataset they can effectively cover. Ssdeep, while precise, is limited in the portion of the dataset it can predict accurately, making it less suitable for smaller datasets. On the other hand, TLSH and "Strings" provide reliable predictions for a larger percentage of samples across different datasets.
Overall, TLSH and "Strings" are objectively better from the results in terms of precision and recall together. "Strings" is very close to TLSH in terms of these parameters, which can be explained by the lost information of only taking the strings of a binary as input to the TLSH fuzzy hasher. With

these results we can conclude that stripping a binary of everything but its strings will worsen the performance of a fuzzy hasher.

6.4 Practicality

With these results, we have shown that, in a theoretical environment, fuzzy hashers (especially TLSH) can perform very well in identifying the family of a binary. This framework was implemented mainly in docker for managing dependencies and quickly running the image, MongoDB as the NoSQL database which hosts the hashed samples and python for linking the applications with hashing the binaries and predicting their families. In a HPC system, this can be implemented as a pre routine before running a SLURM job, given an interface is built to fetch the binary in a SLURM job. Since the login node has internet access, this implementation can communicate with a database in the same network. On a home setup, with a server and client connected to a home router with LAN, it took less than a second (0.3 seconds) to do fuzzy hash, and predict the family of a sample, with an unoptimized script. We will not empiricaly show what the median time is for a full search, because of the interference of other devices in the network and since this is not an isolated environment.

The most barebone version requires only the fuzzy hashers and a lightweight python script for the whole routine. We can see this as very practical to implement and add to current HPC systems like Scicore or miniHPC. The only difficulty is retrieving the binary from a SLURM job.

Chapter 7

Conclusion and Future Work

To summarize the results, TLSH is the overall winner and Machoke is the overall loser. HPC systems can use fuzzy hashers as a static tool to analyse malware and predict with high confidence the right family of a binary. This solution is only relevant if future works include these findings and progress on using fuzzy hashers. This tool alone in itself is not the final solution to the bigger problem of knowing and understanding what runs on a HPC. It builds the fundamental work which can be expanded not only to know and verify future jobs, but to eventually progress into preventing executing of malicious, not intended or not allowed software.

The first problem which has to be solved in a future work is retrieving the main running binaries (or scripts) from a SLURM job and fuzzy hash it. Currently, there is no solution to this problem and SLURM jobs do not always give away which binary is the main one running. If, for example, a SLURM job calls multiple bash scripts to prepare the work, one has to pinpoint the main running application, before the jobs get executed. Doing it afterwards could help by gathering the data of the longest running or most CPU intensive process. Still, if a malicious actor was involved, it would have been too late.

The second problem is keeping the algorithm of retrieving the binary of a SLURM job and predicting a family under the time requirement which Scicore or any other HPC will accept. For some bigger binaries, it took more than 5 seconds of hashing alone. Exceptions can be created to ignore these edge cases, but the risk either has to be accepted or mitigated.

The last problem to solve is using a combination of fuzzy hashers with the right weights to get higher F1 scores. Future research should also look into using fuzzy hashers as features of a machine learning model. TLSH and ssdeep specifically do have distinguishable differences in their hashing ways which possibly could be used together to increase precision.
Ideally, fuzzy hashers like Machoke should be the most accurate in finding similarities, since the binary gets disassembled and analyzed in its context. This is a research area which is not limited to the HPC domain and more associated with the reverse engineering field. The current problem in Machoke is, that only the call flow graph gets analyzed and not the function itslef. The call flow gives decent insight into the architecture of an executable, if no packers are involved. Some compilers, like MinGW, often add the same start call flow which can be seen in some Machoke

hashes. Removing this type of noise and improving the way binaries get compared (deeper than just the call flow graph) could lead to helpful findings.
This research has opened multiple doors to further investigate the powers of fuzzy hashers and their usage. We have shown that fuzzy hashers are not limited to the Cyber Security domain, but can also be used in the HPC domain to solve a difficult task while still respecting the privacy of its users.

Bibliography

[1] 7-Zip. Link: https://www.7-zip.org/.

[2] Machoke. Link: https://github.com/ANSSI-FR/polichombr/blob/dev/docs/MACHOC_HASH.md.

[3] miniHPC. Link: https://hpc.dmi.unibas.ch/research/minihpc/.

[4] sciCORE. Link: https://scicore.unibas.ch/.

[5] Vx Underground. Link: https://vx-underground.org/.

[6] SwissTPH/openmalaria, January 2024. Link: https://github.com/SwissTPH/openmalaria.

[7] Emre Ates, Ozan Tuncer, Ata Turk, Vitus J. Leung, Jim Brandt, Manuel Egele, and Ayse K. Coskun. Taxonomist: Application Detection Through Rich Monitoring Data. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing*, volume 11014, pages 92–105. Springer International Publishing, Cham, 2018. Series Title: Lecture Notes in Computer Science.

[8] Stefan Le Berre and Adrien Chevalier. Démarche d'analyse collaborative de codes malveillants.

[9] Reuben D. Budiardja, Kapil Agrawal, Mark Fahey, Robert McLay, and Doug James. Library Function Tracking with XALT. In *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*, XSEDE16, pages 1–7, New York, NY, USA, July 2016. Association for Computing Machinery.

[10] Sebastian Moss 2 Comments. European supercomputers hacked, apparently to mine cryptocurrency, May 2020. Link: https://www.datacenterdynamics.com/en/news/european-supercomputers-hacked-mine-cryptocurrency/.

[11] Thomas Jakobsche, Nicolas Lachiche, Aurelien Cavelan, and Florina M. Ciorba. An Execution Fingerprint Dictionary for HPC Application Recognition. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 604–608, Portland, OR, USA, September 2021. IEEE.

[12] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3:91–97, September 2006.

[13] Amanda Lee and Travis Atkison. A Comparison of Fuzzy Hashes: Evaluation, Guidelines, and Future Suggestions. In *Proceedings of the SouthEast Conference*, pages 18–25, Kennesaw GA USA, April 2017. ACM.

[14] Huimin Lu, Ming Zhang, Xing Xu, Yujie Li, and Heng Tao Shen. Deep Fuzzy Hashing Network for Efficient Image Retrieval. *IEEE Transactions on Fuzzy Systems*, 29(1):166–176, January 2021. Conference Name: IEEE Transactions on Fuzzy Systems.

[15] Jonathan Oliver, Chun Cheng, and Yanggui Chen. TLSH – A Locality Sensitive Hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pages 7–13, Sydney NSW, Australia, November 2013. IEEE.

[16] Peter K. Pearson. Fast hashing of variable-length text strings. *Communications of the ACM*, 33(6):677–680, June 1990.

[17] Sean Peisert. Fingerprinting Communication and Computation on HPC Machines. Technical Report LBNL-3483E, 983323, June 2010.

[18] Kenneth Reitz. requests: Python HTTP for Humans.

[19] Yuichi Tsujita, Atsuya Uno, Ryuichi Sekizawa, Keiji Yamamoto, and Fumichika Sueyasu. Job Classification Through Long-Term Log Analysis Towards Power-Aware HPC System Operation. In *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 26–34, Valladolid, Spain, March 2021. IEEE.

Appendices

Appendix A

Machoke Scicore Plots

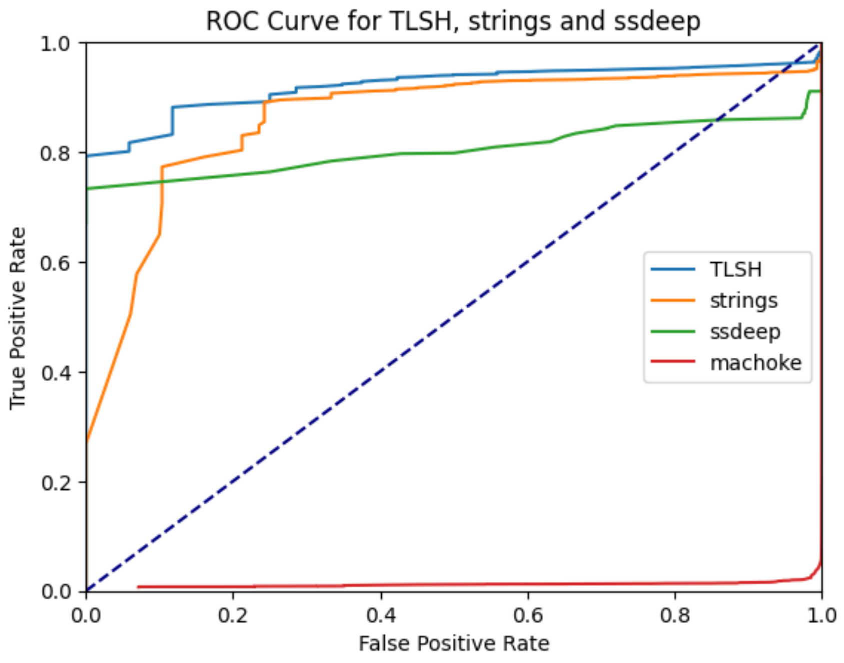


Figure A.1: ROC curve for Scicore dataset with Machoke

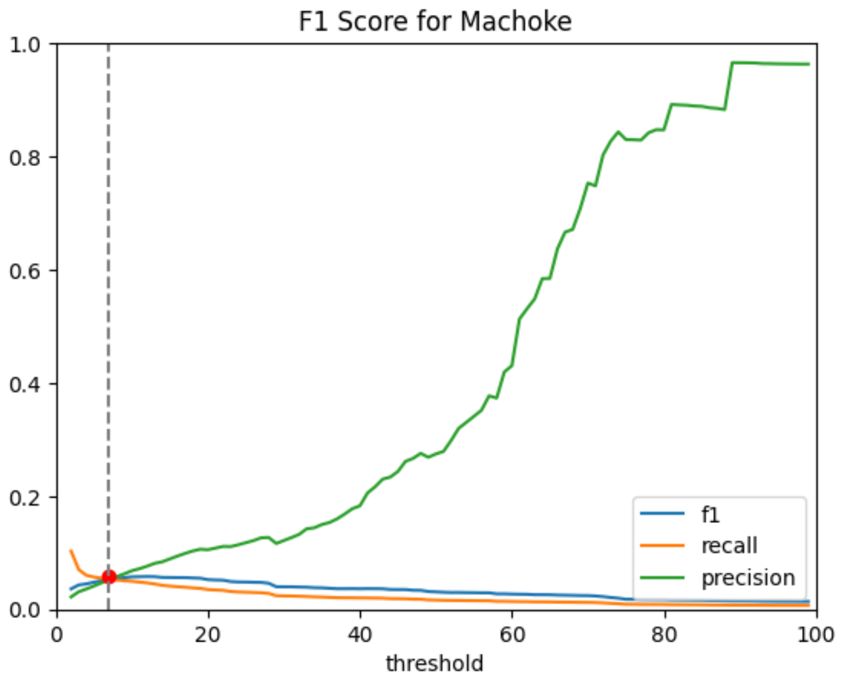


Figure A.2: F1 score for Scicore dataset with Machoke

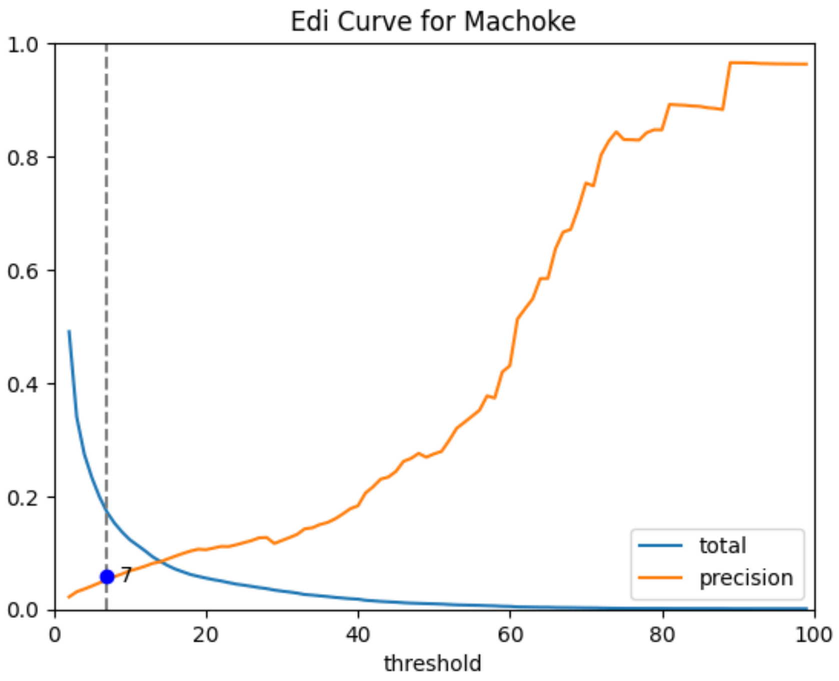


Figure A.3: Edi curve for Scicore dataset with Machoke