BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

SCALABLE BINARY EXECUTABLE FILE SIMILARITY

ŠKÁLOVATELNÁ PODOBNOST BINÁRNÍCH SPUSTITELNÝCH SOUBORŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. PETER KUBOV**
AUTOR PRÁCE

SUPERVISOR **Ing. DOMINIKA REGÉCIOVÁ**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2020/2021

Master's Thesis Specification

Student: **Kubov Peter, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: High Performance Computing
Title: **Scalable Binary Executable File Similarity**
Category: Security
Assignment:

1. Study state-of-the-art methods of detecting binary executable file similarity. Focus on methods utilizing machine code and abstractions over it (i.e., basic blocks, functions).
2. Familiarise yourself with the existing file-similarity technologies used in Avast.
3. Devise a new service to help Avast detect binary executable file similarity on the machine-code level. Focus on the solution's accuracy and scalability to deal with immense amounts of files processed by Avast systems. The comparison itself should be performed on multiple levels. The first level will perform fast and cheap 1:N comparison based on binary sequences. This will find a similar sample set for an inspected unknown sample. The second level will perform sophisticated comparison employing advanced methods based on the semantics of underlying data (e.g., functions, basic blocks, instructions).
4. Implement the service designed in the previous point.
5. Test the service with a set of unit, integration, and regression tests.
6. Use the service on large number of real-world samples and evaluate the results.
7. Judge your solution and discuss future improvements.

Recommended literature:

- P. Szor: The Art of Computer Virus Research and Defense, Addison-Wesley Professional (2005), ISBN 978-0321304544
- Avast internal documentation
- Literature recommended by supervisor and consultant.

Requirements for the semestral defence:

- First three items of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Regéciová Dominika, Ing.**
Consultant: Matula Peter, Ing., Avast
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2020
Submission deadline: May 19, 2021
Approval date: October 23, 2020

Abstract

This work aims to design and implement a new service searching for binary file similarities within known malware samples called YaraZilla. Studying file similarity has a growing potential in malware analysis. The vast amount of new malware is a polymorphic variation of existing malware created for deceiving anti-malware detections. The newly created service is designed to operate by using various binary file similarity techniques on multiple levels of binary code abstraction – instructions, basic blocks, functions. The service is designed to process immense amounts of files supplied by Avast systems. The result of this work is a service that presents malware analysts at Avast with a comprehensive report on malware similarity. Apart from that, the result of service can be integrated into existing services and provides a foundation for new tools.

Abstrakt

Cieľom tejto práce je navrhnúť a implementovať novú službu s názvom YaraZilla pre hľadanie podobností binárnych súborov so známymi malvérovými vzorkami. Veľké množstvo malvéru je iba variáciou existujúceho malvéru upraveného tak, aby unikol pozornosti anti-malvérových systémov. Vďaka rozvíjajúcej sa štúdii podobnosti binárnych súborov sme schopní vytvoriť nástroje, ktoré dokážu odhaliť podobnosť binárne líšiacich sa vzoriek, a zaradiť malvér do rodiny, s ktorou zdieľa najviac podobnosti. Cieľom práce je práve poskytnúť takýto nástroj analytikom v Avaste. Služba, navrhnutá v tejto práci, hľadá podobnosť binárnych súborov využitím rozličných metód na rôznych úrovniach abstrakcie binárnych súborov – inštrukcie, základné bloky, funkcie. Navrhnutá služba je schopná spracovávať obrovské množstvo súborov, ktoré poskytujú interné systémy spoločnosti Avast. Výsledkom práce je nová služba, ktorá poskytuje malvérovým analytikom v Avaste obsiahle štatistiky podobností, ktoré je možné využiť v existujúcich službách alebo ich využiť ako základ pre nové nástroje.

Keywords

Avast, malware, malware family, reverse engineering, binary file similarity, scalable service.

Kľúčové slová

Avast, malvér, malvérové rodiny, revezné inžinierstvo, podobnosť binárnych súborov, škáloveteľná služba.

Reference

KUBOV, Peter. *Scalable Binary Executable File Similarity.* Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Dominika Regéciová

Rozšírený abstrakt

Škodlivý softvér (malvér) predstavuje významnú hrozbu pre všetky druhy počítačových zariadení vo všetkých infraštruktúrach. Techniky používané na vytváranie škodlivého softvéru sa zdokonaľujú rovnako rýchlo ako metódy jeho detekcie. V uplynulých rokoch bola významná časť nového malvéru iba upravená varianta už existujúceho malvéru navrhnutá tak, aby unikla detekcii antimalvérovými produktmi. Takéto varianty malvéru sa líšia predovšetkým syntaktickou reprezentáciou, pričom poskytujú takmer rovnakú funkcionalitu. Skupiny variant rovnakého malvéru sa nazývajú malvérovými rodinami. Ako poznamenali autori článku [4], technológie na detekciu známeho malvéru sú v súčasnosti založené predovšetkým na syntaktických podpisoch (signatúrach). Takéto signatúry špecifikujú binárne inštrukcie alebo dátové sekvencie, ktoré sú charakteristické pre konkrétnu vzorku malvéru. Tvorcovia malvéru sa môžu vyhnúť detekcii tohto typu pomocou rôznych obfuskačných techník, ako sú napríklad techniky opísané v [29]. Takéto techniky sa používajú na vytvorenie polymorfných verzií malvéru, ktoré je následne potrebné spracovať ešte raz. To v konečnom dôsledku stojí ľudský čas výskumníkov. Jedným z riešení, ako zjednodušiť proces detekcie a odhaliť polymorfné varianty rovnakého malvéru, je vytvoriť dôkladnejšiu analýzu založenú na technikách podobnosti súborov.

Táto práca sa zaoberá vytváraním nástroju pre analýzu vstupných vzoriek na základe ich podobnosti s existujúcimi rodinami malvéru. Takýto nástroj je prospešný pre analytikov bezpečnostnej spoločnosti Avast. Konkrétne je cieľom práce zjednodušiť a zlepšiť prácu analytikov vytvorením novej služby pre hľadanie podobnosti binárnych súborov. Práca sa zaoberá vytváraním nástroja, ktorý je schopný podávať podrobné štatistiky o podobnosti binárnych súborov s rodinami malvéru na rôznych úrovniach abstrakcií s ohľadom na škálovateľnosť. Práca kladie dôraz na všeobecnosť a parametrizovateľnosť, tým poskytuje základ pre nový súbor aplikácií vytváraných v spoločnosti Avast.

V prvej časti sa práca venuje ustanoveniu terminológie v problematike podobnosti binárnych súborov. Zároveň sú ukázané rôzne techniky porovnávania binárnych súborov používaných v praxi pre hľadanie podobnosti súborov.

Druhá časť tejto práce je venovaná návrhu novej služby pre hľadanie podobnosti binárnych súborov, špecializovanej na vyhľadávanie podobností malvéru. Návrh takéhoto systému je rozdelený do viacerých špecializovaných modulov integrujúcich špecializované moduly s jasne definovanými zodpovednosťami. Známe škálovateľné mechanizmy hľadania súborov sú integrované na rozpoznávanie podobností na rôznych úrovniach abstrakcie: binárnej, inštrukčnej a funkčnej úrovni. Mechanizmy ako odtlačky senzitívne na lokalitu a Jaccardova podobnosť využívaná pre hľadanie podobných dokumentov. Dizajn sa zameriava najmä na návrh mechanizmov, ktoré dokážu extrahovať esenciálne zložky malvérových rodín. Pre tento účel sú opísané viaceré úrovne filtrácie a predspracovania, ktoré sú integrované do inšpekčného procesu.

Tretia časť práce je venovaná návrhu mechanizmov na extrakciu a ukladanie veľkých súborov dát, ktoré služba potrebuje na svoju prevádzku. Služba pracuje so súbormi údajov zo vzoriek malvéru, ale aj bežných binárnych súborov. Služba je navrhnutá tak, aby ponúkala používateľskú aj automatizovanú extrakciu. Na zabezpečenie filtrácie čistých údajov je extrakčná služba navrhnutá tak, aby fungovala vo veľkom rozsahu s veľkým množstvom binárnych údajov uložených v cache vo forme odtlačkov. Na extrakciu a cache odtlačkov obrovského množstva čistých súborov sú navrhnuté nové mechanizmy na efektívne ukladanie a cachovanie náhodných údajov. Formálne je definovaná nová štruktúra nazvaná pod skratkou IRD, ktorá znižuje pamäťové nároky na ukladanie veľkého množstva náhodných dát na polovicu.

Štvrtá časť je venovaná implementácii služby a opisu navrhnutého automatizovaného procesu nasadenia. Účelom automatizovaného procesu je vytvoriť a zabezpečiť škálovateľnosť nových cloudových systémov pre internú sieť spoločnosti Avast. Navrhnutý je všeobecný postup nasadenia, pričom každý krok poskytuje automatizáciu iného procesu implementácie, testovania a nasadenia. V závere kapitoly sa nachádza úvod do implementovanej služby a jej využitia.

Posledná časť tejto práce je vyhradená na testovanie a vyhodnotenie implementovanej služby a nového databázového riešenia. Nová služba je podrobená hodnoteniu pomocou rôznych prístupov. V prvom rade je funkčnosť služby overená na pripravenej sade príkladov porovnaním výstupu s nástrojom Diaphora. Ako druhé, služba bola použitá na spracovanie rozsiahlej zbierky malvérových súborov získaných z interných služieb Avastu a Malpedie. Je ukázané, že nová služba je využiteľná navrhnutým spôsobom a parametrizáciou je možné skvalitňovať výsledky inšpekcií. Okrem služby sa testuje a meria aj funkčnosť nového riešenia na ukladanie veľkého množstva náhodných údajov. Je ukázané, že vytvorené databázové riešenie je schopné spracovávať a poskytovať výsledky dostatočne rýchlo pre využitie v internej sieti Avastu.

Scalable Binary Executable File Similarity

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Dominika Regéciová. The supplementary information was provided by Ing. Peter Matula. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....

Peter Kubov

May 19, 2021

Contents

Chapter 1

Introduction

Malicious software (malware) poses a significant threat to all kinds of computer devices in all infrastructures. The primary purpose of malware is to harm while not being detected. Techniques used for creating malware are improving as fast as methods for its detection. In the past years, a significant part of new malware has been an altered variant of pre-existing malware designed to escape detection by anti-malware products [12]. Such malware variants differ primarily in syntactic representation while providing almost the same functionality. Such groups of malware variants are called malware families. As authors of paper Measuring similarity of malware behavior [4] noted, technologies to detect known malware are nowadays primarily based on syntactic signatures. Such signatures specify binary instructions or data sequences that are characteristic of a particular malware sample. Malware creators can avoid static detections using various obfuscations techniques, as described in [29]. Such techniques are used to create polymorphic versions of the same malware that need to be re-processed once again, costing time of malware research. One solution to simplify the detection process and expose polymorphic variants of the same malware is to create a more thorough analysis based on file similarity.

This work aims to provide a tool to analyze input samples based on their similarity with known malware families. Such a tool is beneficial for analysts at the security company Avast. Specifically, the work aims to simplify and improve analysts' work by introducing a new file similarity service. The new service can provide detailed statistics on binary file similarity with malware families on various levels of abstraction and is designed with scalability in mind. The goal is to help malware analysts categorize, quickly identify, trace the evolution in time, and uncover the possible severity of a malware sample. Apart from that, the new service is general enough to provide a foundation for a new set of applications created at Avast. One particular use case is an automated generation of statical signatures for malware detection.

This work is divided into four parts. The first chapter provides an introduction to the binary file similarity area. The chapter defines different types of binary code comparisons and describes techniques used in practice to support file similarity. The chapter also introduces the file similarity as a service with references to the market's existing solutions.

The second part of this work is dedicated to designing a new binary file similarity service specialized in searching for malware similarities. The design is delegated into multiple specialized inspection pipelines integrating specialized modules with well-defined responsibilities. Well-known scalable file inspection mechanisms are integrated to recognize similarities on various levels of abstractions: binary, instruction, and function level. The design is especially keen on designing mechanisms that can extract the essence of malware families.

For that multiple levels of filtration and pre-processing are described and integrated into the inspection pipeline.

The third part of the work is dedicated to designing mechanisms for extraction and storing large datasets the service needs for operation. The service operates with datasets from both malware and non-malware samples. Extraction processes are designed as pipelines integrated into the service to provide extraction of datasets on a scale. The service is designed to offer both user-wise and automated extraction. To provide filtration of clean data, a new service is devised to operate on a large scale with many cached binary data in the form of hashes. For the extraction and caching of enormous amounts of clean files, new mechanisms for effective storing and caching random data are devised. A new structure called Indexable random data structure is formally defined to cut the memory requirements of storing a large amount of random data in half.

The fourth part is devoted to the service implementation and description of a devised automated deployment process. The purpose of an automated process is to ensure the scalability of the new cloud-based systems for Avast's internal network. A general deployment pipeline is designed, with each step giving automation of a different implementation, testing, and deployment process. In the end, the chapter provides an introduction to the implemented service and its usage.

The last part of this work is reserved for testing and evaluating the implemented service and the new database solution. The new service is put under evaluation with different approaches. Firstly, the service's functionality is validated on a prepared set of examples by comparing its output to a binary diffing tool Diaphora. Secondly, the service is used to process an extensive collection of malware files gathered from Avast internal services and Malpedia, and the result is examined. Apart from the service, the new solution for storing a large amount of random data is tested and measured in operation. Lastly, the conclusion of implemented techniques is provided, and the following progress is opened for discussion.

Chapter 2

Executable File Similarity

Comparing binary files such as executables is critical in scenarios where the application source code is not available, for example, malware analysis. Identifying similarity in binary code is, however, a challenging task as much of the program semantics are lost in the compilation process (function and variable names, comments, data structure definitions, etc.). Additionally, even when the program source code does not change, the binary code may differ on recompilation. For example, different compiler versions may use various optimizations. Furthermore, developers of malware may use obfuscation transformations that can be applied on both the source code and the generated binary code, hiding the logic of the original code. More about the code obfuscation in [29].

The purpose of this chapter is to provide a summarization of state-of-the-art methods used for finding binary code similarities. Provided information were extracted from a file similarity survey performed by Irfan Ul Haq and Juan Caballero in [15].

2.1 Binary Code Similarity

Approaches for finding binary file similarities are based on comparison of binary code. Such a comparison may take various forms based on comparison type, granularity and cardinality. This section establishes definition of different kind of comparison properties used later in the text.

2.1.1 Comparison Type

Generally, we can distinguish between three types of comparisons: identity, equivalency, and similarity. These three comparison classes are hierarchically organized as illustrated in Figure 2.2. The comparison classes are then defined as following code relations.

- **Identity**. Two or more compared pieces of code are identical if they share the same syntax. Binary code can be represented in different ways, such as a string of raw bytes, a sequence of disassembled instructions, or a control-flow graph. Regardless of the form, we can take two binary code pieces and determine whether they are identical or not. This approach, however, fails to detect similarity in many cases. For example, compiling the same file twice may produce two binary files that contain different binary code parts. This happens because compilers pack different kinds of metadata into the final executable such as the compilation date and time.

- **Equivalency**. Two or more compared pieces of code are equivalent if they share the same semantics, i.e., if they grant the same functionality. Clearly, two identical pieces of binary code will have the same semantics, but different binary code pieces may as well. For example, let us consider a trivial example in Figure 2.1. Without optimizations, a compilation of these examples would most probably produce two equivalent binary codes that are not identical. Similarly, a compilation of the same source code for two different architectures with different instruction sets would create equivalent executables with diametrally varying binary codes. In practice, determining binary code equivalence is a costly process. The problem of deciding whether two programs are functionally equivalent is an undecidable problem that reduces to solving the halting problem [17].

```
1 ...                    1 ...
2 i = 0;                 2 i = i-i;
3 printNumber(i);        3 printNumber(i);
4 exit(0);               4 exit(0);
```

Figure 2.1: Equivalent, non-identical C codes.

- **Similarity**. Two or more compared pieces of code are similar if they have similar syntax, structure, or semantics. Syntactic similarity compares the code representation and looks for approximate identity. Structural similarity compares graph representations of a binary code and looks for morphisms. It sits between syntactic and semantic similarity. Semantic similarity compares the code functionality. A simple approach to finding semantic similarity is to look for system calls and compare results. However, this simple approach is not sufficient as two programs with similar system calls can perform significantly different processing on their input. Section 2.3 further examines similarity approaches.

Hierarchical organization illustrated in Figure 2.2 means that if two pieces of code are identical, they are also equivalent and similar. On the other hand, if compared parts of code are equivalent (or similar), they might not be identical.

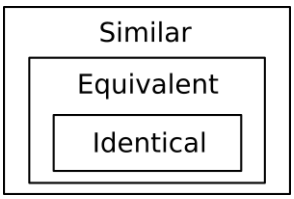


Figure 2.2: Comparison types as sets.

2.1.2 Comparison Granularity

We can compare parts of binary code at different granularities. The commonly compared fundamental binary code abstractions are: instructions, basic blocks, functions, and also programs as a whole. Comparison at a higher level of abstraction (e.g., basic blocks) is often performed by combining different types of comparisons at a finer level of granularity (e.g., instructions). For example, let us consider the situation illustrated in Figure 2.3. Firstly, basic blocks are searched for equivalent and identical instructions. Results of instruction

comparison are then used to deduce the equivalency of the basic blocks. As no information about other basic blocks is provided, it can only be assumed that the given functions are similar.

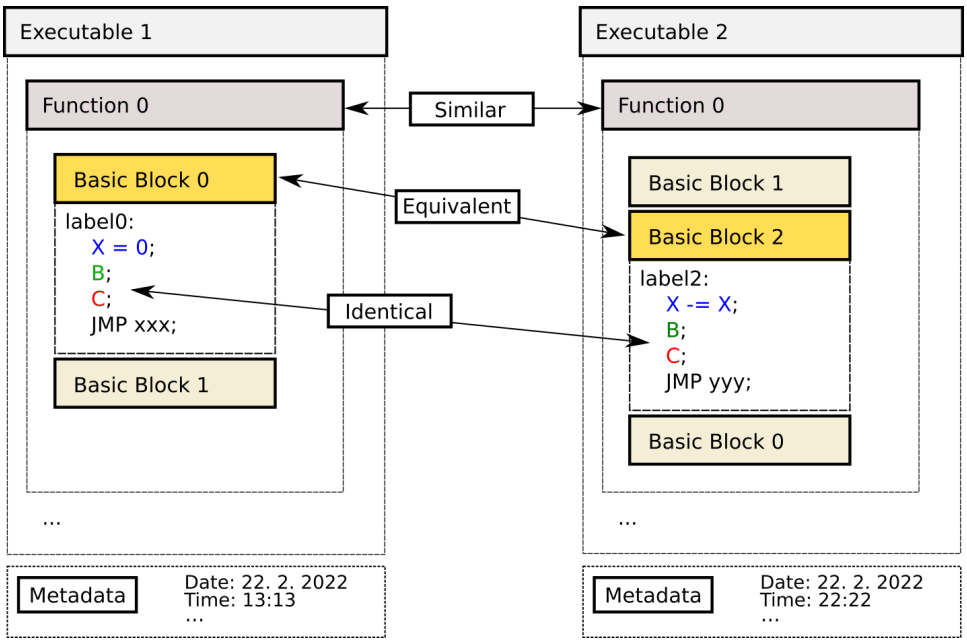


Figure 2.3: Example of file similarity examination.

Figure 2.3 also illustrates that applying a specific comparison at a lower abstraction level restricts the type of comparison at a higher abstraction level. This is shown on basic block comparison, where not all instructions in compared basic blocks are identical. Not having all instructions identical implies that given basic blocks cannot be identical either. Such a relationship between comparisons on different abstraction levels is illustrated in Figure 2.4.

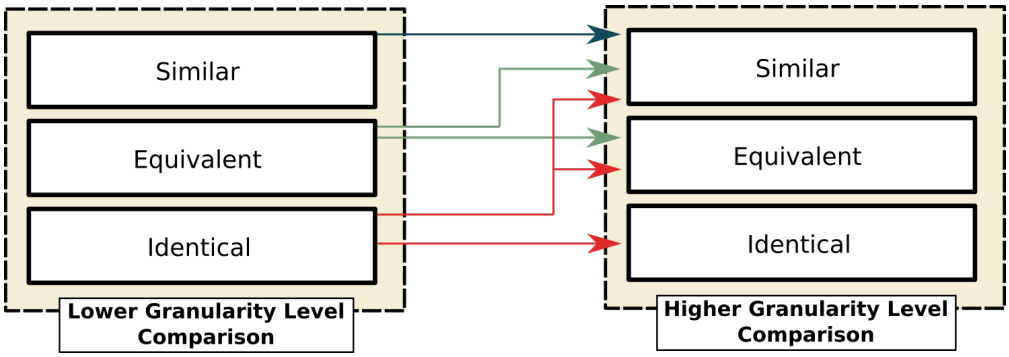


Figure 2.4: Similarity class implications based on lower abstraction level comparison type.

2.1.3 Comparison Cardinality

Based on number of inputs (pieces of code) and how they are compared we define following three comparison cardinalities.

1. **One-to-one** comparisons take two pieces of code (of certain granularity) and compare one to another. The most common approach is to perform binary code diffing – a process that consists of taking two consecutive, or close, versions of the same program to identify what was added, removed, or modified in the subsequent version. The binary code diffing has usually function level comparison granularity, and the process tries to obtain mapping between a function in the original program and another in the target program.

2. **One-to-many** approaches take a piece of code and compare it to a set of pieces of different code versions. As a result, the approaches return a new set of similar pieces of code. Comparison can be applied on a different level of granularity and even on codes compiled for different architectures.

3. **Many-to-many** approaches do not distinguish between source and target pieces and take a set of compared binary code pieces on input. All input pieces are considered equal and compared against each other. The result is typically used for binary code clustering.

2.2 Binary Code Similarity Applications

This section emphasizes the importance of binary code similarity by describing its various applications in practice. As the popularity of binary code similarity increases, new applications may be eventually identified.

- **Bug search** – Let us consider a scenario when a bug in a closed-source program emerges. Due to code reuse, sections of buggy code might have been reused through the repository in various places. It is critical to search for similar code in order to identify all the possibly affected places. Bug search approaches perform one-to-many similarity comparisons throughout the binary code in the repository. Such comparisons are often performed at a function level granularity. More about bug search applications of binary files similarity can be found in [22] and [21].

- **Malware detection** – A malware can be detected by taking an executable file and comparing it to a set of previously detected malware samples. With high binary code similarity the compared sample is likely a variant of the same malware family. Malware detection is therefore performed by one-to-many similarity comparisons on a executable-file-level granularity. More about exploiting malware detection based on binary code similarities can be found in [25].

- **Malware clustering** – The many-to-many approach of executable file comparisons create clusters of similar known malicious executables belonging to the same malware family. Such family clusters contain executable files of the same malware in different versions and modifications. More details about malware clustering is available in [16].

- **Malware lineage** – Lineage approaches construct a graph from a given set of executables known to belong to the same malware. Nodes of the graph represent malware versions, and edges capture the evolution of malware in time. Linage approaches are particularly useful with malware as no official versioning is typically available. More about malware linage and how is it used to detect new malware in [19].

- **Patch generation and analysis** approaches are applied, for example, in distributions of updates of closed-source software. One-to-one identity comparations of two consecutive versions are used to identify what was patched in a newer version of the program. As a result of the comparison, small binary patches are created that are used to effectively distribute software updates without the need to disclose patched source code. On the other hand, this approach is also used from the malware creators side to compare two consecutive versions of software in order to exploit vulnerabilities of the older version. More about patch generation provide authors of [7].

- **Porting information** – Some reverse engineering frameworks use the information porting approach. Malware analysis is an expensive task, but tools can reuse final results with different malware variations once a researcher completes it. Tools often perform one-to-one similarity comparisons to search for similar parts within analyzed files in their database. When an already examined file is matched, pieces of analysis information are ported to the input file. An example of porting information of profiled data can be found in [28].

- **Software theft detection** – In practice, one-to-one similarity comparisons are also used to detect unauthorized reuse of code, such as:

  – a program uses binary parts of stolen source code,

  – a program implements patented algorithm without license,

  – a program violates license of reused parts (e.g., GPL) and so on.

  Example of applying binary file similarity to reveal plagiarism is demonstrated in [20].

2.3 File Similarity Comparison Approaches

This section examines possible approaches used in practice to deal with certain aspects of file similarity issues. Binary file similarity applications differ in methods used for computing similarity, the architecture they support and the process they use to obtain comparable pieces of code - the problem of normalization.

- **Syntactic similarity** approaches compare the representation of two or more binary codes. The typical approach is to sequence consecutive bytes of instructions and compare them between the files. The compared sequences of bytes may have variable lengths but more common is to compare sequences with a fixed size. Fixed-sized sequences of bytes are extracted by sliding a window sequentially over the bytes of the input file, see Figure 2.5. The window size defines the number of bytes captured. The window is slid over a byte stream based on the stride size. Suppose the stride is smaller than the window size, then the consequently extracted sequence overlap. Sequences extracted with a stride of size one and size of the window n are called n-grams.

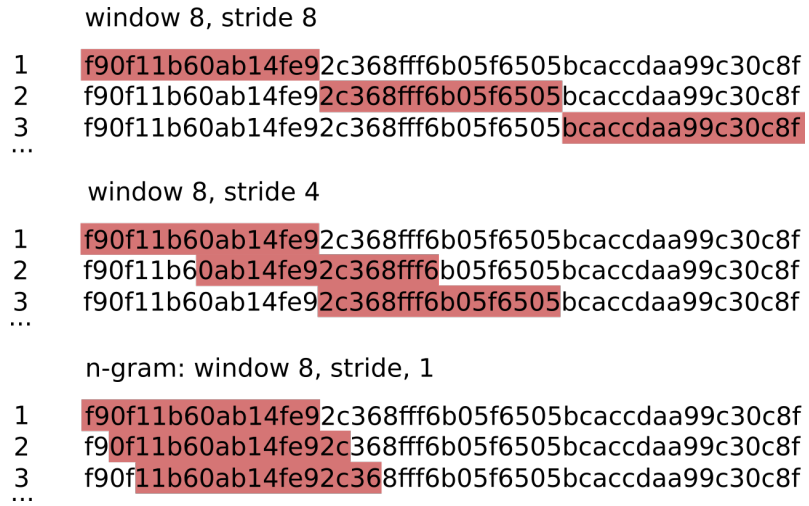


Figure 2.5: Sequences capturing window.

- **Semantic similarity** captures similarity by comparing the behavioral effects of two or more input files. A code's behavior can be described by simulating it and analyzing its effects on the simulation environment, e.g., register usage, memory manipulation. Following three methods are commonly used to capture semantics: instruction classification, input-output pairs, and symbolic formulas.

  1. Instruction classification methods divide instructions into a similarity vector. Each vector member represents instruction type (e.g., arithmetic instructions, logical instructions, memory load). The value represents the number of seen instructions of a particular class. Similarity vector created for a basic block captures semantic effects of this basic block. The instruction classification method is only a heuristics, and such a method cannot capture binary code equivalency.
  2. Input-output pairs methods execute binary input files on the prepared set of inputs and compare their outputs. Based on the number of matches and size of the input set, these methods can provide the likelihood of equivalency.
  3. Input binary code can be transformed into a symbolic formula form. Symbolic formula form is a special form of code representation that captures control and data flow of the program. Such a representation is more suitable for defining formal algorithms to check for similarity or proving equivalence. Equivalence of two pieces of code can be proved by theorem provers, such as STP [14].

- **Structural similarity** has elements of both syntactic and semantic analysis. The structure of input binary code is captured in a graph. A graph structure can typically capture a syntactic representation of the same code, and nodes can be annotated with semantic information. The following three directed graph structures are used for finding binary code similarities:

  1. intra-procedural control flow graph (CFG),
  2. inter-procedural control flow graph (ICFG),
  3. callgraph (CG).

Nodes in CFG and ICFG are basic blocks, and edges indicate control flow transition (e.g., jump). The difference between ICFG and CFG is that basic blocks in a CFG belong to a single function, while basic blocks in ICFG can belong to any program function. On the other hand, nodes in CG are functions, and edges capture a caller-callee relationship.

- **Feature-Based Similarity** - A standard method for finding binary code similarities is to use machine learning. In such methods, compared pieces of binary code are represented as a feature vector, and there is a model that determines whether two or more feature vectors are similar. A feature vector can have syntactic, semantic, or structural properties. Apart from that, a feature can be numeric or categorical. Categorical features have discrete values, e.g., mnemonic of an instruction. Features of a vector are chosen beforehand by an analytic in a feature selection process. Alternatively, features can be generated automatically in a learning process from training data. The result of a generation process is a real-valued feature vector, called embedding.

- **Hashing** - A hash function maps data of arbitrary size to a fixed-size value. A properly chosen hash function can reduce necessary computer resources for working with binary data. Even though hashes are not specifically designed for binary code, there are three classes of hashes suitable for operation on a raw-byte level:

  1. cryptographic hashes,
  2. locality-sensitive hashes,
  3. executable file hashes.

  Cryptographic hashes capture identical inputs and can be used for capturing similarities on a higher level of abstractions (e.g., basic blocks, functions). Cryptographic hashes are sensitive, and a slight change in input results in an immense change of the hash value. On the contrary, locality-sensitive hashes produce for similar inputs similar hash values. Executable file hashes take a whole executable file on input and compute a hash based on parts of its data. Hashes of such type try to capture the polymorphic structure of malware.

- **Normalization**

  A disadvantage of syntactic approaches is that slight change in instruction syntax triggers diverging comparison on higher abstraction levels. To solve this, approaches using syntactic comparisons use a technique called normalization. Normalization is a process of refining input binary code so that minor differences do not affect the overall comparison result. The following three techniques are used for normalization on instruction-level:

  1. **Operand removal** - This technique ensures, that two instructions of the same mnemonics are compared identical regardless of their operands type. The binary code on input is therefore processed before comparison in a way, that all instruction operands are replaced by the same placeholder.
  2. **Operand normalization** - Akin to the operand removal normalization, this technique captures the similarity of instructions by altering their operands. The difference is, however, that operand normalization preserves a type of instruction

operands, such as information whether operand was a register, constant or a value loaded from memory.

3. **Mnemonic optimization** - Sometimes, it is not feasible to differentiate between the same instruction category (i.e., branch and jump instructions). For example, two programs may have similar control flow structures but differ in computing the final output value. Such similarity can be captured by joining all arithmetic and logic instructions into the same category represented by one mnemonic.

Apart from syntactic approaches, normalization can be used on a semantic level as well. Comparison algorithms can reorganize instruction sequences to increase syntactic approach accuracy. Apart from that, pre-processing can remove excessive instructions such as instructions not affecting overall functionality. For example, compilers generate instructions that do not affect the program state, such as no-op instructions, to force memory alignment.

2.4 File Similarity as a Service

The previous section introduced a few examples of binary file similarity applications. The main disadvantage of the introduced approaches is that the more accurate and thorough the approach is, the more computational resources it requires. Therefore, the most accurate tools are available only to users with satisfactory computer performance. One possible solution used in practice is to outsource computer performance to servers or clouds. Specifically, with malware analysis applications, a few binary file similarity applications are available for analytics or businesses. In the following text, the KTAE service [3] is provided as an example; however, other proprietary services are also available, as [13], [26] and [1].

Kaspersky Threat Attribution Engine (KTAE)

An antivirus company Kaspersky created a file similarity service specialized for threat attribution called KTAE [3]. KTAE is a proprietary service that is available only for business usage. The service is capable of providing timely insight into the malware's origin and its possible authors. The tool is built on an extensive database created from years-long research of malware of advanced persistent threats (APT). The service divides input samples on n-grams called genotypes and looks for code similarity with previously analyzed APT samples and related actors. As a result, the service computes the reputation of the input sample and provides a detailed report on the malware history.

2.5 File Similarity Technologies at Avast

The purpose of this section is to summarize technologies used for comparison of malware files and their use case at Avast. Currently, malware researchers at Avast are utilizing binary file similarities in three major ways:

1. Malware detection – A primary technology used for a malware detections are YARA rules (see Section 2.5.1). The approach utilizes syntactic comparisons of binary files with a set of identified sequences and strings found in malware files by researchers. To create the set an extensive work of malware researchers is required anytime a new malware family emerges.

2. Malware clustering – A malware clustering approach is used to group a similar malware samples. Tha approach is used to soften the burden of analysing every single malware sample that emerges on market. The major service for malware clusterization is a called Clusty (see Section 2.5.2) that utilizes syntactic and semantic comparison of samples behvior.

3. Clean samples identification – Another possible utilization of binary files similarity is to create a service that can match parts of binary file with a database of clean samples. There are a few services at Avast that look for identical binary sequences in inspected samples. Approaches used for this use case utilize either hashing, when the binary file is divided into several parts, each hashed and stored for reference, or they match large quantity of binary data stored in database, see Section 2.5.3.

2.5.1 YARA Rules

YARA rules represent mechanisms used to identify files by creating rules that look for specific characteristics. The concept of YARA rules was originally developed by Victor Alvarez and is mainly used in malware research and detection. The main idea of YARA rules is to describe a pattern that identifies particular strains or entire families of malware. More information about YARA rules can be found in [5]. Rules have a hierarchical structure illustrated in Figure 2.6. Each rule has a name, and its body is divided into the following sections.

1. The essential section of a YARA rule is the condition section. The section specifies a Boolean condition to match the investigated file. Figure 2.6 shows a rule is required to find three strings to match a file. The condition can also include other YARA rules.

2. The string section is optional. The authors of rules specify strings or raw bytes in the section that can be used in the condition section. Figure 2.6 shows a rule that defines three string values in ASCII representation.

3. The metadata section presents a way to specify additional information such as the author's name or rule's description. Having stored additional data is useful when several YARA rules are applied on files to determine which rules and why they provided a match.

```
rule nice_program
{
    meta:
        created     = "01/01/2021 13:13:22"
        author      = "HumansAreGreat"
        description = "Search for nice words."
    strings:
        $textstring1 = "thanks" ascii wide nocase
        $textstring2 = "awesome" ascii wide nocase
        $textstring3 = "sorry" ascii wide nocase
    condition:
        $textstring1 or $textstring2 or $textstring3
}
```

Figure 2.6: A YARA Rule example.

There are a few tools in Avast that can generate YARA rules automatically by finding intersections of binary data. These approaches, however, require analyst's time to analyze the similarity of provided files beforehand.

2.5.2 Clusty

Clusty is an internal Avast service for automatic analysis and clustering of newly incoming samples. At Avast, researchers receive between 300 000 — 1 000 000 new samples each day. Some of these samples are malicious, while others are safe. Manually analyzing the samples, one by one, would be tedious and would require the conjoint work of many analysts. Even then, analysts processing samples one by one would find themselves discovering similar samples over and over again. The work of Clusty is to make classification work for analysts simpler by automating the process of grouping similar samples together.

On input, Clusty takes a feed of input samples and from each sample extracts specific properties. These properties are then used to place input samples into a best-suited cluster. Clusty supports various file types (e.g., PE, ELF, Mach-O, APK, archives, Office documents) and looks for a wide range of properties (e.g., static properties, dynamic behavior). Apart from its analysis, Clusty also uses YARA rules and antivirus detections to detect the class of input samples. Clusty also provides analysts with the option to analyze clusters and vote on them, thus classifying files in clusters manually.

Created clusters are typically input for detection generators, which can utilize the fact that a cluster is composed of many samples. Such generators can create YARA rules or antivirus detections. In practice, clustering results in better detection definitions that cover many samples at once.

2.5.3 Avast Cleanset

Services at Avast providing cleanset detection can be grouped into the two categories:

1. Hashing – Services utilize hashing to capture similarity of input files. To do that, they build an extensive database of hashes created from clean files. Hashes are extracted from the input file by dividing it into several parts of predefined size. As a file is divided into a fixed number of parts, one hash typically represents multiple functions. This approach is built to quickly provide results and minimize the number of false positives (the file is clean).

2. Elaborate bytes matching – Services at Avast may query selected bytes in the extensive database of clean byte sequences. The database was created by parsing a vast amount of clean files. Even though the approach is much slower than hashing, it provides better results on finer granularity.

2.5.4 Summarization

Currently, the tools for finding malware similarity used at Avast are either dependent on malware researchers' extensive work or use a limited amount of data from files, like its type and various metadata. Also, automated tools operate only on supported architecture and file format files, typically only PE files of x86 architecture. There is no advanced way a researcher can search for the specific similarity of an analyzed sample with already analyzed files. The purpose of this work is to create such a tool that would enable a malware researcher to trace parts of binary code with already analyzed malware families and to

provide a foundation for the creation of new tools that can build on binary file similarity information.

Chapter 3

YaraZilla File Similarity Service

Chapter 2 introduced mechanisms for file similarity applications. In the following text, the terminology is used to specify requirements and design a new file similarity service for malware analysis – YaraZilla service. Figure 3.1 shows a top-level design of the file similarity service. The service provides an extensive report on binary file similarity with samples of real-world malware families. It provides a graphical user interface and guarantees sufficient computational resources for its users. The service operates in three phases.

1. A user uploads an executable file and specifies comparison granularity.
2. The service processes the uploaded file and searches for similarity matches with already processed executables.
3. As the result, the service returns analysis report specifying a list of similar malware families.

Furthermore, the report analysis is general enough to be used as input of additional tools and frameworks. One possible application for such a file similarity report is an input for an automated YARA rules generator.

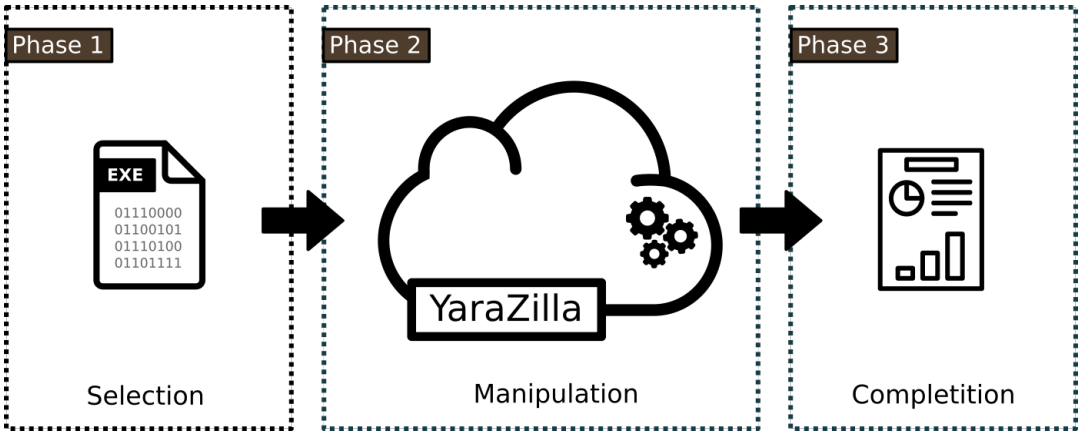


Figure 3.1: YaraZilla malware similarity service.

3.1 Comparison Mechanisms

The created service is expected to compare binary files on three levels of abstractions – binary code sequences level, basic-block level, and function level. Users of the service provide granularity comparison specifications during the file upload phase. The service is designed to have a modular structure illustrated in Figure 3.2 to offer various levels of comparison on demand. Each consequent module is an individual unit, and its analysis works with the output of the previous module. This is because comparisons on a higher level of abstraction are generally more resource-demanding and not always needed by malware analysts at Avast. The optimal approach is to create a service that provides the needed results in an optimal response time and provides more contextual results on demand.

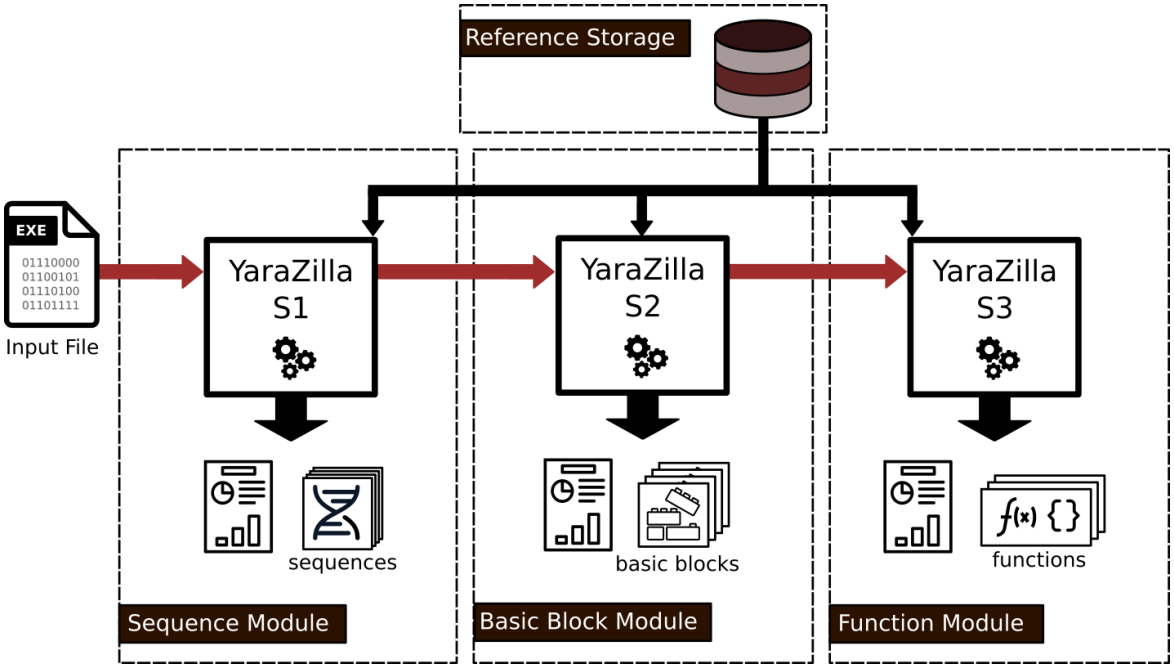


Figure 3.2: Each YaraZilla module is responsible for a different comparison granularity.

3.1.1 Sequence Module

The sequence module takes a binary file sample and provides cheap syntactic identity comparisons on the binary level. The module works by comparing 16-byte n-grams called sequences. As specified in Section 2.3, n-grams are extracted from the input file by moving a 16-byte window with a stride one on the input binary. A 16-byte size of an n-gram was chosen so that the module can be independent of a specific instruction set architecture (ISA). The 16-byte size is sufficient enough to cover instruction sizes of all major ISAs on the market – both x86 and x86-64[1], but also ARM and ARM64[2]. The module is designed to operate as a pipeline shown in Figure 3.3.

[1] `https://wiki.osdev.org/X86-64_Instruction_Encoding`
[2] `https://en.wikichip.org/wiki/arm/a64`

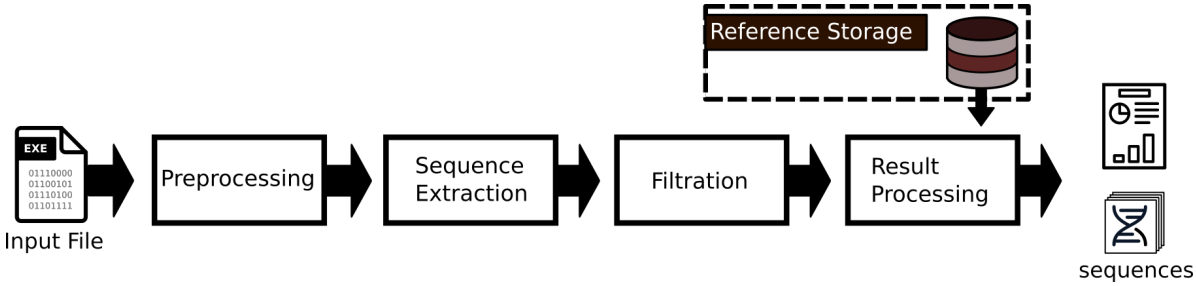


Figure 3.3: Sequence module pipeline.

1. Input file goes through pre-processing. Large chunks of binary data may be discarded during this step, for example, statically linked code included by compilers (Section 3.2.1).
2. Not discarded binary data is processed, and n-grams (sequences) are extracted by moving an extraction window of size 16 and stride of size 1.
3. Extracted sequences are passed to the filtration process that uses different techniques to reduce input set of sequences. Figure 3.4 illustrates filtration in the following steps.
   (a) Sequences with low informational value (entropy) are discarded, see Section 3.2.2.
   (b) Sequences that are stored in the clean database are filtered, see Section 3.2.3.
   (c) User-specified filtration techniques are used to further reduce the input set size, see Section 3.2.4.

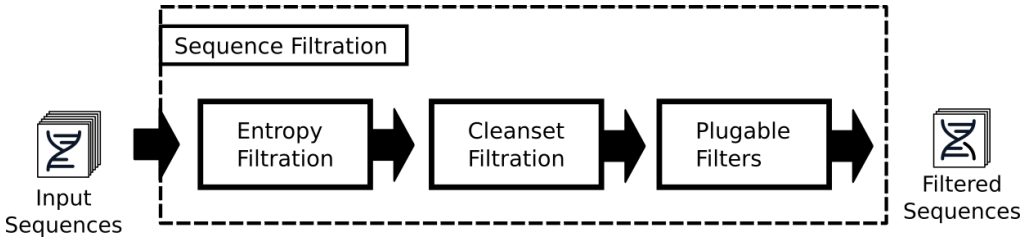


Figure 3.4: Sequence filtration.

4. The reference storage is queried for the presence of non-filtered sequences. Any sequence not present in the query result is discarded.
5. Remaining sequences are formatted into structured data containing information about the family of each sequence extracted from the reference storage.

The module performs comparisons with pre-processed malware sequences stored in reference storage. This means that a sufficient number (and type) of sequences must be stored in the reference storage alongside their relationship with malware families and specific files.

3.1.2 Basic Block Module

The basic block module takes a binary file on input and performs syntactic similarity comparisons on the basic block level. Unlike the sequence module, the basic block module is architecture dependent. The semantics of binary data is extracted by using reverse

engineering frameworks working with specific ISA (Section 3.3). The basic block module is designed to work as a pipeline shown in Figure 3.5.

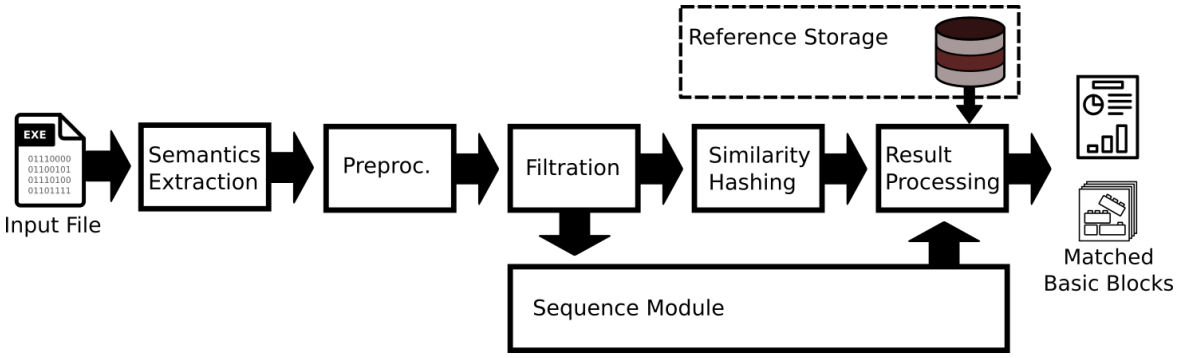


Figure 3.5: Basic block module pipeline.

1. Information about basic blocks is extracted from the input binary. The module uses a general semantic extractor, described in Section 3.3. This step aims to extract all the possible information about basic blocks, like their virtual and physical address and specific bytes.
2. Pre-processing identifies chunks of binary data, for example, statically linked code included by compilers (Section 3.2.1). Each basic block laying inside of an identified chunk of binary data is discarded.
3. Basic blocks are filtered in two steps as shown in Figure 3.6. Firsty, bytes of basic blocks are checked for presence in the cleanset database (see Section 3.2.3). After that, user-specified filtration steps are deployed to further reduce input basic block set (see Section 3.2.4).

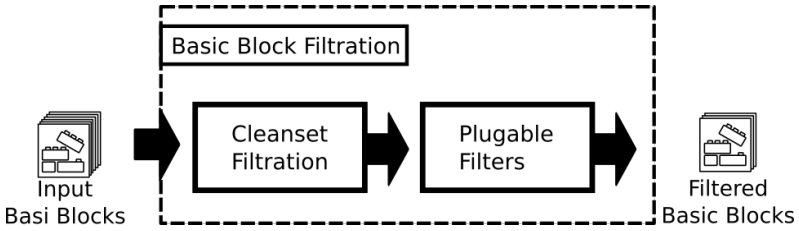


Figure 3.6: Basic blocks filtration.

4. In the similarity hashing step, the basic blocks on input are converted into hashes for similarity matching. To search for similarities and not identities, the normalization technique is used on disassembled binary data as shown in Figure 3.7. Normalization is responsible for discarding or altering basic blocks' instructions; see more details in Section 3.4. The result of the normalization module is a similarity hash for each basic block.

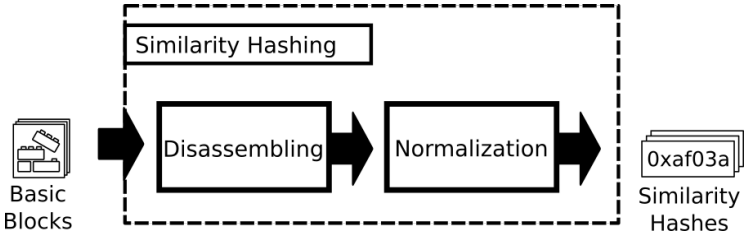


Figure 3.7: Similairty hashing.

5. Similarity hashes are used to query similar basic blocks stored in the reference storage. Original basic blocks, basic blocks queried from the database, and their families form a structured result. For each family, a similarity ratio based on the number of basic blocks matched is included in the result. Apart from that, matches of the sequence module are used to highlight bytes in the context of basic blocks.

3.1.3 Function Module

The module searches for structural similarity on a function level by comparing intra-procedural control flow graphs of functions (CFG), described in Section 2.3. The problem of finding graph isomorphism is NP complete [10]; therefore, the module searches for approximate structural similarities using a suitable representation of CFGs, described in Section 3.5. The module itself operates in steps shown in Figure 3.8.

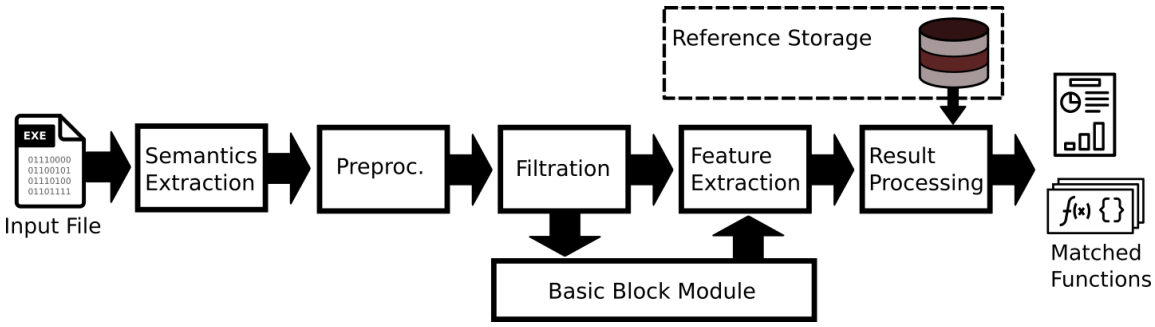


Figure 3.8: Function module pipeline.

1. The module takes an executable file on input and extracts semantic information regarding functions alongside their CFG and bytes using semantic extractor described in Section 3.3.
2. The pre-processing is used to discard specific functions, for example, ones that were detected to be statically linked (Section 3.2.1).
3. Similarly to the basic block module, two-step filtration is used to reduce the extracted set of functions, Figure 3.9.

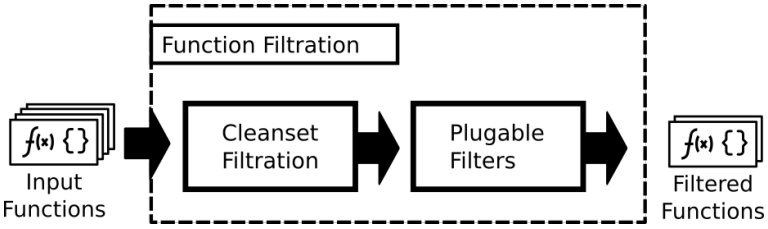


Figure 3.9: Function filtration.

4. The feature extraction step uses the basic block module to extract similarity hashes of basic blocks for each input function. The hashes of basic blocks of a function represent their value in the function's CFG. The CFG is then converted into a feature vector in a process described in Section 3.5.
5. Extracted feature vectors are used to search for those functions that share similar feature vector in the reference storage. The distance of vectors represents their similarity. More details in Section 3.5. For each function on query a function with the highest similarity is returned with information about its family.
6. Original functions, functions queried from the database, and their families form a structured result. Apart from that, the result includes a computed similarity the ratio for each family, comparing the number of hit functions and families' total number of functions.

3.2 Filtration Mechanisms

Filtration mechanisms are designed to reduce memory requirements of reference storage, reduce query size and minimize probability of storing data for a family that is not unique for that family. The service is designed to employ four types of filtration mechanisms:

1. Semantic pre-processing of binary file in specific file formats,
2. entropy filtration to discard random data,
3. filtration by cleanset database designed in this work, and
4. filtration by existing Avast services in form of pluggable filters that can be integrated into the service infrastructure to filter binary data.

3.2.1 Semantic Pre-processing

The purpose of semantic pre-processing step is to remove standard library functions linked to the binary file by a compiler. To search for clean chunks a technology that was created by hex-rays called F.L.I.R.T.[3] is utilized. The technology defines a recognition algorithm that removes specific data from a binary file. The information required by the recognition algorithm is kept in a signature file where each function is represented by a pattern. More information about how flirt works can be found at hex-rays website[4]. To discard a statically linked code a signature file is needed. One option is to include signature files with a service, for example from a publicly available database[5]. The solution chosen was to provide users

[3] `https://www.hex-rays.com/products/ida/tech/flirt/`
[4] `https://www.hex-rays.com/products/ida/tech/flirt/in_depth/`
[5] `https://github.com/Maktm/FLIRTDB`

with option to specify their own signatures to give them more control of what is discarded during this step.

3.2.2 Entropy Filtration

The sequence module utilizes Shannon's entropy to measure the informational value of each sequence to reduce the number of analyzed data. Shannon's entropy quantifies the amount of information in data, thus providing the foundation for a theory around the notion of information. [27]. In the concept of storing and analyzing binary data, Shannon's entropy specifies storage in the number of bits required to represent each value of binary data. The general approach is to look at the binary data as an array of bytes. Then, Shannon's entropy is computed as:

$$E_{Sh} = -\sum_{i=1}^{N} f_i \cdot log_2(f_i)$$

Where N is the number of bytes in input data, and f_i is the frequency of each byte in the input data. The closer the value of E_Sh approaches to its theoretical maximum, the higher the informational value of input data. In the context of 16-bytes sequences ($N = 16$) the maximum theoretical value of E_Sh is when each byte of a sequence is unique; thus, each appears with $f_i = 1/16$:

$$E_{Sh}^{Max} = -16 \cdot \left(\frac{1}{16} \cdot log_2\left(\frac{1}{16}\right)\right) = 4$$

The entropy filtration is wrapped in the module that is shown in Figure 3.10. The module takes a set of sequences and configuration on input. The configuration is presented by user and provides option to specify interval of desired Shannon's entropy. On output, filter returns reduced set of sequences with informational value in selected interval.

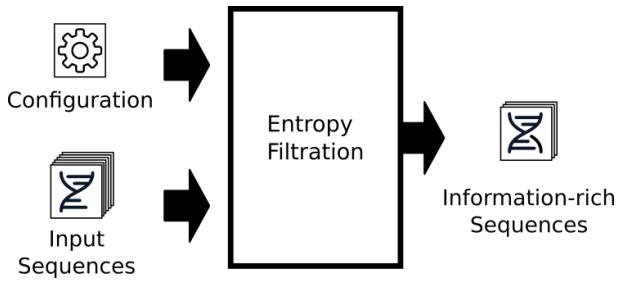


Figure 3.10: Entropy filtration module.

3.2.3 Cleanset Filtration

Similarly to semantics pre-processing, a cleanset filtration is an approach designed to eliminate large portions of input data very fast. Apart from the pre-processing, the approach is not limited to statically linked functions. The filter takes a set of binary data on input as shown in Figure 3.11. The filter is independent of a specific input structure, which can be sequences, basic block, or functions as Figure 3.12 shows. This decision makes it possible to reuse the module throughout designed pipelines in Section 3.1. The bytes on input are hashed, and their hashes are used to query cached data in a cleanset database. If a hash is

present in the database, the filter discards it from the output. The functionality and implementation of the cleanset database are described in Section 4.4. As a result, the module returns a reduced set of input data.

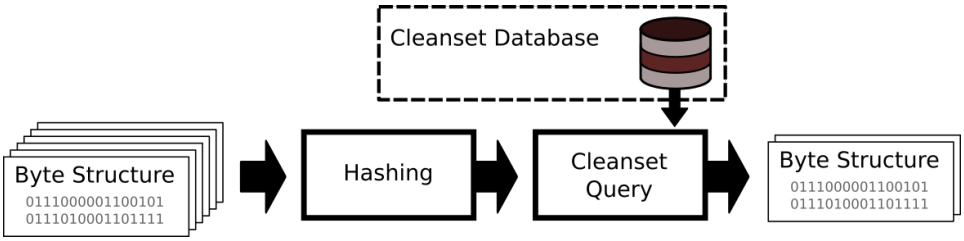


Figure 3.11: Cleanset filtration module.

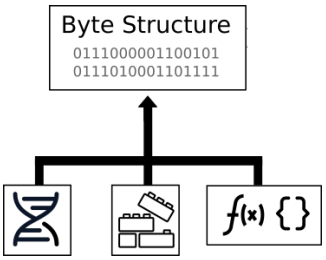


Figure 3.12: Cleanset filration is independent of what bytes represent.

3.2.4 Pluggable Filters

Avast has in its infrastructure services that may be used to query clean files. The current options are, however, limited as discussed in Section 2.5.3. Currently, available solutions tend to work on very high granularity or are noticeably slow. Nonetheless, the YaraZilla service should be capable of communicating with existing or new services easily. The purpose of pluggable filters is to define an interface for plugins to abstract a specific filtration service. Such plugins will be used in a pipeline as shown in Figure 3.13. On input, the pipeline receives a set of binary data. Based on the user-specified configuration, specific filtration plugins will be selected to reduce the input set size. Users may also specify to cache output of these services in the cleanset database designed in Section 4.4.

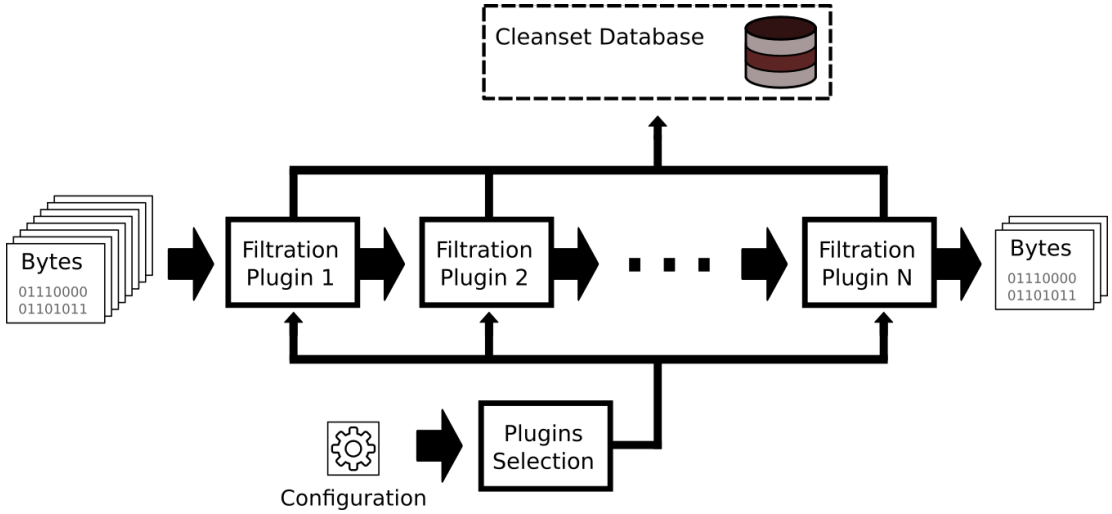


Figure 3.13: Pluggable filtration modules.

3.3 Semantics Extraction

The basic block and function modules require for their functionality the following information from the input binary file:

- Architecture of the input file and wordsize,
- size, physical and virtual addresses of functions,
- size, physical and virtual addresses of a basic blocks,
- intra-procedural control flow graph (CGF) of functions.

This can be achieved by using a third party reverse engineering frameworks like IDA[6], or Rizin[7]. Both IDA and Rizin support large collection of architectures and can extract required information from input binary files. The semantics extraction module is designed to be black-box as shown in Figure 3.3. The module defines interface and structures needed to be filled by specific implementation and returns them on output.

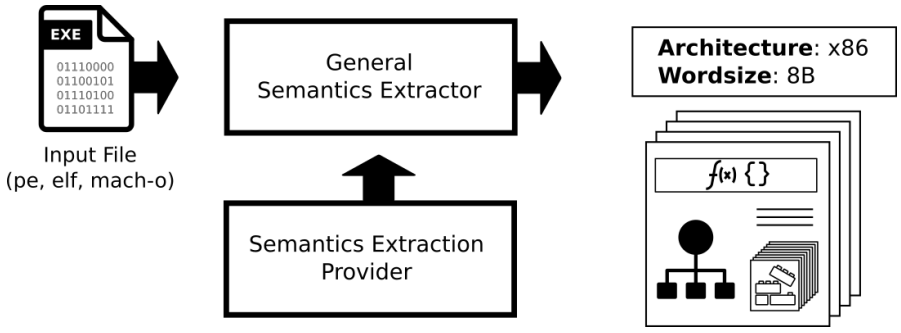


Figure 3.14: Semanitcs extraction module.

[6] https://www.hex-rays.com/
[7] https://rizin.re/

Disassembling

Disassembling is a process of taking binary data and elevating them into the language of symbolic instructions (assembly). The disassembling is generally not time-critical as it is done by mapping object data to text representation. The disassembly is needed in the normalization to recognize operands of instruction so that they can be altered, see Section 3.4.

3.4 Normalization Module

The basic block module utilizes a normalization process to map similar basic blocks into the exact representation. Specifically for the basic block module, the result representation is a hash so that it can be used to query data in the reference storage. The module that implements the normalization process is shown in Section 3.15. It takes a set of disassembled blocks and transforms them into hashes with the procedure described below. However, the normalization is architecture dependent. Each supported architecture must specify normalization steps, like how to replace registers in instructions.

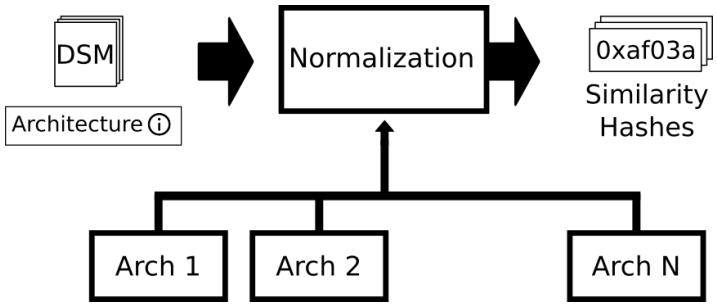


Figure 3.15: Normalization module.

There are different approaches available to normalize basic blocks (described in Section 2.3). The method used in this work is to normalize each instruction of the basic block separately. Then, assemble bytes of a new basic block from these newly created instructions. These new bytes are then hashed to satisfy the requirements of the basic block module. To demonstrate the process of normalization, let us consider the example in Figure 3.16.

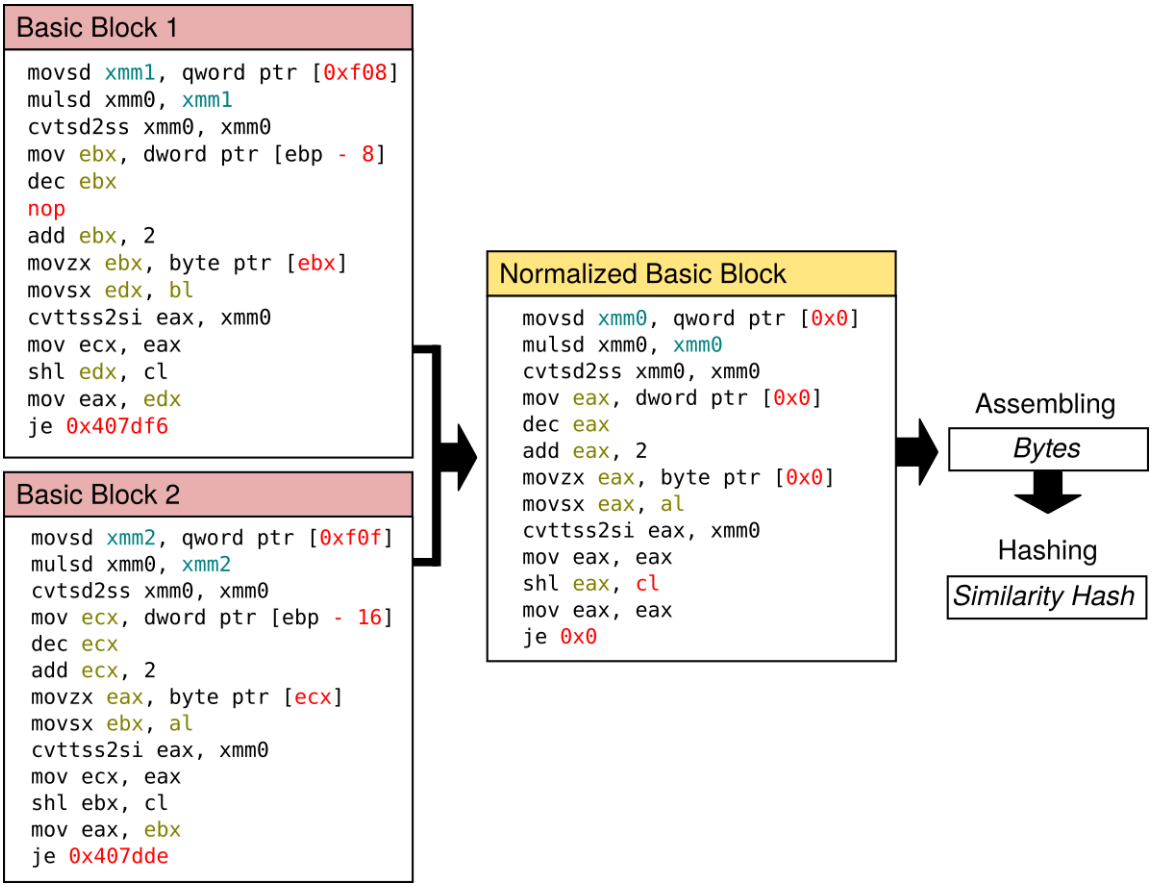


Figure 3.16: Normalization process for x86 architecture.

Basic blocks in Figure 3.16 contain instructions of architecture x86-64. The highlighted parts of basic input blocks are parts where the two basic blocks differ. Both basic blocks in the example are semantically the same. The only difference is in operands they use and one additional `nop` instruction. To remove these differences and create the same similarity hash, both of the basic blocks are normalized in the following steps, specific for the x86 architecture.

1. Each GPR register is mapped to `RAX`, `EAX`, `AX` or `AL` based on the size of the register.
2. Each floating point register is mapped to `ST0`.
3. Each vector register `{X,Y,Z}MM` is mapped to `{X,Y,Z}MM0`.
4. Memory references are replaced by reference to the address `[0x0]` with respect to the loaded/stored size.
5. Immediate values are replaced by value `0x0`.
6. `NOP` instructions are ignored.
7. Instructions that cannot be transformed because of their semantics are either left alone, or must contain special handler method. In the example is instruction `shl` that requires the second operand to be register `CL`).

3.5 Approximation of Structural Similarity

Several approaches can be applied when approximating the structural similarity of control flow graphs. Authors of [9] showed that with a good string representation of CFG, well-known string distancing methods could be employed to search for structure similarity. Authors of [10] took that further and used string representation of CFG to create a feature vector to search for similarities on a scale. This work uses a similar method for scaling structural similarity, employing a technique used in document searching. Specifically, method is using MinHash signatures [8] that represent a form of locality-sensitive hashes.

3.5.1 MinHashes and Jaccard Similarity

By comparing values of MinHashes it can be approximated Jaccard similarity of sets they were generated from. The Jaccard similarity is a metric used commonly to compare sets of given values. Let A and B be sets. Jaccard similarity S_J of A and B is computed as:

$$S_J = \frac{|A \cap B|}{|A \cup B|}$$

3.5.2 Comparable Representation for CFG

Control flow graph G_F of a function F is a pair $G_F = (N, E)$ where N is a set of basic block values, and E is a set of edges between these basic blocks. As we are trying to capture the similarity of control flow graphs, it is essential to ensure that we can compare basic blocks. To compare basic blocks, we assign them a value created by the normalization process described in Section 3.15. To demonstrate process of CFG conversion, let us consider example in Figure 3.17.

By looking at $Function1$ and $Function2$ in Figure 3.17, we can say that these functions are similar. Function 2 added one basic block and switched results of conditional jump. To quantify their similarity we first create suitable string representation of their CFG. The control flow graph of each function in the example is extracted in the following way. For each basic block $n \in N$ of a function F:

1. append "$> n$" to the result CFG string, and
2. for each value $(n, p) \in E$ append "$\#p$" to the result CFG string, so that for each value $(n, r), (n, q) \in E$ is true that if $r < q$ then "$\#r$" is appended before "$\#q$".

The result string can be used with any string distance method to approximate their similarity. With YaraZilla, this work employs a technique that was shown to be scalable. Therefore, an option of how to employ the Jaccard similarity on the control flow graph strings will be further examined in the following section.

3.5.3 Extracting MinHashes from CFG String

The technique used in document matching, but also described in control flow graph matching by authors of [10], is splitting string representation into set of n-grams. Then, the n-grams can be converted to MinHashes as described in [8]. The n-gram can be extracted by sliding a window of pre-defined size on the characters of string, however, the approach used in this work is to extract it from objects represented in the string. In terms of CFG string defined in Section 3.5.2, the objects captured in the string may be of three types.

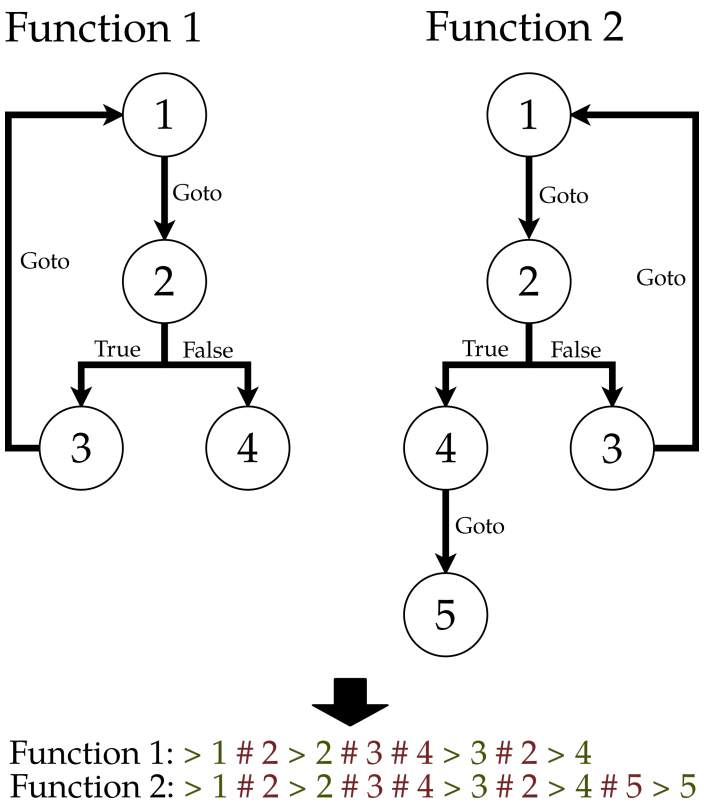


Figure 3.17: Conversion of similar control flow graphs to comparable string representation.

1. Basic block identifier $n \in N$ – a hash of the basic block in the string.
2. Start of basic block character $>$ – a sign that the following basic block is in the CFG and that it starts a connection edge to another basic blocks.
3. Jump on a basic block character $\#$ – a sign that the following basic block terminates connection from a basic block.

By defining size of the n-gram, we decide how much of the call graph structure we want to match. For example.

1. n-grams of size one do not capture structure, only the objects in the call graph.
2. n-grams of size two capture relations of four types.
   (a) Start of a specified basic block ($> n$).
   (b) The basic block jumps on another basic block ($n\#$).
   (c) The basic blocks is followed by start of another basic block ($n >$).
   (d) The basic block is jumped on ($\#n$).
3. n-grams of size more than three capture more context of the structure and therefore are more strict.

3.6 YaraZilla Architecture Design

Section 3.1 provides designs of mechanisms and modules for the YaraZilla file similarity functionality. This section places the functionality into a perspective of a new scalable service architecture. The result architecture is illustrated in Figure 3.18 and functionality of each node of the architecture is broken down in the following text.

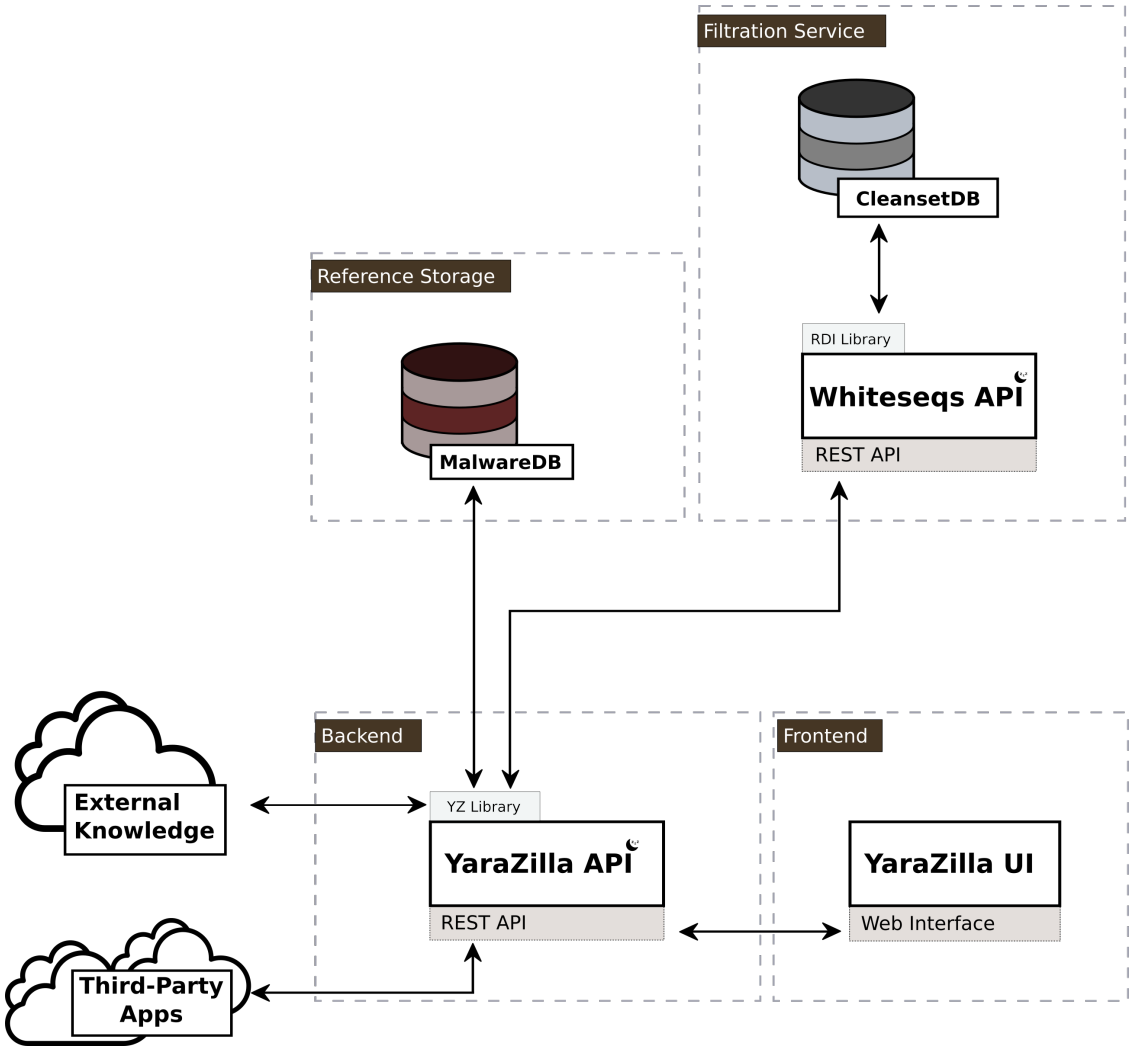


Figure 3.18: YaraZilla Architecture.

3.6.1 YaraZilla Frontend

The frontend of the YaraZilla service provides a user-friendly gateway for accessing file similarity functionality. Users of YaraZilla service can interact with the frontend in form of web application to find similar families of provided sample or extract malware datasets. On its own the frontend does not implement any functionality and uses web interface to access computational resources of YaraZilla API.

3.6.2 YaraZilla backend

The YaraZilla backend consists of two parts. The first part that implements designed file similarity functionality is called the YaraZilla library. The library provides an interface for modules of fast comparison mechanisms designed in Section 3.1, but also modules of extraction mechanisms designed in Section 4.3. Internally, the library uses the Avast internal network to communicate with other services, for example, to filter clean sections of provided samples (see Section 3.2).

The second part of the backend is REST API, wrapping the library functionality and making it available for other tools in Avast infrastructure. For this work, the REST API exposes functionality for shallow YaraZilla frontend; however, the REST API is expected to be integrated into other tools, like automated Yara rule generator.

3.6.3 Filtration Service

The filtration service is a new service for Avast infrastructure that can be used as separate service out of YaraZilla file similarity service. The service consists of the three parts.

1. WhiteseqsDB is a implementation of the database for large storage and quick query of bytes of clean samples. The database is designed in this work, see Section 4.4.
2. RDI Library provides interface for the WhiteseqsDB and exposes it in API calls.
3. REST API wraps RDI Library in web interface to expose the WhiteseqsDB to the internal Avast network.

3.6.4 Malware Database

The malware database module is a cache-based database that provides reference storage that upholds structured data about malware families, files in these families, and additional info used in each YaraZilla service module. The reference storage provides fast queries and is capable of automated backups. This work expects that currently available databases on the market can meet expectations for the designed reference storage.

Chapter 4

Dataset Extraction and Management

The quality of the YaraZilla similarity service designed in Chapter 3 highly depends on the dataset it works with. This work aims to provide a service that can be improved on the go. For that, good dataset extraction and storage options are required. Apart from that, mechanisms for users to modify and enhance existing datasets are also needed. This chapter provides the design of dataset extraction processes in the form of extraction service and database storage needed to uphold extracted datasets.

4.1 Malware Dataset Extraction Service

One particular interaction that malware analysts require from the service is to be able to enhance the existing dataset. For a good service deployable for Malware analysts, it is essential to minimize the complexity of user interactions. For that, the database extraction process is designed to be provided as a service as well. As Figure 4.1 shows, the YaraZilla extraction service will, on input, take a set of binary files. On output, the service will let a selection process decide what information is useful and, therefore, what should be stored in the malware reference storage. The extraction process works on multiple levels of abstractions: sequences, basic blocks, and functions level.

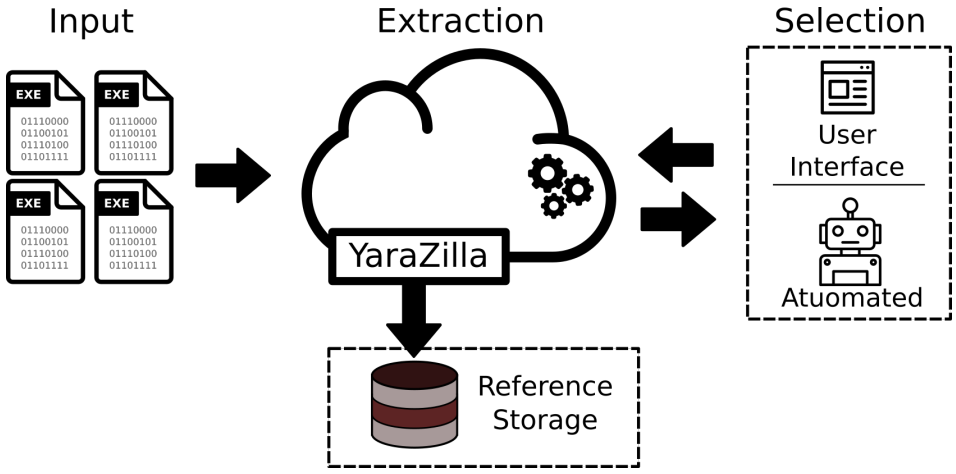


Figure 4.1: YaraZilla extraction service.

4.2 Selection Process

The selection process shown in Figure 4.1 is either a malware analyst or automated script. A malware analysis can interact with extracted data in the user interface. The service, however, must be general enough so that choosing optimal parameters can be later automated by a script or another service.

4.3 Dataset Extraction Modules

For comparison functionality, each module from Section 3.1 must be provided with structured data of malware families. To not limit users of YaraZilla with pre-defined dataset, each module offers a complementary extraction pipeline. Extraction pipelines are modified versions of inspection pipelines that can be used to create a dataset for each inspection module. Processes of extraction and storing dataset as well as ways to engage users are discussed in the following sections.

4.3.1 Sequences Extraction

Sequence extraction process reuses parts of Sequence module (Section 3.3) and is designed as pipeline in Figure 4.2. Its input is a set of binary files and the name of the malware family they are part of. As a result, the process extracts and stores selected sequences into the reference storage.

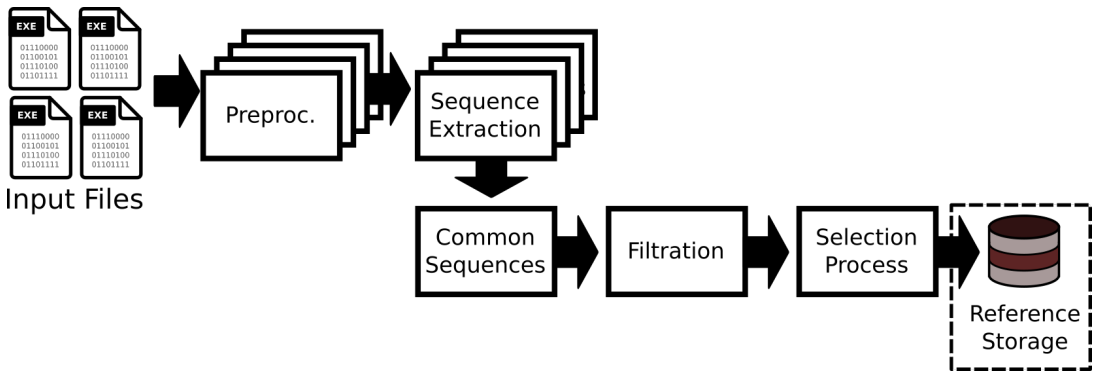


Figure 4.2: Dataset extraction for Sequence Module.

1. Each file on input goes through pre-processing as described in Section 3.2.1. This step may reduce large chunks of binary data containing functions statically linked.
2. Sequence extraction process is run in parallel on each input file and provides a set of sequences (n-grams) for each file.
3. Set of sequences are joined together and for each sequence is tracked what files it was seen in.
4. Selection process chooses sequences based on different parameters. The selection process can be automated or implemented in the user interface. More about selection process in Section 4.2.
5. Selected sequences are passed to the filtration step that is identical to the one described in Section 3.1.1.

6. Sequences that were not filtered are stored into the reference storage with reference to files they appeared in and family they represent.

4.3.2 Basic Blocks Extraction

The basic blocks extraction process reuses parts of designed Basic block module (Section 3.5) and is designed as pipeline in Figure 4.3. The pipeline's input is a set of binary files and the name of the malware family they are part of. As a result, extracts and stores selected basic blocks into the reference storage.

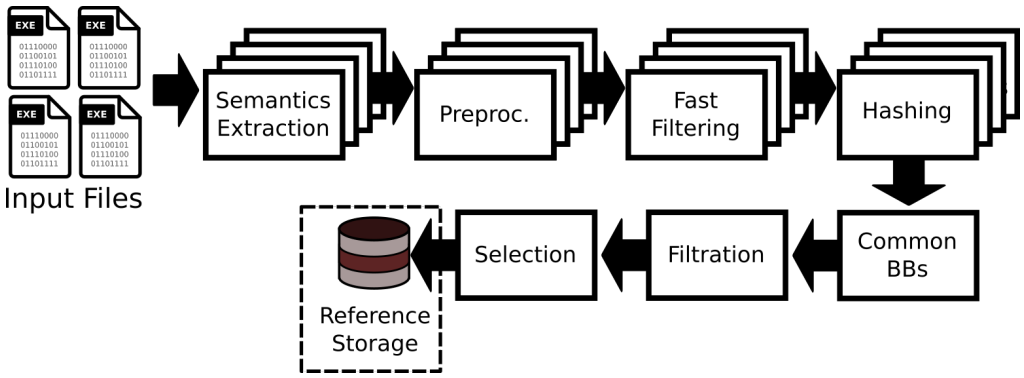


Figure 4.3: Dataset extraction for Basic block module.

1. Semantic extraction step extracts for each file information about their basic blocks. This step utilizes module described in Section 3.3.
2. Each input file goes through pre-processing (see Section 3.2.1) and basic blocks located in discarded chunks of binary data are removed.
3. Altered filtration step is deployed in parallel. This step aims to quickly identify clean basic blocks to lighten the normalization process. This step is not configurable and uses only fast filtering methods, like cleanset filtration 3.2.3.
4. Similarity hashing is used in parallel on input sets of basic blocks. The similarity hashing is described in Section 3.4.
5. Set of normalized basic blocks are joined together and for each basic block is tracked what files it was seen in.
6. Selection process chooses sequences based on different parameters. The selection process can be automated or implemented in the user interface. More about selection process in Section 4.2.
7. Selected basic blocks are passed to the filtration step that is identical to the one described in Section 3.1.2.
8. Basic blocks that were not filtered are stored into the reference storage with the following information:
   - Association to malware family,
   - similarity hash for queries,
   - origin file of the basic block,
   - physical address in the origin file,

- virtual address in the origin file, and
- original bytes.

4.3.3 Functions Extraction

The functions extraction process reuses parts of designed Functions module (Section 3.8) and is designed as pipeline in Figure 4.4. The pipeline's input is a set of binary files and the name of the malware family they are part of. As a result, extracts and stores selected functions into the reference storage.

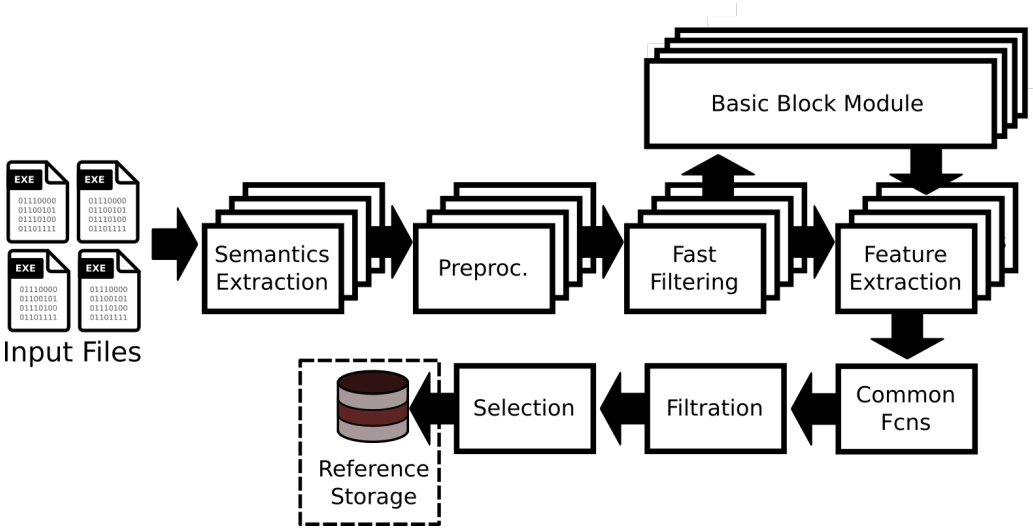


Figure 4.4: Dataset extraction for Function Module.

1. Semantic extraction step extracts for each file information regarding functions alongside their CFG and bytes. This step utilizes module described in Section 3.3.
2. Each input file goes through pre-processing (see Section 3.2.1) and basic functions in discarded chunks of binary data are removed.
3. Altered filtration step is deployed in parallel. This step aims to quickly identify clean functions to lighten the normalization process. This step is not configurable and uses only fast filtering methods, like cleanset filtration (Section 3.2.3).
4. The feature extraction step is run in parallel to extract feature vectors of functions in input sets. Similarly to Fuctcion module pipeline described in Section 3.1.3, the feature extraction step uses Basic block module to extract similarity hashes of functions' basic blocks.
5. Sets of functions are joined together and for each functions is tracked in what files were found similar functions (similarity that satisfies user-defined threshold).
6. Selection process chooses functions based on different parameters. The selection process can be automated or implemented in the user interface. More about selection process in Section 4.2.
7. Selected functions are passed to the filtration step that is identical to the one described in Section 3.1.2.

8. Functions that were not filtered are stored into the reference storage with the following information:
    - Association to malware family,
    - feature vector,
    - control flow of the function,
    - origin file of the function,
    - physical address in the origin file,
    - virtual address in the origin file, and
    - original bytes.

4.4 Database for Cleanset

A cleanset database aims to provide reference storage for already seen bytes of clean files. The designed database must be considered efficient for filtration service to provide stable and usable filtration capabilities. From the perspective of this work only a presence of bytes in the database is interesting, not bytes themself. Storage is considered efficient enough for storing cleanest data if it satisfies the following list of requirements.

- The storage can persistently and reliably hold a vast amount of stored data. The design of cleanset database expects holding at least ten billions different values.
- The storage is capable of quickly providing a large number of values on a query.
- Data are stored compactly to minimize memory resources demand.
- The order of stored data is irrelevant and can be neglected.

After the research, it was concluded that there is no current implementation that can provide all of the qualities required for the cleanset database.

4.4.1 Efficient Storage and Query

One requirements of the cleanset database is to store a large quantity of binary data. The one possible solution that was chosen for the cleanset database is to reduce the storage requirements by using an efficient hash function and therefore not storing the binary data directly. An efficient hash function is capable of uniformly transforming sequences of bytes into hash values of chosen size. For cleanset database it is essential to produce hash values fast enough, not affecting the service functionality. There are two problems of storing large amount of hashes.

1. Hash values are uniformly distributed over output space and are therefore random. No compression algorithm can further reduce the size of random data [24]. For this work, an efficient specialized storage that meets specified requirements was proposed and is further described in Section 4.5.
2. Storing large amount of hashed data increases probability of hash collisions. A very small number of collisions can be accepted if the probability of a byte sequence on query colliding with the stored sequence is small. More detail on collisions in Section 4.4.2

4.4.2 Collisions of Hashes

A hashing is used for fast querying a large set of data commonly in practice. The hashing, however, has limits, and the main one is in the probability of two hashes colliding. A hash collision is a case when two different input values are transformed into the same hash. The probability of this happening varies depending on the type of hashing mechanism chosen. For the storage of cleanset data, we expect a hash function that uniformly distributes hashes across the output space. In this case, the probability of collision depends on the size of generated hashes. An example of hash collision probability based on the number of hashes of a particular size is in Table 4.1 taken from [23]. When choosing a hash size for the cleanset database, we must consider collisions on three levels.

1. The probability of two hashes colliding on the input sample must be negligible. One of the use cases of the cleanset database is to query a large quantity of 16-byte sequences. For example, a one-megabyte input file contains around 16 million sequences. Table 4.1 shows, that for such cases 32-bit hash would not be optimal as the collisions would be prevalent.
2. The probability of a hash collision during the dataset creation is small. The large dataset is expected to contain a large amount of data. The collisions in the dataset should be minimal; however, some are acceptable. The dataset is generated from clean samples, and collisions will only reduce the number of data stored.
3. The probability of hash collision in the data query should be minimal. When we are trying to compute the probability of collision on the data query, we ask about the likelihood of the collision affecting the queried data. To answer the question, we can use the following conditional probability:

$$P(Q|Q \cup N) = \frac{P(Q \cap (Q \cup N))}{P(Q \cup N)} = \frac{P(Q)}{P(Q) + P(N)} \tag{4.1}$$

Where Q represent event of a hash colliding in the input data. The N represents event of hash colliding in the reference data. The $Q \cup N$ represents event of a hash collision happening in either input data or reference data.

32-bit Hashes	**64-bit Hashes**	**160-bit Hashes**	**Collision Probability**
77163	$5.06 \cdot 10^9$	$1.42 \cdot 10^{24}$	1 in 2
30084	$1.97 \cdot 10^9$	$5.55 \cdot 10^{23}$	1 in 10
9292	$609 \cdot 10^6$	$1.71 \cdot 10^{23}$	1 in 100
2932	$192 \cdot 10^6$	$5.41 \cdot 10^{23}$	1 in 10^3
927	$60.7 \cdot 10^6$	$1.71 \cdot 10^{22}$	1 in 10^4
294	$19.2 \cdot 10^6$	$5.41 \cdot 10^{22}$	1 in 10^5
93	$6.07 \cdot 10^6$	$1.71 \cdot 10^{21}$	1 in 10^6
30	$1.92 \cdot 10^6$	$5.41 \cdot 10^{21}$	1 in 10^7
10	607401	$1.71 \cdot 10^{20}$	1 in 10^8

Table 4.1: Probability of a two hashes colliding based on the dataset size.

4.5 Efficient Random Data Storage

This section describes a design of efficient storage capable of storing uniformly distributed numbers. Based on the use case of the YaraZilla filtration service, stored numbers have an 8-byte size. However, the latter created data storage is invariant of specific data size. The designed storage is based on the fact that stored data order is irrelevant, and the maximum number of stored data is known beforehand. Thanks to the data order irrelevancy, we can sort data on input and reduce the search and insertion algorithm's time complexity.

Let a P be a set of integers and let $N = |P|$. Each element $p \in P$ can be represented in a binary representation of B bits size, denoted $\#_2(p) = B$. This means that it takes $N \cdot B/8$ bytes to store the set P sequentially on a disk. For example, it would take 80GB of storage to store a set of ten billion 8-byte integers. To reduce the amount of memory needed, we can sort the integers and remove such parts of their binary representation that can be computed dynamically on demand. The following text proposes a data structure that uses such a principle to effectively store an integer set and allows effective manipulation with the data in terms of time complexity.

4.5.1 Indexable Random Data Structure IRD

Indexable Random Data Structure (IRD) is a proposed data structure to store uniformly distributed data efficiently. To provide calculations of required storage and computations of time complexity, it is required to define IRD structure formally. An IRD structure is illustrated in Figure 4.5 and we define it as a 5-tuple $R = (I, D, B_A, B_D, B_I)$, where

- $I = (I_1, \ldots, I_N)$ is called a index tuple, where $N = 2^{B_A} \wedge \forall i \in \{1, \ldots, N\} : I_i \in \mathbb{N}$
- $D = D_1 \oplus \cdots \oplus D_N$ is called a data tuple, where $N = 2^{B_A}$. The operator $\oplus$ denotes tuple concatenation. $\forall i \in \{1, \ldots, N\} : D_i = (D_1^i, \ldots, D_{I_i}^i)$, meaning the size of D_i is represented by I_i in the index tuple.
- B_A is a bit size of the index tuple address. The number N of elements of the index tuple is always $N = 2^{B_A}$.
- B_D is a bit size of stored data in the data tuple. $\forall i, j : \#_2(D_j^i) = B_D$.
- B_I is a bit size of elements of the index tuple. $\forall i : \#_2(I_i) = B_I$.

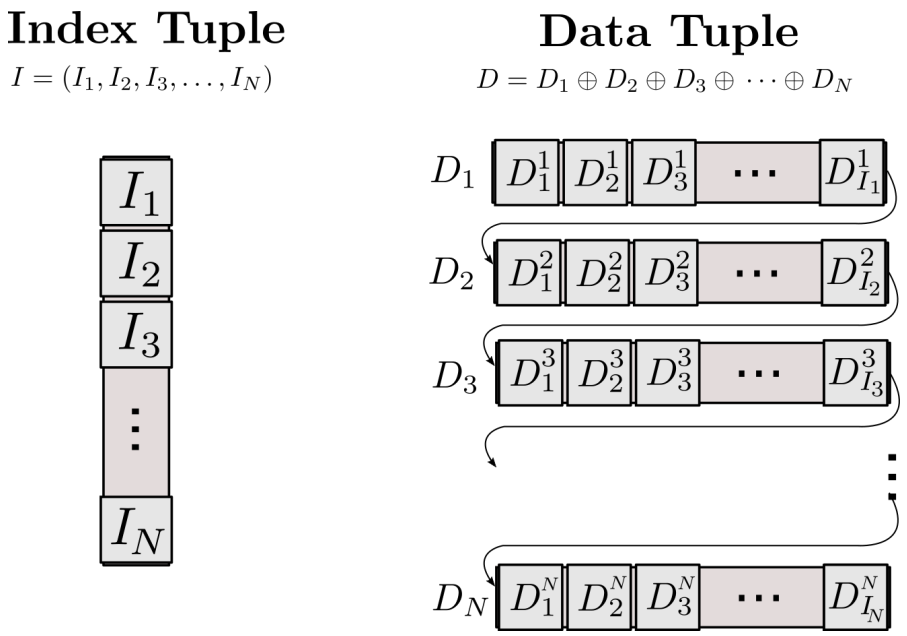


Figure 4.5: IRD Data Structure.

We can intuitively look at IRD as a structure that divides values into the two parts and store them separately. The first part of data is stored in a consecutive chunk of memory labelled the data tuple. The other part is, however, stored in the location of data in the data tuple. The location of data, therefore, is an essential part of the storage. The reduction of storage size comes from the removal of the part stored in location and by grouping data with the same value of the removed part. A Figure 4.6 provides an example of IRD structure $R = ((1, 0, 1, 1), (1, 0, 3), 2, 2, 1)$.

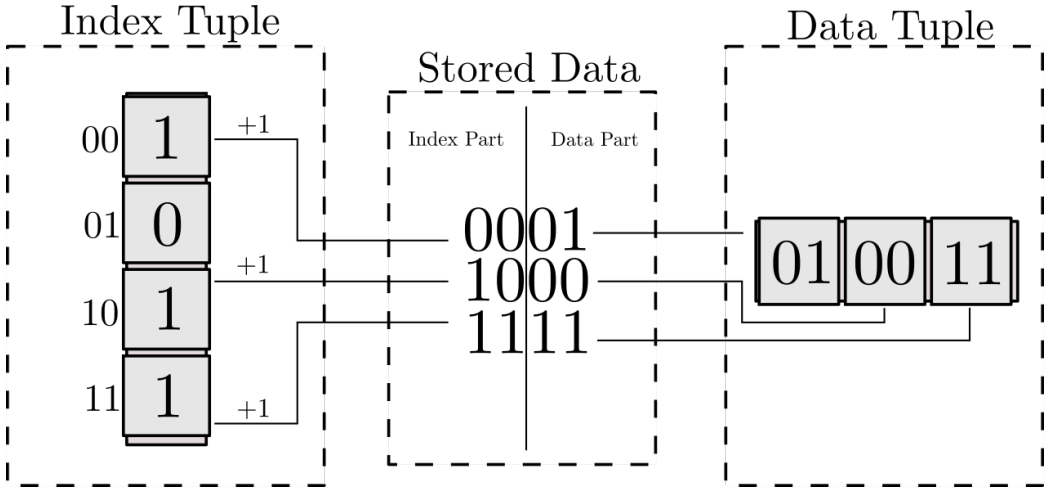


Figure 4.6: Example of storing data into IRD.

In the provided example in Figure 4.6, three values are stored into the IRD data structure: 1, 8, and 15, each represented in a 4-bit representation. Firstly, each stored value is divided into index and data parts. The index part references an index tuple element and increments it by one to indicate the size of stored data. In this example, each stored value

references a different element of the index tuple. Value 1 of the element 00 of the index tuple means that the data tuple stores at the beginning a group of one element with the upper two bits set to zero. We can note that in the example, we are storing three values, each having 4-bits initially. Sequential storage of these values would require 12 bits. By storing input data in the IRD structure, we reduce the size from 12 bits to 10 bits (4 bits in the index tuple and 6 bits in the data tuple). Having a formal definition of IRD structure $R = (I, D, B_A, B_D, B_I)$, we express the size in bits required to store R as:

$$S_R = 2^{B_A} B_i + B_D N \tag{4.2}$$

where N is the number of stored data. For example, to store 10 billion 64-bit integers in a structure $R = (I, D, 32, 32, 8)$, it takes $44GB$ of data storage. On the contrary, it takes 80GB of storage to store these data sequentially. It is possible to remove part of the stored data and safely count sizes of groups in the data structure (with a counter of limited size) because we know the distribution of stored data. Based on the uniform distribution of data we assume, that each group of stored data has roughly the same size, computed as:

$$S^D_{avg} = \frac{N}{2^{B_A}} \tag{4.3}$$

Based on the average size of a group S^D_{avg} and the maximal value of a group counter 2^{B_I}, we compute the maximal possible size of stored data in IRD as:

$$S^D_{avg} \leq 2^{B_I} \tag{4.4}$$

$$\frac{N}{2^{B_A}} \leq 2^{B_I} \tag{4.5}$$

$$N \leq 2^{(B_I + B_A)} \tag{4.6}$$

Based on the relationship 4.6 we compute the minimal size of B_I and B_A as:

$$B_I + B_A \geq log_2(N) \tag{4.7}$$

There are situations where sequential storage of random data would use memory resources more efficiently. Following relations are used to compute the minimum size of input data required for a IRD to be more memory efficient than the sequential storage:

$$S_{seq} \geq S_R \tag{4.8}$$

$$B_0 N \geq 2^{B_A} B_i + (B_0 - B_A) N \tag{4.9}$$

$$B_0 N \geq 2^{B_A} B_i + B_0 N - B_A N \tag{4.10}$$

$$B_A N \geq 2^{B_A} B_I \tag{4.11}$$

$$N \geq \frac{2^{B_A} B_I}{B_A} \tag{4.12}$$

For example, having a IRD $R = (I, D, 32, 32, 8)$ we can compute, that the minimal number of stored data for IRD to be more efficient than a sequential storage is around 1 billion numbers.

4.5.2 Memory Manageable IRD

The deficiency of an IRD is that it stores large chunks of data, either in the index or data tuple. Implementation of such structure in a computer is not feasible as it would require, for example, a static array of 2^{32} elements for an IRD with $B_A = 32$. Apart from that, the last group's indexation would require to sum all elements of the index structure. Such an operation would cost a lot of CPU time. The solution that is proposed for this work is to create another layer of abstraction over IRD as shown in Figure 4.7. Instead of storing continuous index data, we divide the index tuple into chunks and organize them into a hierarchical structure. Algorithm 1 defines insertion of new element in the hierarchical IRD, while Algorithm 2 defines search of an element in the structure.

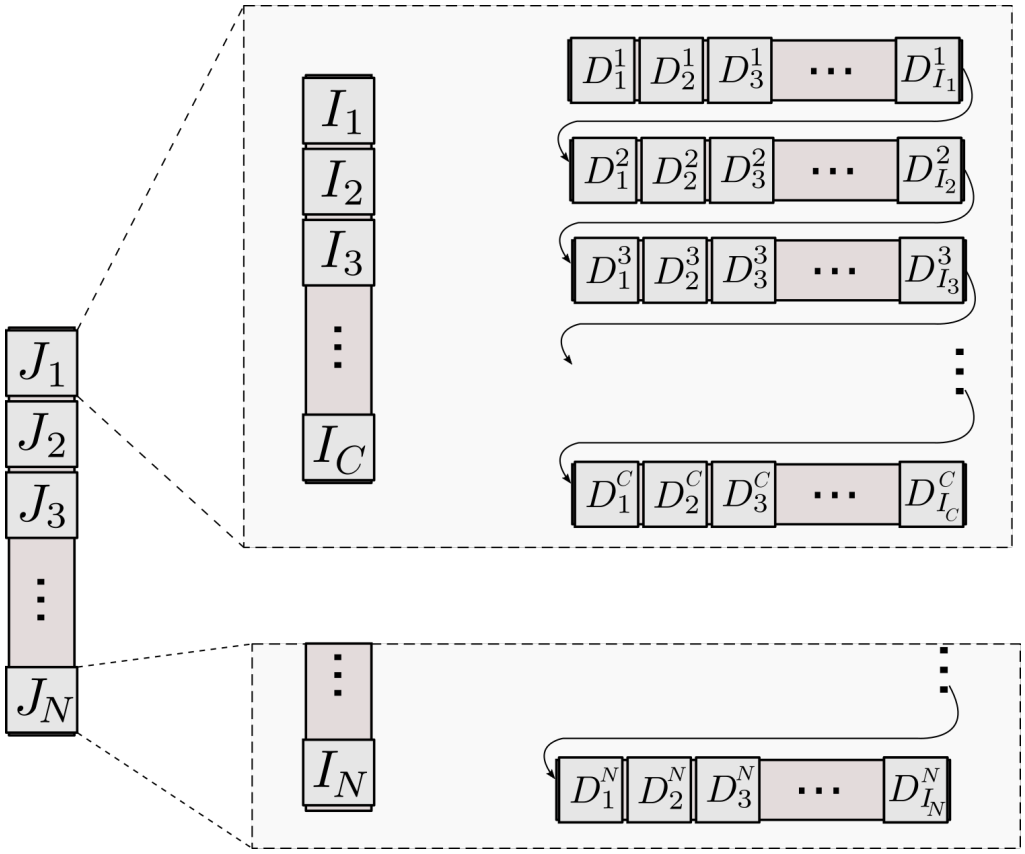


Figure 4.7: A IRD divided into smaller chunks to optimize efficiency.

Algorithm 1: Data Insertion Algorithm.

Input : IRD Structure S, number N to be inserted into J.
Output: $True$ on successful insert. $False$ otherwise.

begin
  Divide N into index and data part: $(N_I, N_D) \leftarrow N$.
  Compute a memory chunk index $c \leftarrow M(N_I, C_S)$.
  Use c to reference a storage structure $J_c = (I, D)$.
  Compute a size sector index $i \leftarrow S(N_I, C_S)$.
  If $I_i + 1 = 2^{B_I}$ return $False$.
  Compute index j to data structure $D : j \leftarrow \sum_{x<i} I_x$.
  Insert N_D on index j and update D:
   $D \leftarrow (D_0, \ldots, D_{j-1}, N_D, D_j, \ldots, D_L), L = \sum_{x \in I} x$.
  Update $I_i \leftarrow I_i + 1$.
  Return $True$.
end

Algorithm 2: Data Search Algorithm.

Input : IRD Structure S, number N to be found into J.
Output: $True$ if found. $False$ otherwise.

begin
  Divide N into index and data part: $(N_I, N_D) \leftarrow N$.
  Compute a memory chunk index $c \leftarrow M(N_I, C_S)$.
  Use c to reference a storage structure $J_c = (I, D)$.
  Compute a size sector index $i \leftarrow S(N_I, C_S)$.
  If $I_i = 0$ return $False$.
  Compute index j to data structure $D : j \leftarrow \sum_{x<i} I_x$.
  $i \leftarrow 0$
  while $i < I_i$ **do**
    If $D_{i_1} = N_D$ return $True$.
    $i \leftarrow i + 1$
  end
  Return $False$.
end

Chapter 5

Scalable Service Implementation

One of the goals of this work is to implement a scalable service. Good traits of a scalable service are that the service is available during high demands but also can be scaled down when need [2]. Apart from that, this work aims to create scalability that can be maintained with growing complexity. To ensure these goals and provide performance, availability, and resilience of designed service, its implementation was inspired by DevOps practices commonly used by growing enterprises as Avast. Automation is the backbone of successful DevOps strategies [2]. Authors of [18] argue that correctly employed DevOps strategies can also help to automate continuous delivery of new software versions while guaranteeing their correctness and reliability.

This chapter is divided into three parts. Section 5.1 describes the implementation of each YaraZilla service using containerization. Section 5.2 introduces automation and how it is employed throughout the development, testing, and deployment steps. The last part of this chapter summarizes implemented service, its usability, and extensibility.

5.1 YaraZilla in Microservices

Research of authors of [18] showed, that highly coupled monolithic architectures are obstacles to effective continuous delivery. As a solution, they suggest that complex dependency management of software components should be imposed to the deployment pipeline. An essential principle for successfully adopting and implementing deployment pipeline is an architecture composed of small and independently deployable units called microservices. Microservices design also provide two architectural attributes that are essential for continuous delivery: testability and deployability.

The design of YaraZilla services discussed in Section 3.6 has taken microservices approach to account and each service can be deployed independently. Each service is deployed using containerization technique provided by Docker[1], see Section 5.2.1. Containerization ensures that build of each service is automated and services can be run in any environemnt.

5.1.1 YaraZilla Backend

The backend service of YaraZilla is implemented as a three level stack shown in Figure 5.1. The figure shows each level of YaraZilla backend together with its dependencies and technologies used. The backend is built and deployed as a container using Docker.

[1] `https://www.docker.com/`

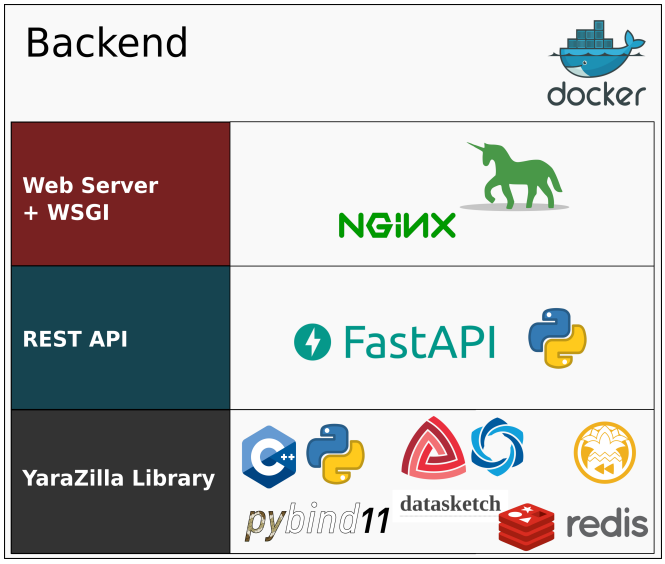


Figure 5.1: Backend technology stack.

1. *YaraZilla Library* implements comparison and extraction mechanisms described in Chapter 3. The critical core of the library is written in C++ to ensure speed of the implementation. This core functions are exposed to Python implementation by using PyBind11[2]. The Python implementation connects and extends core functions and exposes them to REST API. The following major third-party tools are utilized by YaraZilla library.
   (a) Capstone[3] is a lightweight multi-platform, multi-architecture disassembly framework used in Disassembling module, see Section 3.3.
   (b) Keystone[4] is a lightweight multi-platform, multi-architecture assembler framework used in Normalization module, see Section 3.4.
   (c) Rizin[5] is an open source reverse engineering framework utilized during semantics extraction, see Section 3.3.
   (d) Datasketch[6] is a package providing probabilistic data structures that can quickly process and search very large amount of data. The package is used for structural approximation, see Section 3.5.
   (e) Redis Plus Plus[7] is a C++ client library for Redis that is used by MalwareDB service, see Section 5.1.4.
2. *REST API* layer is implemented in Python and exposes YaraZilla Library. The implementation and framework is described in Section 5.1.5.
3. *Web Server and WSGI* layer is responsible for service availability and load balancing. More about this layer in Section 5.1.6.

[2] `https://pybind11.readthedocs.io`
[3] `https://www.capstone-engine.org`
[4] `https://www.keystone-engine.org`
[5] `https://www.rizin.re`
[6] `https://www.ekzhu.com/datasketch`
[7] `https://www.github.com/sewenew/redis-plus-plus`

5.1.2 YaraZilla Frontend

The frontend service of YaraZilla is responsible for user interface and user experience. The service exposes the backend functionality to YaraZilla users in the form of a web application. The service is implemented as a two-layer stack shown in Figure 5.2.

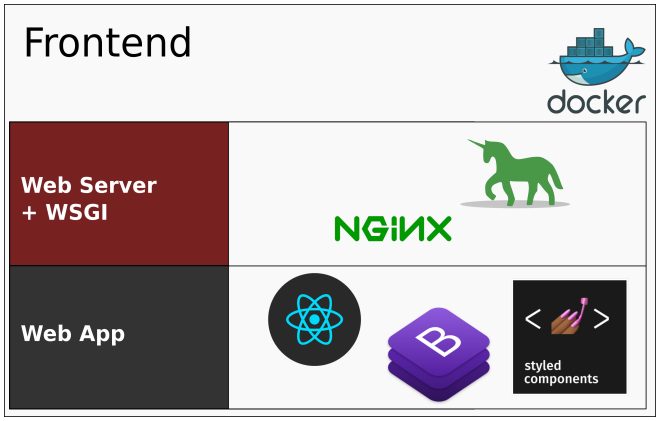


Figure 5.2: Frontend technology stack.

1. *Web App* implements web user interface. The web application is implemented in JavaScript using the following three major frameworks:
   (a) React[8] is a JavaScript library for building user interfaces. React is responsible with managing state of UI components and rendering that state to the `DOM`[9].
   (b) Bootstrap[10] is a popular CSS Framework for developing responsive websites. It is distributed with a large collection of pre-defined UI components that can be used in JavaScript web applications.
   (c) Styled components[11] is a popular JavaScript library for managing CSS styles and writing styled UI components in the JavaScript code.
2. *Web Server and WSGI* layer is responsible for the web service availability and load balancing. More about this layer in Section 5.1.6.

5.1.3 WhiteseqsDB Service

WhiteseqsDB is implemented as a three-layer stack shown in Figure 5.3. The service stack shares a similar structure with the backend service as both services share similar traits – emphasizing speed, reliability, and availability. That is why the core of the WhiteseqsDB is implemented in C++, while manipulation methods and API are implemented in Python.

[8] `https://reactjs.org`
[9] `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model`
[10] `https://getbootstrap.com`
[11] `https://styled-components.com`

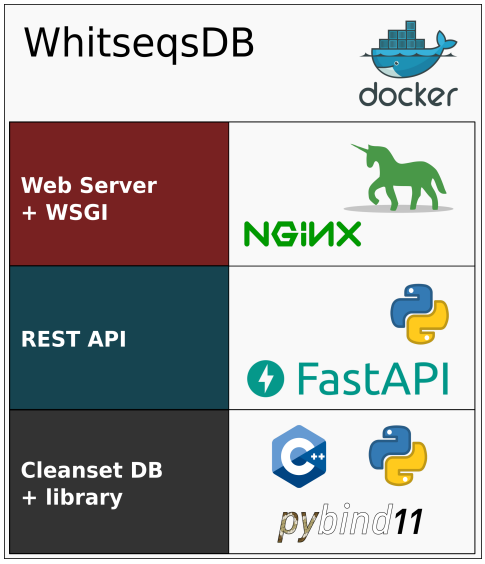


Figure 5.3: WhiteseqsDB technology stack.

1. *WhiteseqsDB library* implements IRD algorithm designed in Section 4.5.1. As speed is crucial, the algorithm is implemented in C++ and Pybind11 is used to expose library functions to Python.
2. *REST API* layer is implemented in Python. The implementation and framework is described in Section 5.1.5.
3. *Web Server and WSGI* layer is responsible for service availability and load balancing. More about this layer in Section 5.1.6.

5.1.4 MalwareDB Service

MalwareDB is a service wrapping Redis[12] database to store data used in YaraZilla backend. Redis is an open-source project that is widely used as an in-memory data structure store, database, cache, and message broker. The advantages of using Redis for storing malware data by YaraZilla are following.

- Redis is an in-memory data storage that can be used as a cache with optional persistence options.
- As data are stored in-memory, Redis is capable of achieving very high throughtput[13].
- Redis provides options to ensure high-availability and monitoring that is desired for ensuring reliability of YaraZilla service.

MalwareDB specifies configuration files of Redis Sentinel[14] to ensure high-availability. Redis Sentinel provides a master-slave architecture shown in Figure 5.4. Each node in the figure contains one sentinel and either master or a replica, running as services on different ports. Each node can be deployed to a server in a different location. The master and replicas can be used to query data; however, writing new data can be done only on the master. Each write operation on the master node propagates changes to replicas. It is

[12] `https://redis.io/`
[13] `https://redis.io/topics/benchmarks`
[14] `https://redis.io/topics/sentinel`

ensured that there is always only one master that is elected by sentinel nodes in case of a failure on the master node. More in-depth description and configuration of Redis, Redis Sentinel and other options like Redis Cluster provide authors of [11].

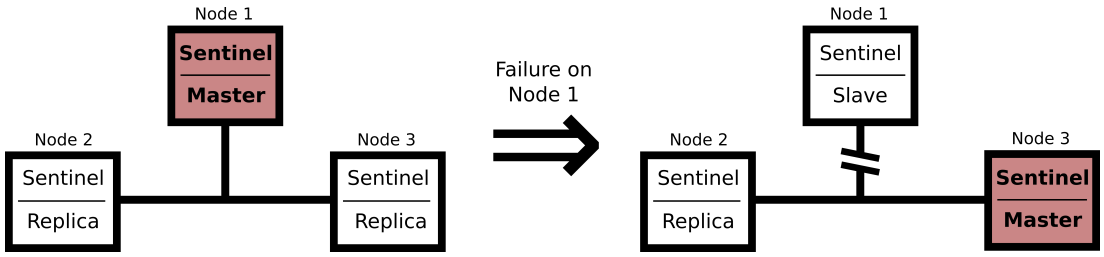


Figure 5.4: Redis Sentinel when failure occures.

5.1.5 REST API

Both YaraZilla backend and WhiteseqsDB services implement core features in C++ and expose them as a library for Python. The reason behind this is that Python provides a vast amount of mature frameworks for building well-documented and testable REST APIs. One such framework is FastAPI[15] that was chosen for YaraZilla services. The FastAPI has many features that are desirable when building scalable and reliable service, some of which are in the following list.

- FastAPI utilizes asynchronous communication with emphasis on speed.
- FastAPI builds on top of data validation framework Pydantic[16] that enforces type hinting system.
- Endpoints are documented in the code. FastAPI automatically generates and hosts up-to-date documentation from the code when deploying service.
- The framework is based on the open standards for APIs: OpenAPI[17] and JSON Schema[18].

5.1.6 Web Server and WSGI

The purpose of web server is to be a gateway for requests from the internet. Its job is to handle requests very quickly and based on the configuration pass only relevant requests to next layer. One such a application is nginx[19] that can handle slow clients, forwarding requests and terminating SSL.

The Web Server Gateway Interface (WSGI) is a specification describing how a web server communicates with web applications. YaraZilla uses Gunicorn[20] WSGI HTTP server that communicates with FastAPI. The purpose of Guniorn is to provide multi-processing and multi-threading when serving multiple clients. Gunicorn translates requests from Nginx into a format which FastAPI understands, and makes sure that its code is executed on demand.

[15] `https://fastapi.tiangolo.com/`
[16] `https://pydantic-docs.helpmanual.io/`
[17] `https://swagger.io/specification/`
[18] `https://json-schema.org/`
[19] `https://nginx.org/`
[20] `https://gunicorn.org/`

5.2 Automated Deployment Pipeline

A deployment pipeline is an automated process that takes source code and makes it readily available to users. The pipeline created for the YaraZilla service is shown in Figure 5.5. The first and the most critical part of the pipeline is the well-documented and versioned source code. The whole pipeline relies on the concept called infrastructure as a code. Each part of the pipeline is configured and versioned in the source code, making it easy to revert or modify any infrastructure changes based on the production feedback. The following sections provide more information about each step of the created pipeline.

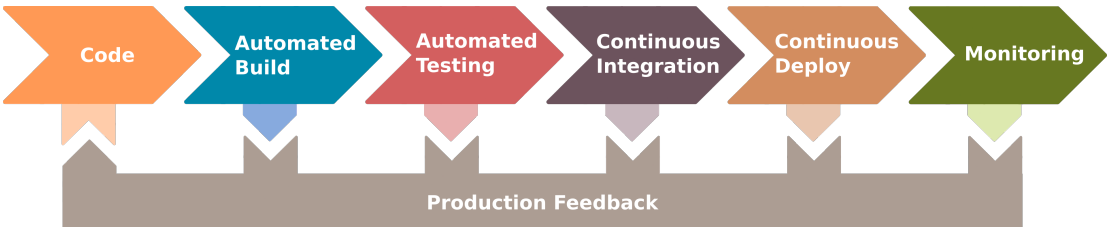


Figure 5.5: Deployment pipeline.

5.2.1 Automating Build

Each microservice defined in Section 5.1 uses Docker to create an image that can be deployed as a Docker container on the desired system. Docker containers resemble virtual machines. The main difference is that Docker containers are more lightweight [6] in terms of memory usage and boot-up time, which results in high portability and scalability.

YaraZilla microservices implement a Dockerfile configuration file that specifies how to build a deployable image. The configuration is versioned in the source code of each service. Apart from the process of building, Dockerfile specifies how to run the service in the deployed Docker container as well.

5.2.2 Automating Testing

Each YaraZilla service provides three sets of test suites: integration, unit and regression testing suite.

1. *Unit tests* ensure verification of designed functionality of each sub-module.
2. *Regression tests* ensure that on same inputs designed pipelines produce the same output.
3. *Integration tests* ensure that services can communicate with each other to fulfil desired tasks and changes to APIs are propagated throughout the infrastructure.

Integration tests verify connectivity and network communication and therefore, each microservice must be online before running test suites. Manually setting up each service is a tedious task and, if done incorrectly, can be time-consuming. Employing an automation process, therefore, minimizes possible mistakes and saves time. The automation process implemented in the YaraZilla structure sets up services with respect to their dependencies as shown in Figure 5.6.

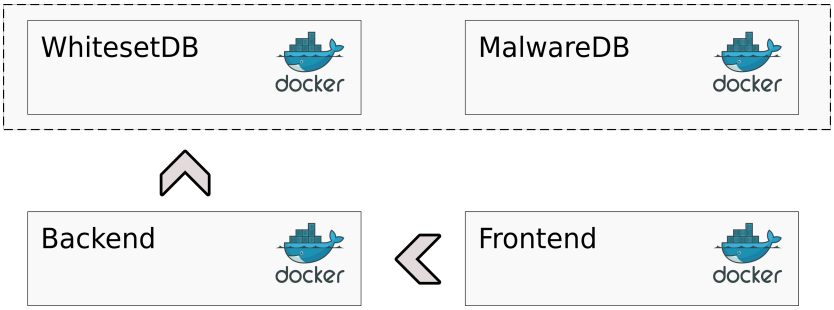


Figure 5.6: Dependency graph of YaraZilla microservices.

The automated testing utilizes Docker Compose[21]. The tool sets up a testing network of microservices by reading a specification in YAML file versioned in the source code repository. Apart from microservices YAML file contains information about testing containers that trigger integration and unit tests. This automation process is wrapped in a shell script to provide single-step triggering for development and continuous integration purposes. Figure 5.7 shows running tests using single script.

```
Tests:  backend-tests frontend-tests whiteseqs-tests
RC   :  0 0 0
 ~/p/yz  bash tests/run_tests.sh -vd                    2.9m
```

Figure 5.7: Automated testing as a single script call.

5.2.3 Continuous Integration

Continuous integration connects automated build and testing to verify new changes to a codebase. Having such a process in the deployment pipeline ensures that any software version committed to the repository is a verified, production-ready candidate. One of the most popular tools for continuous integration is Teamcity[22]. YaraZilla uses Teamcity and integrates it into the Github[23] – a popular source code hosting service. When a new commit is pushed to master, Teamcity runs automated build and testing to ensure its validity. Each checked commit shows the result of testing as seen in Figure 5.8.

[21] https://docs.docker.com/compose/
[22] https://www.jetbrains.com/teamcity/
[23] https://github.com/

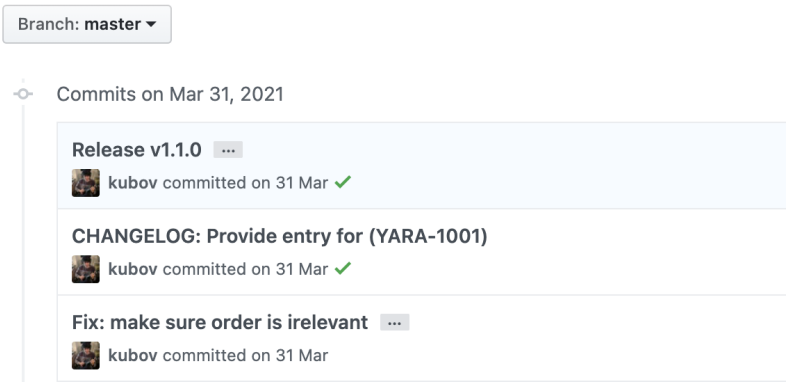


Figure 5.8: Each commit to master should be production ready.

Apart from that, every time a user opens a pull request Teamcity automatically checks if it is possible to merge it to master. Figure 5.9 shows UI of mergeable and UI of a non-mergeable pull request on YaraZilla Github. The Teamcity also provide option to specify configuration as a code written in Kotlin DSL[24]. The option is used in YaraZilla and all of Teamcity configuration is tracked in the YaraZilla repository.

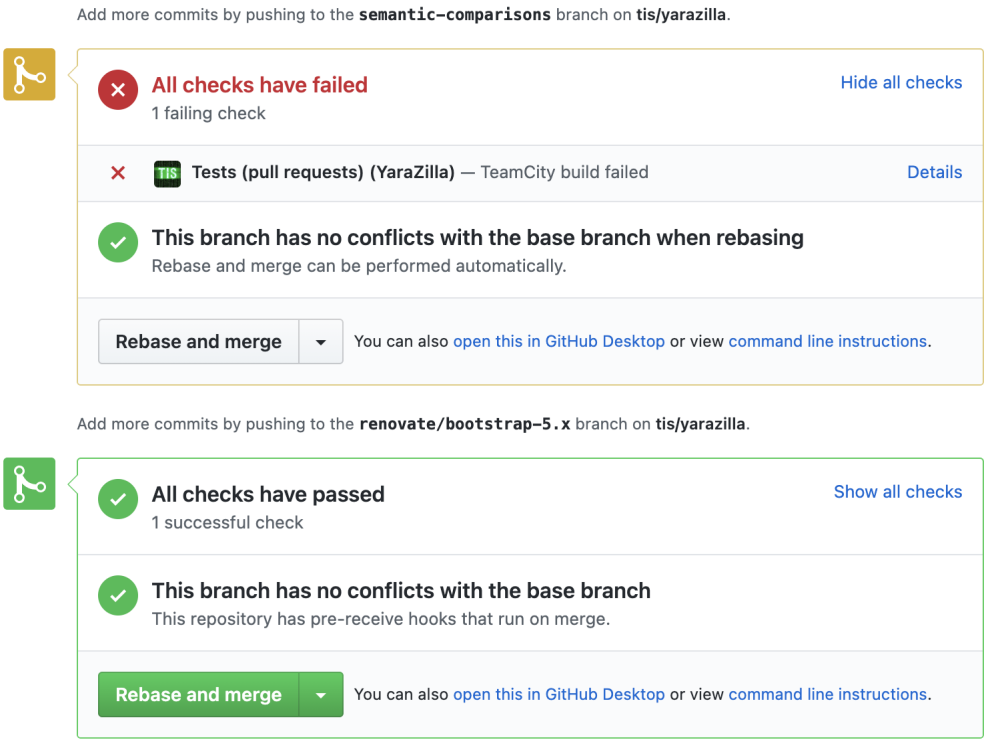


Figure 5.9: Mergeable vs. non-mergeable pull request.

[24] https://www.jetbrains.com/help/teamcity/kotlin-dsl.html

5.2.4 Continuous Deployment

Avast has built a network around Kubernetes[25] for automation of deploying services to a cloud. Kubernetes is an open-source system providing a scalable solution for the deployment of containerized services. This work takes advantage of Avast's Kubernetes network to ensure enough computational resources for the service when by malware analysts. Releases of both YaraZilla backend and frontend are deployed on a cloud using Kubernetes. To deploy YaraZilla services to Kubernetes it is needed to:

1. build Docker image of each service, and
2. push each built image to internal Docker image collection.

After that, each service can be deployed using `kubectl`[26]. The build of Docker images is automated (Section 5.2.1) and other steps can be automated as well. The automation of deployment ensures that after successful continuous integration, any changes committed to the repository can be deployed to production with ease by anyone. For that, the deployment was integrated to Teamcity to provide a one-click release of services to production. Figure 5.10 shows the implementation of the deployment automation in Teamcity.

Figure 5.10: One-click deployment of YaraZilla backend and frontend in Teamcity.

Kubernetes is not, however, reliable to store any data persistently. Therefore both Malware and Cleanset databases need to be deployed on a reliable server. The Malware database uses Redis Sentinel (Section 5.1.4), for which Avast has available dedicated servers, pre-configured and ready to use. The WhiteseqsDB, however, is implemented and deployed in the YaraZilla pipeline directly. Automated deployment minimizes the number of steps needed to deploy the database on the desired server by pushing the built image to internal image collection. After that, the database can be deployed on the selected server using the single command as shown in Figure 5.11.

```
kubov@tis02:~$ docker run --name whitseqsdb docker.ida.avast.com/avast/tis/yarazilla-whi
teseqs-db:1a64c3811b0f4f5a0bac816dcc3a32140d85dcca
INFO:     Started server process [1]
INFO:     Waiting for application startup.
WARNING: Unable to contact SMTP server.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8080 (Press CTRL+C to quit)
```

Figure 5.11: Running WhitseqsDB in a single command.

5.2.5 Monitoring and High Availability

The goal of monitoring is to ensure high service availability, reliability, and security. One of the crucial parts of monitoring is logging. Each YaraZilla service running in the cloud collects events of connections in logs that can be inspected in case of a service error. An example of examining logs of YaraZilla service using `kubectl` is in Figure 5.12.

[25] `https://kubernetes.io/`
[26] `https://kubernetes.io/docs/reference/kubectl/overview/`

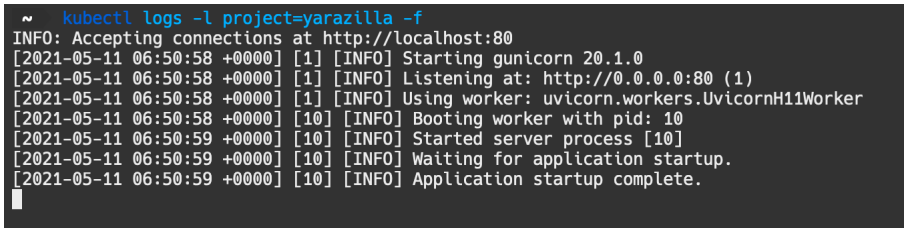


Figure 5.12: Using `kubectl` to inspect logs on the YaraZilla production deploy.

Another important feature is recovery on crash. Services deployed to Kubernetes network are managed by Kubernetes that ensures this feature. MalwareDB uses Redis Sentinel that also handles recovery on crash with a process discussed in Section 5.1.4. WhiteseqsDB is, however, a service implemented in the YaraZilla pipeline and cannot be managed by Kubernetes. The high-availability of the service is managed in three ways:

1. The service configuration in Dockerfile specifies to always restart the service if not active.
2. The service is configured to run automatically on server start-up.
3. The service implements a notification system.

The notification system was employed to ensure that service is always online, and problems are resolved in minimal needed time. When service has problems or crashes, administrators of the server and people specified in the configuration file are notified by e-mail. The example of service shut-down notification is in Section 5.13.

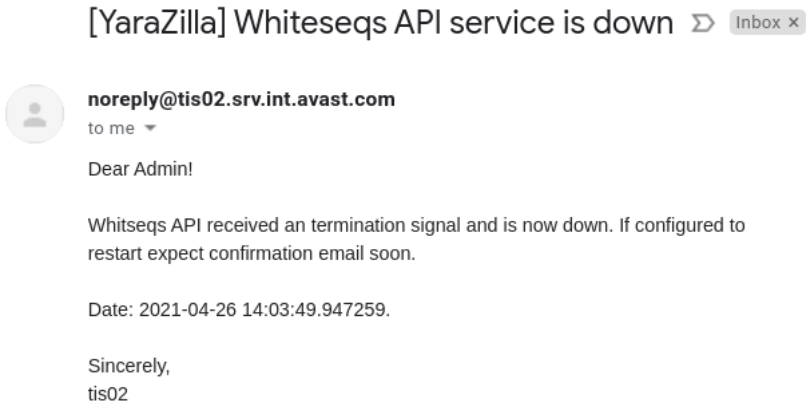


Figure 5.13: Shut-down notification of WhiteseqsDB.

5.3 User Interface

The service is available for analysts to use as a web application on Avast's internal network. The initial screen in Figure 5.14 shows the main layout of the created web application. Two main components of web interface are analysis and overview interface. The overview section shows statistics on stored data. The analysis section is integrates the core of the service and is further divided into two sub-sections. The inspection section of the analysis

lets analysts upload and examine unknown samples. The extraction section provides tools for the extraction of malware datasets.

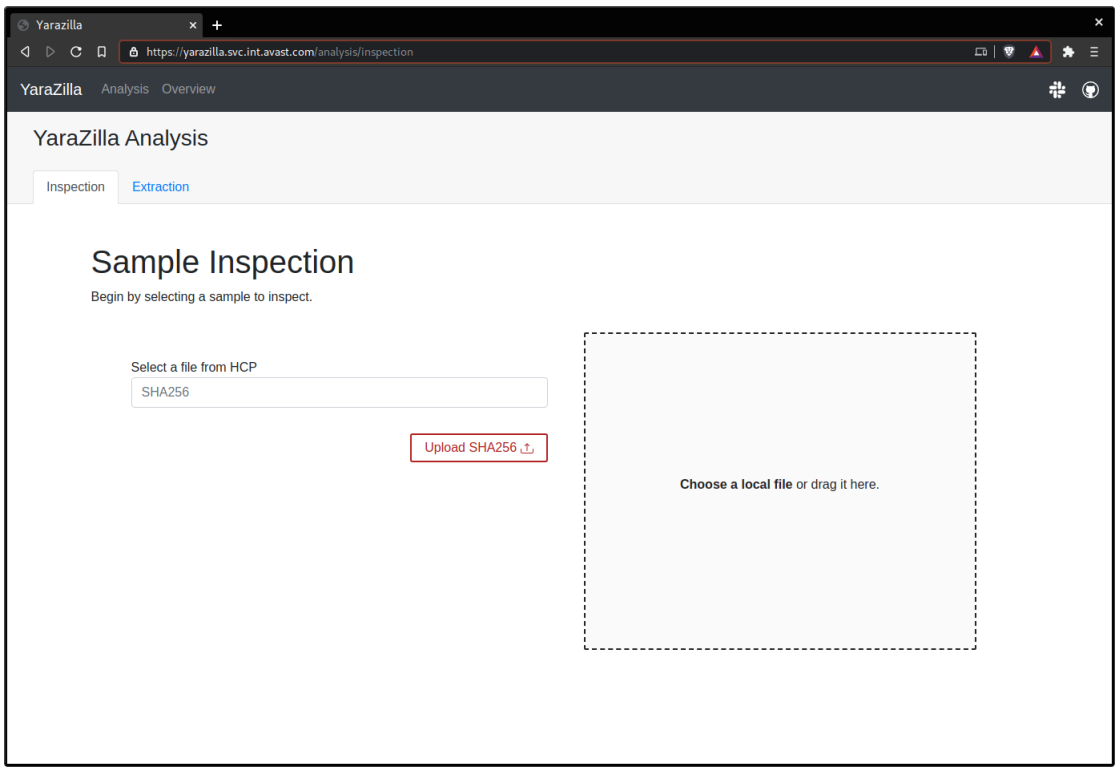


Figure 5.14: Initial screen of the YaraZilla web interface.

Chapter 6

Testing and Evaluation

The YaraZilla service was implemented as specified in Chapter 5. The service was set up with an initial dataset and is currently available for malware researchers to use and improve. Firstly, this chapter describes testing of the new service. However, the central part of this chapter is dedicated to evaluating implemented comparison mechanisms, storage implementation, and the integrated dataset.

6.1 YaraZilla Functionality Testing

Section 5.2.2 mentioned that each created service of YaraZilla was provided with three sets of testing suites: unit, integration, and regression testing suite. As each service is written using various programming languages, different testing frameworks were integrated into the YaraZilla infrastructure. The C++ code in YaraZilla and Whiteseqs libraries is tested using GoogleTest framework[1]. The suite is mainly used for unit testing and verification of each library function on a prepared set of inputs and outputs.

The Python code in YaraZilla API and Whitseqs API is tested using PyTest[2]. The framework is used for each level of testing.

1. The complementary functionality to YaraZilla nad Whiteseqs library in Python code is tested by a set of unit tests with prepared inputs and outputs.
2. The communication of APIs with databases, filtration services, and with each other is tested using test client provided by FastAPI[3].
3. In YaraZilla library, the regression testing uses a set of previously analyzed binary files and verify that extraction and inspection work the same way.

The JavaScript code in YaraZilla frontend is tested using Jest[4] framework. The framework is used for unit testing to ensure that implemented components are rendered in the intended way and show user data properly.

[1] `https://github.com/google/googletest/`
[2] `https://docs.pytest.org/`
[3] `https://fastapi.tiangolo.com/tutorial/testing/`
[4] `https://jestjs.io/`

6.2 Functionality Evaluation

In this section the comparison modules are evaluated to show that the new YaraZilla service is capable of finding similarities in binary files if they are present. Firstly, the service is examined on constructred patch example to show, that service is capable of finding modified basic blocks and functions. Secondly, two subsequent versions of the same software are compared with output of Diaphora[5] to show, that results are comparable with an existing diffing tool. Lastly, the service is provided with binaries of two different programs and result is examined.

In the following examples, the non-malware files are used for evaluation of the service results. Even though the service is designed mainly for malware analysis, inspection of any binary file is possible. The service implements filtration module to remove clean parts of binary, however, these modules can be turned off.

6.2.1 Constructed Patching Example

In the first example, two binary files were created on a Linux system for x64 architecture. The first binary file represents a vulnerable program that calls an insecure function to get user input and has a logic error in the condition evaluation. See the source code in Appendix A.1. The second program was created as a patch for the program, replacing the function call with a secure call and fixing the condition; see Appendix A.2.

Examination Options

Two methods can be employed to find similarities between input files using the YaraZilla service. The first method is to use the extraction part of the service and feed it all of the files. This would extract all sequences, basic blocks, and functions from the files, compare them and return them as a potential dataset for a malware family. This option is quick, but the result is much harder to use to examine patches as all functions would be grouped in the single view and there is no disassembly diffing option included. The second approach chosen for this example is to feed the original file to the YaraZilla service. Then, create a "malware family" from the file, extracting all its sequences, basic blocks and functions and store it to reference storage. The data in the database can be then compared to any binary file by using the YaraZilla inspection service.

Inspection Results

Table 6.1 shows the result of the inspection of the patched file. The YaraZilla service found the patched file similar to the vulnerable file on all three levels of abstractions. The higher the abstraction, the more similar they were found to be.

1. On the sequence level, the changes in source code translate to different binary data in the result binary file. That is why the sequence module found a lot of different sequences in the patched file.
2. In general, the basic block module searches for basic blocks with the same flow of instructions. The lines changed in the patched version resulted in adding or removing instructions in one basic block. The reason for why just one of the basic blocks were

[5] `http://diaphora.re/`

affected is that the changes were performed in one branch. The one changed basic block was not matched on the result.

3. On the function level, all functions have stayed the same just structure of one has been modified. Therefore, all functions were matched on the result.

	Matched	**Total**	**Similarity**
Sequences	438	645	68%
Basic Blocks	21	22	95%
Functions	6	6	100%

Table 6.1: A similarity result returned by YaraZilla.

The function result can be further examined to find a modified basic block in a function and see what instructions have been changed. Figure 6.1 shows that YaraZilla offers an option to filter only similar functions. Using the option filters out the other five matched identical functions. That leaves only one similar function, that is, the one that was patched. The YaraZilla inspection can show comparison of disassembly of the two functions. See the result disassembly comparison in Figure A.3.

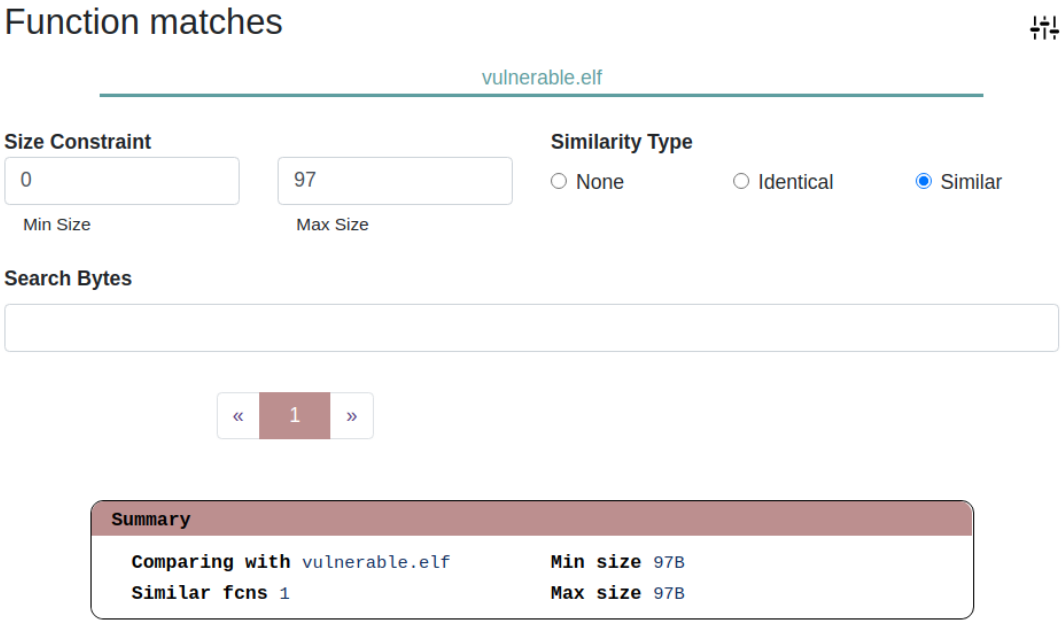


Figure 6.1: Filtration option in the YaraZilla function inspection result.

The result in Figure A.3 shows that in the patched version, there is a call of a new function with a different number of arguments. Apart from that, the change of the condition in the source code resulted in a change of instruction `jne` to instruction `je` in the binary file. The computed similarity (showed in the top-right corner) is 59% meaning that the structure of the functions is 41% different. In the future, we may want to boost the similarity result in such examples as this is a confident result. The possibilities for this are discussed in Section 6.4.

6.2.2 Examination of Subsequent Software Versions

In this example, the YaraZilla function module is compared to a widely used tool for binary file comparison called Diaphora. The purpose of the comparison is to take two subsequent versions of the same software and see if YaraZilla can find matches between these files. Then, compare the quality of matches to Diaphora. For this example, two close versions of Avast antivirus were chosen – Avast version 19.5 and 19.9.

For YaraZilla, the matching was performed the same way as described in Section 6.2.1. The Diaphora requires users to export a function database for both binary files and then compares the databases with a Python script. The result of function matching is in Table 6.2.

	Identified Functions	**Reference Functions**	**Matched**	**Identical**	**Similar**	**Similarity**
YaraZilla	106	103	102	100	2	99%
Diaphora	720	702	672	639	33	96%

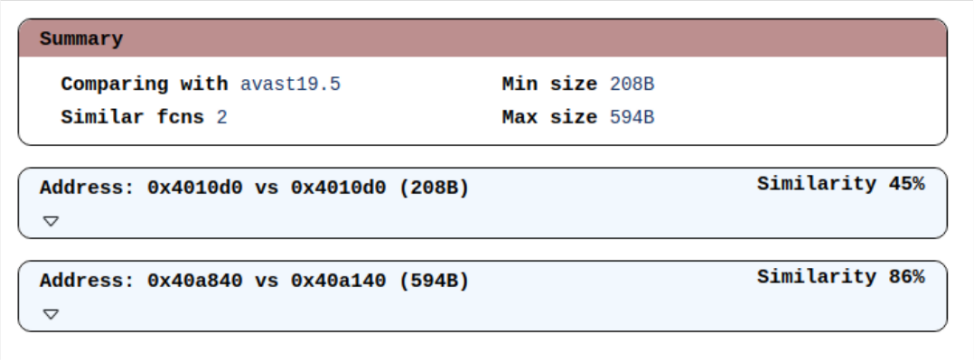
Table 6.2: Comparison of results of function structural matching with Diaphora

The first noticeable thing in Table 6.2 is that the results of Diaphora and YaraZilla were highly divergent in the number of identified functions. After investigation of the output of Diaphora, it was found out that Diaphora includes statically linked functions in the result. Apart from that, Diaphora is built on top of IDA, and the current implementation of YaraZilla works with Rizin. The IDA seems to divide some functions into multiple smaller units, and therefore Diaphora may contain a large dataset even without statically linked functions.

Evaluation of Found Similarities

Firstly, the functions with identical structures identified by YaraZilla were compared to the result of Diaphora. Both Diaphora and YaraZilla did not diverge in results. Each of the YaraZilla identified functions was found in the Diaphora result. Secondly, there were two pairs of functions identified by YaraZilla to have only a similar structure. Each of the functions was found in Diaphora, as Figure 6.2 shows.

YaraZilla



Diaphora

Partial Matches

Line	Address 1	Name 1	Address 2	Name 2	Ratio	BBs 1	BBs 2	Comment
00022	004014a0	sub_4014A0	00401490	sub_401490	0.830	13	13	Same constants
00009	00402070	sub_402070	00401e40	sub_401E40	0.910	1	1	Same constants
00000	004010d0	start	004010d0	start	0.920	11	9	Perfect match,...
00032	00401020	sub_401020	00401020	sub_401020	0.930	7	6	Same address ...

Best Matches

Line	Address 1	Name 1	Address 2	Name 2	Ratio	BBs 1	BBs 2	Comment
00555	00406300	sub_406300	00405bc0	sub_405BC0	1.000	3	3	Same constants
00556	00408820	sub_408820	00408100	sub_408100	1.000	33	33	Same constants
00557	0040a840	sub_40A840	0040a140	sub_40A140	1.000	44	44	Same constants
00558	0040b580	sub_40B580	0040ae80	sub_40AE80	1.000	48	48	Same constants

Not Matched In YaraZilla

Line	Address 1	Name 1	Address 2	Name 2	Ratio	BBs 1	BBs 2	Comment
00010	00402200	sub_402200	00401fd0	sub_401FD0	0.720	63	67	Same constants

Figure 6.2: Similar functions from YaraZilla in Diaphora result.

The first function with 45% similarity showed in Figure 6.2 contains the core of changes in the two binary file versions. The function contains a new condition with an added jump to a new basic block. Apart from that, the function seems to modify function calls. The structural similarity can be considered to be 55% different; however, the confidence of the match should be higher. The possibility to improve confidence is discussed in Section 6.4. For the second match, one basic block was modified in the new version of the function. Other parts of the function stayed similar.

To complete the evaluation, the one function that did not match by Yarazilla was found and compared to the Diaphroa result. Diaphora matched the function with 75% confidence. The match was, however, based on constants present in the function. The function has a highly modified structure, with six basic blocks removed and many others differing. Such

a finding would be possible only by also matching with a different feature vector, discussed in Section 6.4.

6.2.3 Examination of Distinct Binary Files

This section is an addition to the example in Section 6.2.2. In this example, the binary file of Avast 19.5 is compared to a binary file that should be distinct. As a distinct binary file a Notepad++[6] executable was chosen in version 7.9.5. The result of both YaraZilla and Diaphora are in Table 6.3. The Notepad++ binary file was much larger, containing a lot of functions. The file is distributed as a standalone binary and includes a lot of statically linked functions. The YaraZilla removed a lot of statically linked functions. The Diaphora, however, had possibly all of them in the result. From the similarity perspective, the results of YaraZilla and Diaphora are close to each other. The similarity was computed as a ratio of matched and reference functions.

	Identified Functions	**Reference Functions**	**Matched**	**Identical**	**Similar**	**Similarity**
YaraZilla	2880	103	25	23	2	24%
Diaphora	5835	702	151	26	125	21%

Table 6.3: Comparison of matches in distinct binary files.

Firstly, most of the functions with the identical structure matched by YaraZilla were found in the result of Diaphora. However, the functions not found in Diaphora were not even present in the not-matched functions section. The reason is unclear, as these functions were present in IDA for view.

Secondly, similar functions were examined. These are functions that differ in structure in some way. There were two functions matched by YaraZilla to be similar as Figure 6.3 shows. The first with 35% structural similarity was not, however, matched by Diaphora. After examination, it seems that it is indeed a false positive. Even though the structural similarity is low, YaraZilla might implement mechanisms in the future to minimize such cases. For example, adequately chosen similarity cut-off in results can be easily integrated. Another solution is to deploy other strategies for post-matching examination to decrease confidence (see section 6.4).

[6] `https://notepad-plus-plus.org/`

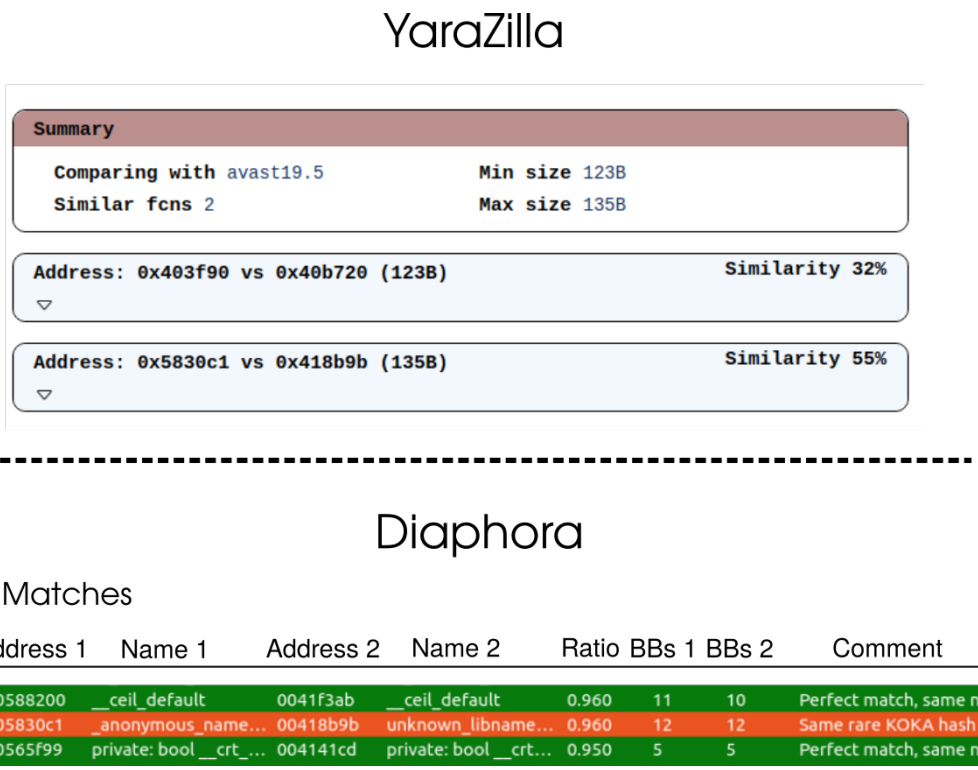


Figure 6.3: Not all similar functions matched in YaraZilla were matched in Diaphora.

6.3 Using YaraZilla on Large Number of Samples

The initially deployed service available for malware analysis at Avast was pre-filled with various malware families from multiple sources. The dataset integrated into the service was processed partly manually and partly by a script. This section examines the initially integrated dataset and describes how the dataset was extracted. In the end, the results of the service with the dataset are evaluated.

The reference storage of the YaraZilla service was filled with samples from two primary sources. The first source of the datasets was provided by Avast and consists of malware families previously examined by malware analysts. The second source was a publicly available malware database hosted at Malpedia[7]. Table 6.4 shows a comparison of the number of families in these two datasets.

	Total Families	**x86 Families**	**1 Sample Families**	**All Files Packed**	**Included Families**	**Unprocessed**
Avast Zoo	1458	1283	362	19	302	600
Malpedia	1906	1454	896	72	434	0

Table 6.4: Comparison of malware datasets.

Families from Malpedia were processed all, while families from Avast were included to complement the ones from Malpedia. The purpose is that in the first step, samples were

[7] `https://malpedia.caad.fkie.fraunhofer.de/`

processed all by hand to filter, clean, and select only relevant data from malware families. More about how the samples were processed in Section 6.3.1. The processed families were, however, saved for later automated processing by a script; see Section 6.3.3.

6.3.1 Dataset Processing

The datasets from Malpedia and Avast were both processed manually, firstly to extract common sequences of malware families. Then, the extracted dataset was used to process families on higher level of abstraction – basic blocks and functions level. To be included in the database, the family was required to satisfy following criteria:

1. The family contains samples of x86 architecture. In the initial version of the YaraZilla service, the preference is to tune the matching for x86 architecture, as most malware is written for the architecture.
2. The samples of the family are unpacked. The packed samples are most of the time packed with a non-standard packer per sample. The packed samples share little to no sequences, and there is no guarantee that the sequences might be meaningful in the result. The retdec-fileinfo[8] tool was used to check if samples are packed. Even though the tool cannot guarantee that a sample is unpacked, it can reliably tell if a sample is packed.
3. The family contains at least two samples. Extraction from only one sample results in a vast amount of sequences most of the time. In the extracted sequences are, however, ones that are not unique for the malware family and lead to an increase in false positives in the result.

During extraction, it was found out that some families need to be manually split into multiple versions. In some cases, families like Hermes in Figure 6.4 contained samples from various dates. However, there were no sequences on the intersection between the dates. In the example of Hermes in Figure 6.4, it was also needed to split samples from a single date into two parts. There were two clusters of samples with a large number of common sequences. However, the two clusters shared only three sequences in between the clusters, as shown in Figure 6.5. On the contrary, this approach may provide more insight into specific version similarity during unknown samples inspection.

Malware Families

hermes

Family Name ↑	Sequences ●	Files ●
hermes.2017-05-01	255	7
hermes.2018-06-27.majors	53332	211
hermes.2018-06-27.minors	48498	27
hermes.2019-01-03	14858	2

Figure 6.4: Example of malware family split into multiple versions.

[8] `https://retdec.com/`

Family	Size
hermes.2018-06-27.majors	3

Figure 6.5: Families sharing sequences with `hermes.2018-06-minors`.

After the sequence extraction and the service's initial deployment to the malware analysts, the service was extended to support basic blocks and functions comparisons. The extracted dataset from sequence processing was used again to extract basic blocks and functions for the same families. As malware families were pre-processed, a script for extraction was created and used for this purpose. The extraction and parameters of the script are described in Section 6.3.3. The evaluation of the extraction is described in Section 6.3.4.

6.3.2 Sequences Extraction

The datasets of sequences of malware families were created in such a way to include just enough sequences that should be unique for the family. This criterium is vague as there is no one solution for all families. For each family, the number of sequences in the result dataset varies. Multiple variables influence the number of sequences for family. For example, the number of samples, their size, and difference in time of creation/identification of the samples. Also, filtration might have left some families with many sequences while reducing the number significantly in others. Even though YaraZilla was designed to minimize the number of clean sequences, it cannot be generally guaranteed. Each sequence was run through a filtration process with a cleanset database and Avast cleanset services during extraction. Results of filtration services were cached for reproduction purposes.

Section 3.1.1 describes that the sequence module uses an entropy selection process. The lower bound of entropy was set in half with the premise that if it yields too many false positives in result, the YaraZilla provides an option to set more strict entropy interval on inspection. Apart from that, more sequences in the database can be removed in post-processing. The evaluation of extraction results is provided in Section 6.3.4.

6.3.3 Script Extraction

An automated script that works with the pre-processed database was created to extract function and basic blocks from input datasets. The script uses YaraZilla REST API to extract data of the specified input samples. After YaraZilla extracts functions and basic blocks from input samples, the script selects individual functions and basic blocks based on input parameters. As a result, the script returns the selected functions and basic blocks to YaraZilla to store them in the reference storage for the specified family. The extraction script has the three following parameters that alter how many basic blocks and functions are extracted from the input samples.

1. Appearance ratio. The appearance ratio means how many files the basic block must have been seen to be counted to result.
2. Maximal allowed size. The parameter tries to reduce the appearance of standard compiler-generated constructs. An example of such constructs is in Appendix B.1.
3. Structure similarity. This parameter applies only to function selection. Each function was extracted from at least one sample. The extraction process of YaraZilla for each

function computes which files contain structurally similar function and then computes average similarity for each function.

The script is on the start provided with a maximal and minimal allowed boundary for each parameter. For each parameter, the script starts on the maximal boundary. Then, it iteratively decreases each element's value until a specified minimum amount of functions and basic blocks is selected. Suppose the conditions provided to the script cannot be satisfied. In that case, the script logs the input family and selects it for manual re-processing.

6.3.4 Examining Relationships

The initially created dataset was examined for interesting cases such as shared sequences, basic blocks, and functions between stored families. The purpose of the examination was to review the parameters used for extraction and gather a broader sense of how to improve the extraction process to include just unique data for each family. From the perspective of sequences, 283 families share sequences with at least one other family. There were obvious cases when the family was extracted from Malpedia and then from Avast Zoo with a different name.

In other cases, the relationship is not that clear. Their similarity might cause the relationships between families, but it also might be caused by shared undetected statically linked functions. As an example, Figure C.1 shows similar malware families to Matrix banker based on the number of shared basic blocks. The example is a particular case when most shared basic blocks are not unique for the family. The shared basic blocks were examined case by case. It was found that it is a single basic block for all families that share one basic block with Matrix banker. The basic block is shown in comparison with the Mespioza family in Figure C.2. The basic block varied only in slight variation of parameters of a few of its instructions. The basic block is part of a larger function that is also shared between the families shown in Figure C.3. The function is an undetected statically linked function that could have been removed with a larger set of pre-processing signatures. Such a function and its basic blocks can be further removed from the dataset in three ways, by:

1. user interaction in the frontend and let user specify uninteresting relationship,
2. updating pre-processing signature files used to remove statically linked functions and preventing the function from returning to the dataset later, or
3. selecting the function to be clean and storing it in the cleanset database for future filtration.

6.4 YaraZilla Limitations and Discussion

The current version of YaraZilla provided a good introduction to a new file similarity service. Even though not all data included in the dataset is unique for each family, the service is designed and works in such way that misleading or non-interesting results can be minimized on the user level. On option provided for users is option to use various parameters when inspecting new unknow samples. Another that was discussed is to improve processing and filtration part of the service for example by extending pre-processing capabilities. Apart from that, the dataset can be improved by extending the dataset of cleanset database also provided by this work.

Apart from similarity service, YaraZilla has been proven to work as a binary diffing tool on a scale too. The function extraction and inspection functionality can be used to find changes in the code on larger scale in one-to-many comparison way. Currently, the matching mechanism of function extraction relies solely on Jaccard similarity, however, this can be extended in another comparison layer. A fast matching will be the core of function module and upon match another layer of comparison will be deployed to boost confidence of matches.

Another possible way to extend the service is further automating dataset extraction. For example, by integrating machine learning mechanisms to automate the process of selection interesting sequences, basic blocks and functions to integrate into the reference storage.

6.5 Cleanset Database

This work provided a new IRD structure for storing and fastly querying a large amount of uniformly distributed data. The design of the structure in Section 4.5 was general, and one of the outputs of this work is implemented, tested, and deployed variant of the IRD. The implementation is currently deployed and available for traffic on Avast's internal network. As of May 2021, the deployed service in Avast's network stores two billion unique hashes representing bytes of clean samples. The dataset was partly generated by processing a large number of binary data and partly by caching results of internal services for querying clean files.

6.5.1 IRD Parameters

Section 4.5 defines the IRD structure as a 5-tuple $R = (I, D, B_A, B_D, B_I)$. The I and D are tuples that represent stored data. B_A, B_D and B_I are parameters constraining the I and D tuples. Based on the hash collision analysis in Section 4.4.2 and convenience of implementation on architecture x64, the following parameters were chosen to store hashes:

1. Address size of the index tuple $B_A = 32b$,
2. size of elements stored in data tuple $B_D = 32b$, and
3. size of elements stored in address tuple $B_I = 8b$.

We can compute the theoretical maximum and minimal number of stored hashes based on analysis of IRD parameters in Section 4.5.1. The computation of effectiveness with comparison to sequential storage is from the following equation:

$$N \geq \frac{2^{B_A} B_i}{B_A}$$
$$N \geq 1.0737 \cdot 10^9$$

Theoretical maximal number of stored data deduced from the type of data distribution is:

$$N \leq 2^{B_I + B_A}$$
$$N \leq 1.0995 \cdot 10^{12}$$

6.5.2 Scaling and Resources

Another parameter that was specified as an addition to formal IRD is a chunk size specified in Section 4.5.2. The chunk size parameter affects the complexity of the structure in the memory and the time complexity of the query. The time required to compute an address of an element in the structure rises with a larger chunk size. On the contrary, the structure is more memory efficient with larger chunk size. For demonstration of this principle, the Figure 6.6 shows how the size of the chunks of the implemented solution affects the size of the structure in memory. The figure shows multiple graphs, each for a different number of stored hashes. The vertical axis summarizes the memory usage of each solution. The *Seq* represents sequential storage, IRD theoretical implementation, and numerical values denote chosen chunk size. The input values were randomly generated for purpose of the example. The memory load was measured with `pmap`[9].

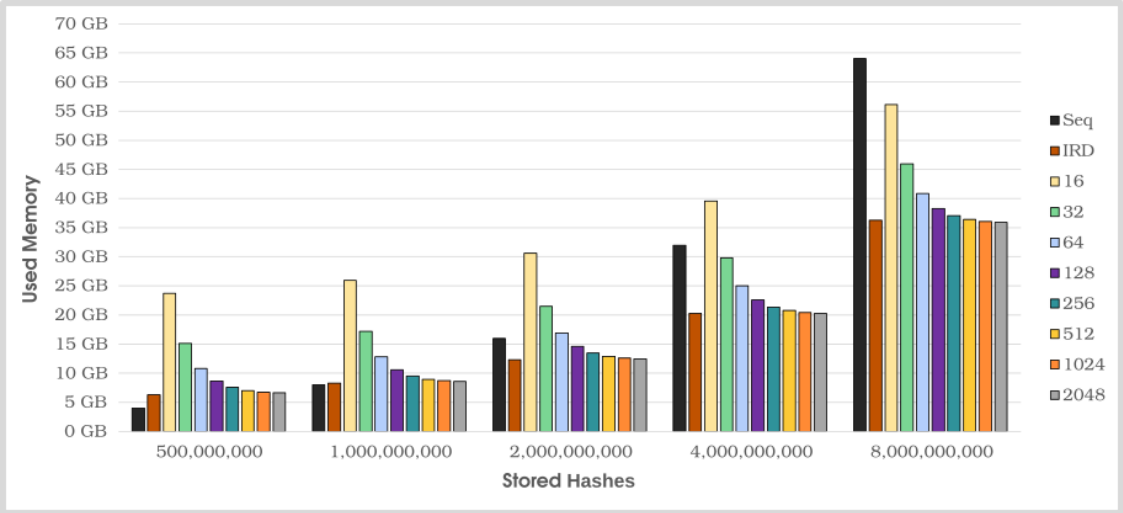


Figure 6.6: Effect of chunk size on memory requirements.

Table 6.5 shows how larger chunk size (CS) affects the time to query data of specified size (Q). The query data were randomly generated and monitored storage was the deployed database with real-world data. As the measurement was performed on the deployed storage, a REST API was used to query the result. The time for communication and data processing is therefore included in the measurement as well. The experiment was conducted on the same machine as the service is deployed to minimize communication delay.

[9] `https://linux.die.net/man/1/pmap`

Q \CS	128	256	512	1024	2048	4096	8192	16384	32768	65536
50000	0.06	0.06	0.07	0.07	0.07	0.10	0.10	0.13	0.22	0.34
100000	0.09	0.10	0.11	0.13	0.14	0.16	0.21	0.28	0.34	0.59
200000	0.18	0.19	0.21	0.25	0.25	0.30	0.45	0.57	0.82	1.37
400000	0.36	0.37	0.44	0.48	0.59	0.59	0.86	1.07	1.53	2.42
800000	0.70	0.73	0.86	0.94	0.98	1.43	1.52	2.28	3.39	5.62
1600000	1.43	1.48	1.71	1.75	2.31	2.36	2.93	4.03	5.61	10.66
3200000	2.93	2.94	3.33	3.76	4.06	4.79	6.33	7.89	11.48	21.25
6400000	5.99	5.95	6.54	7.27	8.26	9.90	12.18	16.07	24.25	43.85

Table 6.5: Effect of the chunk size parameter on the time to execute query in seconds.

6.5.3 Improvements and Future work

Even though the current implementation is fast, it can be improved. Currently, the computation is done only sequentially. In the future, multiple processors can be integrated into the algorithm to compute prefix sums more quickly and speed up the query. The prefix sums can also be pre-computed and stored in the index, providing constant-time complexity on the query. However, this will mean that the time complexity will transfer to insert and remove operations.

The implemented service provided a solid foundation for future improvements. It was shown that the database could hold a large number of hashes and scales. Even though the current implementation can store a large number of data, it might cache a much larger amount of hashes in the future. To do that, IRD was designed with variable parameters that can be altered for the use case. For YaraZilla, the parameters were chosen to fit x64 architecture and provide fast queries even with sequential implementation. The IRD is not limited to be in-memory structure and can be used with loading and storing data on disk.

Chapter 7

Conclusion

This work studied the binary file similarity area and its usage to deal with malware in practice. A review of commonly used Avast tools was provided, and a use case for a new file similarity service was formed. The established file similarity terminology was used to devise mechanisms for comparing binary files on multiple abstraction levels, focusing on a malware analysis use case. Then, the outlined mechanisms provided a foundation for designing a binary file similarity service. The service operation was divided into inspection and extraction sub-services.

From the inspection perspective, well-known mechanisms for comparison were used. The mechanisms were modified to provide usability on large-scale querying. The inspection was designed to perform identity searches of binary data at the lowest level of abstraction. The normalization process was analyzed and integrated to extract similarity hashes usable for large queries on the instruction and basic-block level. On the function level, the service was designed to approximate structural similarity on a scale, using a well-known mechanism from document similarity searching.

The extraction part of the service was designed as a complement to the inspection. The design devised mechanisms that are extensible and parametrizable. Filtration mechanisms were formed to extract only the essence of binary files needed for the reference dataset. One particular mechanism created by this work is a new method of filtration of clean binary data. The new mechanism is utilizing hashing and a specialized database for storing a vast amount of hashed data. For the new database, a new data structure for efficiently storing uniformly distributed values was proposed to deal with a random nature of hashes – Indexable Random Data Structure (IRD). A formal model of IRD was created and used to prove the memory limitations of this structure. Then, the IRD structure was altered for computer usage, and data manipulation algorithms were constructed.

During implementation, multiple issues were addressed to automate the process of build, testing, and deployment to create a service that can be scaled on increasing demand. After the implementation, the service was put under testing on various levels. On the first level, the service usability was validated by showing it can be used for examination similar binary files. The result was compared to Diaphora. It was shown that the service is capable of comparable results. Then, the service was filled with vast datasets from Avast and Malpedia, and results were evaluated. Even though not all data were showed to be unique in the dataset for each family, the service was shown to minimize unwanted results by a user-level parametrization.

Both the file similarity and filtration services were deployed and are available on the internal company network at Avast. Designed file mechanisms devised in this work are

available for malware researchers, and their demands are iteratively improving the file similarity service.

Bibliography

[1] *BinaryAI: The Neural Search Engine for Binaries* [online]. Tencent Security KEEN Lab, july 2020 [cit. 2021-01-25]. Available at: https://binaryai.readthedocs.io/en/latest/index.html.

[2] *How Can DevOps Make IT Infrastructure More Scalable? 5 Tips to Consider* [online]. United Perfectum, august 2020 [cit. 2021-01-25]. Available at: https://unitedperfectum.com/news/how-can-devops-make-it-infrastructure-more-scalable-5-tips-to-consider/.

[3] *The power of threat attribution* [online]. AO Kaspersky Lab, july 2020 [cit. 2021-01-25]. Available at: https://media.kaspersky.com/en/business-security/enterprise/threat-attribution-engine-whitepaper.pdf.

[4] Apel, M., Bockermann, C. and Meier, M. Measuring similarity of malware behavior. In:. October 2009, p. 891–898. DOI: 10.1109/LCN.2009.5355037.

[5] Arntz, P. *Explained: YARA rules* [online]. March 2016 [cit. 2021-01-25]. Available at: https://blog.malwarebytes.com/security-world/technology/2017/09/explained-yara-rules.

[6] Avi. *Docker vs Virtual Machine – Understanding the Differences* [online]. [cit. 2021-04-18]. Available at: https://geekflare.com/docker-vs-virtual-machine/.

[7] Baker, B., Manber, U. and Muth, R. Compressing Differences of Executable Code. november 2000.

[8] Broder, A. Identifying and Filtering Near-Duplicate Documents. In:. June 2000, p. 1–10. DOI: 10.1007/3-540-45123-4_1. ISBN 978-3-540-67633-1.

[9] Cesare, S. and Xiang, Y. Classification of Malware Using Structured Control Flow. In: *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing - Volume 107*. AUS: Australian Computer Society, Inc., 2010, p. 61–70. AusPDC '10. ISBN 9781920682880.

[10] Cesare, S., Xiang, Y. and Zhou, W. Control Flow-Based Malware VariantDetection. *IEEE Transactions on Dependable and Secure Computing*. 2014, vol. 11, no. 4, p. 307–317. DOI: 10.1109/TDSC.2013.40.

[11] Dayvson Da Silva, M. and Tavares, H. L. *Redis Essentials: Harness the power of Redis to integrate and manage your projects efficiently*. 3rd ed. Packt Publishing, 2015. ISBN 1784392456.

[12] Dimitrova, M. *97% of Malware Infections Are Polymorphic, Researchers Says* [online]. [cit. 2021-01-25]. Available at: https://sensorstechforum.com/97-of-malware-infections-are-polymorphic-researchers-say/.

[13] Díaz, V. *Why is similarity so relevant when investigating attacks* [online]. [cit. 2021-01-25]. Available at: https://blog.virustotal.com/2020/11/why-is-similarity-so-relevant-when.html.

[14] Ganesh, V. and Dill, D. L. A Decision Procedure for Bit-Vectors and Arrays. In: Damm, W. and Hermanns, H., ed. *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, p. 519–531. ISBN 978-3-540-73368-3.

[15] Haq, I. U. and Caballero, J. A Survey of Binary Code Similarity. *CoRR*. 2019, abs/1909.11424. Available at: http://arxiv.org/abs/1909.11424.

[16] Hu, X., Bhatkar, S., Griffin, K. and Shin, K. G. MutantX-S: Scalable Malware Clustering Based on Static Features. In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USA: USENIX Association, 2013, p. 187–198. USENIX ATC'13.

[17] Lakhotia, A., Preda, M. D. and Giacobazzi, R. Fast Location of Similar Code Fragments Using Semantic 'Juice'. In: *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. New York, NY, USA: Association for Computing Machinery, 2013. PPREW '13. DOI: 10.1145/2430553.2430558. ISBN 9781450318570. Available at: https://doi.org/10.1145/2430553.2430558.

[18] Leite, L., Rocha, C., Kon, F., Milojicic, D. and Meirelles, P. A Survey of DevOps Concepts and Challenges. *ACM Computing Surveys*. Association for Computing Machinery (ACM). Jan 2020, vol. 52, no. 6, p. 1–35. DOI: 10.1145/3359981. ISSN 1557-7341. Available at: http://dx.doi.org/10.1145/3359981.

[19] Lindorfer, M., Federico, A., Maggi, F., Comparetti, P. and Zanero, S. Lines of malicious code: Insights into the malicious software industry. *ACM International Conference Proceeding Series*. december 2012. DOI: 10.1145/2420950.2421001.

[20] Luo, L., Ming, J., Wu, D., Liu, P. and Zhu, S. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection. *IEEE Transactions on Software Engineering*. 2017, vol. 43, no. 12, p. 1157–1177. DOI: 10.1109/TSE.2017.2655046.

[21] Pewny, J., Garmany, B., Gawlik, R., Rossow, C. and Holz, T. Cross-Architecture Bug Search in Binary Executables. In: *2015 IEEE Symposium on Security and Privacy*. 2015, p. 709–724. DOI: 10.1109/SP.2015.49.

[22] Pewny, J., Schuster, F., Bernhard, L., Holz, T. and Rossow, C. Leveraging Semantic Signatures for Bug Search in Binary Programs. In: *Proceedings of the 30th Annual Computer Security Applications Conference*. New York, NY, USA: Association for Computing Machinery, 2014, p. 406–415. ACSAC '14. DOI: 10.1145/2664243.2664269. ISBN 9781450330053. Available at: https://doi.org/10.1145/2664243.2664269.

[23] Preshing, J. *Hash Collision Probabilities* [online]. [cit. 2021-04-18]. Available at: https://preshing.com/20110504/hash-collision-probabilities/.

[24] Salomon, D. *Data Compression: The Complete Reference*. 3rd ed. Springer, 2004. ISBN 9780387406978.

[25] Santos, I., Brezo, F., Nieves, J., Penya, Y. K., Sanz, B. et al. Idea: Opcode-Sequence-Based Malware Detection. In: Massacci, F., Wallach, D. and Zannone, N., ed. *Engineering Secure Software and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, p. 35–43. ISBN 978-3-642-11747-3.

[26] Tevet, I. *Introducing Cybersecurity DNA: the Intezer Company Blog* [online]. [cit. 2021-01-25]. Available at: https://www.intezer.com/blog/gdpr/intro-intezer-blog/.

[27] Vajapeyam, S. *Understanding Shannon's Entropy metric for Information* [online]. AO Kaspersky Lab, march 2014 [cit. 2021-01-25]. Available at: https://arxiv.org/pdf/1405.2061.pdf.

[28] Wang, Z., Pierce, K. and Mcfarling, S. BMAT – A Binary Matching Tool for Stale Profile Propagation. june 2000, vol. 2.

[29] Wroblewski, G. *General Method of Program Code Obfuscation*. 2002.

Appendix A

Constructed Patches Example

```c
#include <stdio.h>
#include <string.h>

int main(void) {
    char buff[14];
    printf("Enter password: ");
    gets(buff);

    if (!strcmp(buff, "popocatepetl"))
        printf("Try again!\n");
    else
        printf("You're in.\n");

    return 0;
}
```

Figure A.1: Original vulnerable code.

```c
#include <stdio.h>
#include <string.h>

int main(void) {
    char buff[14];
    printf("Enter password: ");
    // Fixed: gets is not secure!
    fgets(buff, 14, stdin);

    // Fixed: Error in condition!
    if (strcmp(buff, "popocatepetl"))
        printf("Try again!\n");
    else
        printf("You're in.\n");

    return 0;
}
```

Figure A.2: Patched version of the vulnerable code.

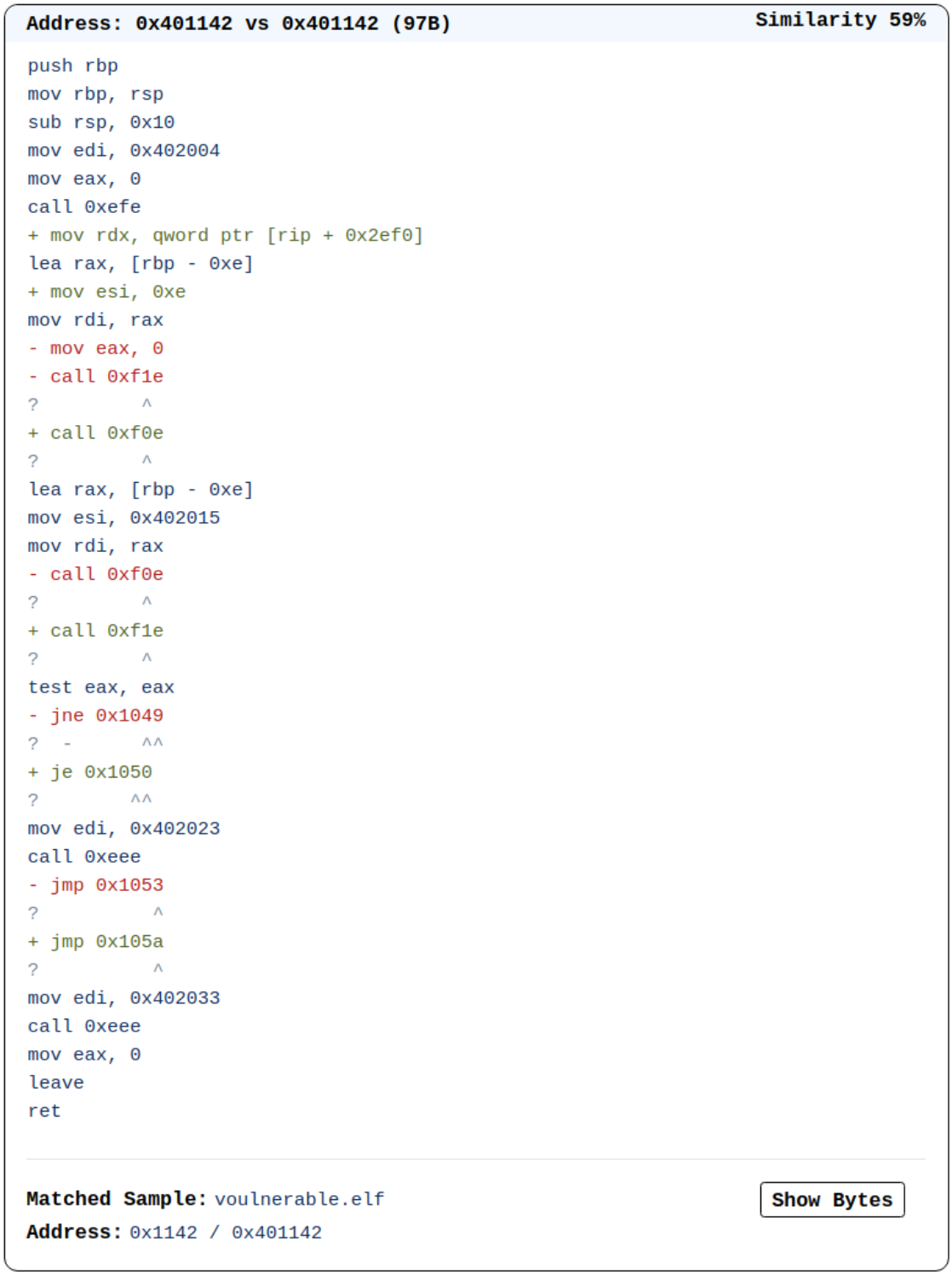


Figure A.3: The patched function found by YaraZilla.

Appendix B

Extraction Constraints

Figure B.1: Setting the size of basic blocks low yields many common constructions.

Appendix C

Evaluation of Dataset

```
"win.matrix_banker": {
        "total": 332
        "win.balkan_door": 6, "win.stop": 5,
        "win.breach_rat": 4, "win.funny_dream": 2,
        "win.skipper": 2, "win.milum": 2,
        "win.tonedeaf": 2, "win.thanatos": 2,
        "win.mokes": 2, "win.hotcroissant": 1,
        "win.bolek": 1, "win.kwampirs": 1,
        "win.xxmm": 1, "win.adhubllka": 1,
        "win.ismagent": 1, "win.retefe": 1,
        "win.unidentified_060": 1, "win.pupy": 1,
        "win.taintedscribe": 1, "win.coredn": 1,
        "win.wastedlocker": 1, "win.dented": 1,
        "win.sedreco": 1, "win.gootkit": 1,
        "win.wndtest": 1, "win.zeus_openssl": 1,
        "win.torrentlocker": 1, "win.gophe": 1,
        "win.ketrican": 1, "win.badnews": 1,
        "win.ddkeylogger": 1, "win.kagent": 1,
        "win.helminth": 1, "win.mariposa": 1,
        "win.dadstache": 1, "win.flawedammyy": 1,
        "win.teslacrypt": 1, "win.raccoon": 1,
        "win.flusihoc": 1, "win.rhino": 1,
        "win.adkoob": 1, "win.phandoor": 1,
        "win.babar": 1, "win.ryuk_stealer": 1,
        "win.blindingcan": 1, "win.dadjoke": 1,
        "win.ketrum": 1, "win.purplewave": 1,
        "win.plugx": 1, "win.vidar": 1,
        "win.dtrack": 1, "win.keyboy": 1,
        "win.mespinoza": 1, "win.electricfish": 1
    }
```

Figure C.1: Database dump for Matrix Banker similar families.

Summary

Comparing with win.mespinoza **Min size** 41B

Matched bbs 1 **Max size** 41B

Address: 0x3787225 (41B)

```
push ebx
push esi
push edi
mov eax, dword ptr [esp + 0x10]
push ebp
push eax
push -2
- push 0x43da90
+ push 0x37871e0
push dword ptr fs:[0]
- mov eax, dword ptr [0x475234]
?                        ^^^^^
+ mov eax, dword ptr [0x3794008]
?                       +++ ^^^
xor eax, esp
push eax
lea eax, [esp + 4]
mov dword ptr fs:[0], eax
```

Bytes:

5356578b44241055506afe68e071780364ff3500000000a10840790333c4508d44240464a300000000

Physical address: 0x16625

Figure C.2: The basic block Matrix Banker shares with Mespioza and other families.

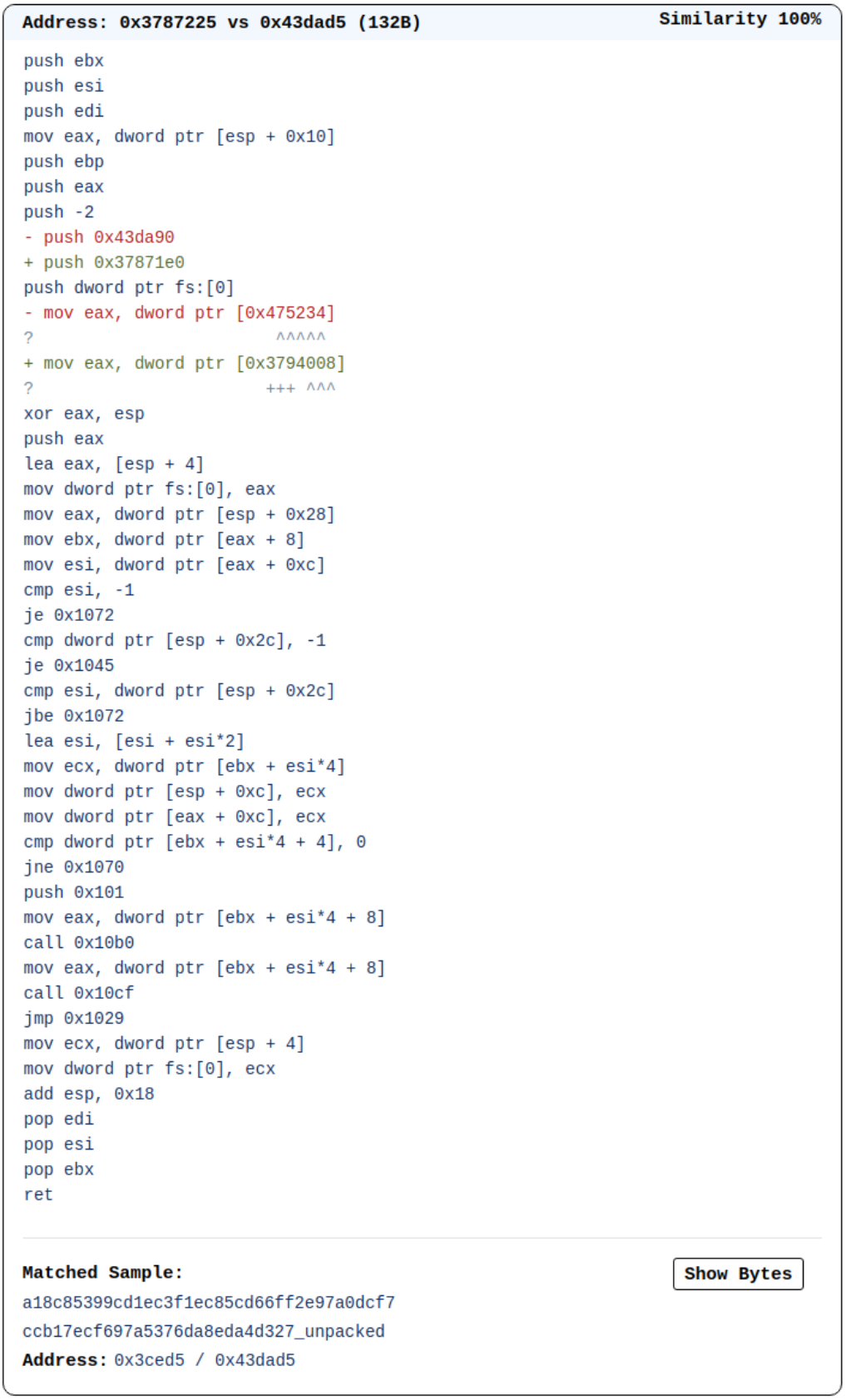


Figure C.3: The one shared basic block of Matrix Banker in context of commonly shared function.

Appendix D

Included Content

The medium included with this work contains the following content:

- text of the work in PDF format – `dp-xkubov06.pdf`,
- directory `tex` with LaTeXfiles,
- directory `yarazilla` containing all the implemented source codes,
- file `README.md`.