Bachelor’s Thesis
submitted in partial fulfilment of the
requirements for the course “Applied Computer Science”

Peeking behind the curtains:
Automatically Deriving Malware Signatures from Commercial AV-Scanners

Kevin Freeman

Institute of Computer Science

Bachelor’s and Master’s Theses
of the Center for Computational Sciences
at the Georg-August-Universität Göttingen

May 5th 2015
I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Göttingen, May 5th 2015
Abstract

Anti-virus scanners gain more and more interest due to the increasing size of the internet and thus the increasing incidences of malicious files. Only little is known about the inner functionality of such anti-virus scanners. The core is constructed by the signature databases, which are used in order to classify programs for maliciousness.

In this thesis, an attack on the database privacy is executed in order to find out the possible signatures, that are used in each anti-virus scanner. A total of four scanners have been tested. Different derivation approaches for this attack are described whilst using malicious file samples. Afterwards, the derived signatures are used to mark benign files as malicious. Finally, the evaluation shows, that approximately 63% of the signatures for a set of samples are derivable for the open source scanner ClamAV within a short time. Two of the other three scanners reach good results as well. The last one, however, presents only a success rate of 18%. Additionally, cross-AV detection tests show, that a few shared signatures exist. Finally, the runtime of these approaches will be evaluated.

Zusammenfassung


Contents

1 Introduction

2 Malware Signatures
   2.1 Pattern-matching
      2.1.1 Aho-Corasick
      2.1.2 Boyer-Moore
      2.1.3 Static Analysis
   2.2 Smart Scanning
      2.2.1 Reducing Scanned Amount of a File
      2.2.2 Reducing Amount of Scanned Files
   2.3 Cryptographic Methods
   2.4 Excursion: ClamAV Database
   2.5 Conclusion

3 Deriving Malware Signatures
   3.1 Filetype Independent Approaches
      3.1.1 Byte and Nibble Classification
      3.1.2 Byte-Replacement Strategies
      3.1.3 Gap Sizes
   3.2 Filetype Dependant Approaches
      3.2.1 Relevance of theEntrypoint and Sections of PEs
      3.2.2 Exact Position of the Signature

4 Evaluation
   4.1 Sample Set
   4.2 Evaluation Strategy
      4.2.1 Runtime Limitations
      4.2.2 Memory Limitations
   4.3 Derived Signatures vs. ClamAV Database
   4.4 Injection of the Derived Signature into a File
CONTENTS

4.4.1 Testing the Corresponding Scanner ........................................ 27
4.4.2 Cross-AV Detection ............................................................ 29
4.5 Runtime ............................................................................... 30

5 Limitations .............................................................................. 33
  5.1 Additional Wildcard Gaps ....................................................... 33
  5.2 Boolean Operations within the Signature ................................. 34
  5.3 Multiple Occurrences of Substrings ........................................ 34
  5.4 Fileformat Dependencies ....................................................... 35

6 Conclusion ............................................................................. 37

Bibliography ............................................................................. 39
List of Figures

2.1 Initialisation of the trie. ........................................ 5
2.2 Adding eleven and duel. ....................................... 5
2.3 Adding eve and duel and extending the node with index 0. ........................................ 6
2.4 Extract of the byte-based signature DB. ........................................ 11
2.5 Extract of the hash-based signature DB. ........................................ 12
2.6 Extract of the boolean-based signature DB. ........................................ 12
3.1 Scanning for important bytes, only showing first 4 splits. ........................................ 17
3.2 Testing the relevance of the entry point. ........................................ 19
3.3 Injecting relevant bytes at different reference points. ........................................ 20
4.1 Timing random and constant approaches in Python ........................................ 23
4.2 Levenshtein distance between derived signatures and the ClamAV database ........................................ 25
4.3 Nibble wildcards can be included in a gap. ........................................ 26
4.4 PE-Header included in signature. ........................................ 26
4.5 Additional wildcard gaps. ........................................ 26
4.6 Logical features which were replaced with *. ........................................ 26
4.7 Same content before and after a gap. ........................................ 26
4.8 Levenshtein distance between derived signatures and the ClamAV database including gaps. ........................................ 27
4.9 Injecting derived signatures. ........................................ 28
4.10 Relation for ClamAV signatures. ........................................ 28
4.11 Relation for AV1 signatures. ........................................ 28
4.12 Relation for AV2 signatures. ........................................ 29
4.13 Relation for AV3 signatures. ........................................ 29
5.1 Additional wildcard gaps. ........................................ 33
5.2 Logical features which were replaced with *. ........................................ 34
5.3 Same content before and after a gap. ........................................ 34
List of Tables

2.1 Initialisation of depth $d = 1$ ........................................ 6
2.2 Additions for depth $d = 2$ ......................................... 6
2.3 Consecutively for depth $d = 3$ ...................................... 6
2.4 Final table ............................................................... 7
2.5 Output function having regard to the Fail-function .......... 7

4.1 Scan results of various AVs ........................................ 22
4.2 Cross-AV detection result ............................................ 30
4.3 Runtimes per sample for ClamAV including gap sizes ...... 30
4.4 Runtimes per sample for all scanners ............................ 30
Chapter 1

Introduction

Anti-virus (AV) scanners rely on a database (DB) of signatures to detect malicious files. These signature databases are often kept private and therefore, there is only little information about the structure and functionality of used signatures. Additionally, there is only very limited related work.

In this thesis, the open source ClamAV is inspected to gain insight into structures of used signatures in other anti-virus scanners. As a first step, different types of signatures are reviewed and evaluated for their possibility of being used in an AV (Chapter 2). Additionally, an extract of the ClamAV database is given to present the used structures of signatures in this thesis. Afterwards, in Chapter 3 approaches are described to automatically derive the signatures for a specific AV scanner given malicious samples. These approaches are tested for four different scanners, including ClamAV. The derived signatures can be used to give general insight into anti-virus scanners. Furthermore, they can be used to build an own database of signatures without manual crafting. After completing this procedure, the derived signatures have to be evaluated in terms of correctness in Chapter 4. This can be done by comparing them to known databases. In this case, the ClamAV DB. Another approach of evaluating the correctness is the injection into benign files. If they are classified as malicious afterwards, the correct signature has been derived. Successful injections may be used afterwards to create false positives. Additionally, inter anti-virus scans will be done, to review shared signatures. The runtime of the approaches will be evaluated as well. Due to the fact, that signatures may have unfavourable structures, there are certain limitations which will be reviewed in Chapter 5. Some types of signatures might be undervisible because of their structure. These structures will be presented. Finally, an assessment of the approaches is done in the conclusion in Chapter 6.
Chapter 2

Malware Signatures

To be able to automatically derive AV signatures we have to understand how they possibly appear in databases and how they are used. There are different methods and algorithms to create and use signatures. In this chapter the open source anti-virus scanner ClamAV is analysed. From this point of view, it is assumable which types of signatures potentially are better than others or more certainly used in AV scanners. In this case ‘better’ implies a lower false negative as well as false positive rate. These methods can be split up into three groups: (1) static methods, (2) dynamic methods and (3) cryptographic methods.

Static methods read the content of a file and analyse it. On the one hand one can search for certain byte sequences by using pattern matching algorithms and on the other hand one can search for certain passages that point to viral behaviour to heuristically classify the file as done by J. Aycock [1]. A similar approach is used by J. Aycock for dynamic methods. In contrast to static methods, the behaviour of the file at runtime is reviewed. In this case for example system calls can be logged. If there is a malicious sequence of operations, the file can be classified as malicious as well. However, when using dynamic methods, it is compulsory that the malicious activity is logged. For example a virus that only executes viral code at certain times of the day or when run outside of a virtual machine can easily evade these mechanisms. Cryptographic methods generate a hash or checksum for a malicious file as seen in Chapter 2.3 later on.

2.1 Pattern-matching

Pattern-matching methods rely on the bytes of the file. They inspect the content bytewise and search for specific sequences or substrings. In our case, this substring is a virus signature. The simplest way of comparing signature and file content is the iterative approach.
CHAPTER 2. MALWARE SIGNATURES

Example 1 (Iterative signature comparism)

Let

\[ f = \text{4d 5a 00 00 ff 84 00 f3 84 fa} \]

be the bytewise hexadecimal content of a file and let

\[ s = \text{00 ?? 84 fa} \]

be the signature that has to be matched. ?? denotes a wildcard byte and will match any byte.

The matching takes place iteratively and on mismatch the signature will be shifted by one byte and the comparism starts over:

\[
\begin{align*}
\text{4d 5a 00 00 ff 84 00 f3 84 fa} &= f \\
\text{00 ?? 84 fa} \\
\text{00 ?? 84 fa} \\
\text{00 ?? 84 fa} \\
\text{00 ?? 84 fa} \\
\text{00 ?? 84 fa}
\end{align*}
\]

This example needs many comparisms of bytes to inspect a file for a match of one signature. AV scanners use many signatures which have to be applied to a single file all at once or successively. Therefore, this algorithm works but consumes a lot of performance as well. In ClamAV two different algorithms were implemented to avoid long runtimes:

2.1.1 Aho-Corasick

This algorithm was published by A. Aho and M. Corasick and is based on a finite non-reentrant deterministic state machine with states \( S \) and alphabet \( \Sigma \), or in short trie, which will be constructed from multiple signatures. Afterwards, the content of a file can be tested for all signatures at once. The whole algorithm can be split up into three functions:

1. GoTo-function, uses signatures to build up a trie
2. Fail-function, creates failure links in the trie
3. Output-function, marks some states as accepting, which implies that a signature was successfully matched

The GoTo-function constructs a trie out of a set of wildcardfree signatures. Otherwise, the total amount of states would be immense and possibly the construction will fail due to high workload.
GoTo-function

The GoTo-function \( g(s, e) \) with state \( s \in S \) and letter \( e \in \Sigma \) is defined as follows:

\[
g(s, e) = \begin{cases} 
\text{index of next node in trie, if edge from } s \text{ with label } e \text{ exists} \\
\text{fail, otherwise}
\end{cases}
\]

In other words, the GoTo-function will evaluate if there is a path with the current letter \( e \) or not. If there is no such path, an intended error will occur and the \text{Fail}-function will be called. Furthermore, the first node of the trie is the only node that will never cause an intended fail. It is the only node that is provided with an additional edge, that will link to itself. Resulting in \( g(0, e) \) returning an index bigger or equal to zero in any case.

This first node initialises the trie. The algorithm extends it iteratively for each signature. It starts at the node with the index \( 0 \) and creates a path for each letter of the signature if necessary. If an edge already exists it will be traversed and not recreated.

Example 2 (Construction of the GoTo-trie)

Let \( s = \{ \text{eve, eleven, duels, dud} \} \) be our set of signatures. In this case, we use strings to simplify the illustration.

![Figure 2.1: Initialisation of the trie.](image1)

Now we iterate through our set of signatures and extend the trie step by step.

![Figure 2.2: Adding eleven and duel.](image2)
CHAPTER 2. MALWARE SIGNATURES

Figure 2.3: Adding \textit{eve} and \textit{dud} and extending the node with index 0.

\textbf{Fail-function}

The \textit{Fail}-function is necessary in order to combine coinciding pre- and suffixes of signatures. This can occur in different paths of the state machine and therefore have to be taken care of. This information is stored in a table to guarantee low storage and access time. If a \textit{fail} of the \textit{GoTo}-function occurs, we see which node to consult next.

Each node with a depth of \(d = 1\) is initialised with 0. Afterwards, the depth is iteratively traversed. For each node \(z\) of depth \(d\) with underlying node \(s\) (hence \(g(z, e) = s\)) will be reviewed as follows:

1. Initialise \(v\) with \(f(s)\)
2. Loop: Set \(v\) to \(f(v)\) as long as \(g(v, e)\) returns \textit{fail}
3. Finally set \(f(z) = g(v, e)\)

\textbf{Example 3 (Construction of the \textit{Fail}-function table)}

\textit{This example relates to the previous one.}

\begin{table}[h]
\centering
\begin{tabular}{l|cccccccccccc}
  \(i\) & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
  \hline
  \(f(i)\) & 0 &  |  &  &  |  &  & 0 & |  &  &  |  &  &  |  \\
\end{tabular}
\caption{Initialisation of depth \(d = 1\).}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{l|cccccccccccc}
  \(i\) & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
  \hline
  \(f(i)\) & 0 &  &  &  &  &  & 0 &  &  &  |  &  &  |  \\
\end{tabular}
\caption{Additions for depth \(d = 2\).}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{l|cccccccccccc}
  \(i\) & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
  \hline
  \(f(i)\) & 0 & 0 & 1 &  |  &  &  | 0 & 1 |  & 0 & 1 |  &  \\
\end{tabular}
\caption{Consecutively for depth \(d = 3\).}
\end{table}
Output-function
If we reach an accepting state in the deterministic state machine we have found a successful match of a signature. The Output-function will take care of adding all necessary accepting states to the trie. Obviously, while adding each signature the end of the signature will be marked as accepting, because the signature ends here. This can be seen in Figure 2.2 for example. But if a fail link from a non-accepting node to an accepting node exists, we can mark this one as accepting as well, due to the fact, that a signature can contain parts of other signatures again.

Example 4 (Output-function table)
This example relates to the previous one. As stated before, each node at the end of a string is marked as accepting and therefore added to the Output-table. Whilst matching eleven, the signature eve is found as well. The fail link $f(5) = 12$ shows us, that $output(12)$ has to be added to $output(5)$.

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(i)$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.5: Output function having regard to the Fail-function.

2.1.2 Boyer-Moore
This algorithm was published by R. Boyer and J. Moore [3] and uses some properties of strings, that allow to skip parts of the input, without further inspection. The input will be for example the content of a file and a signature. In the following, both will be called strings. In contrast to the
Aho-Corasick algorithm this algorithm is able to use signatures including wildcards. The first letters of both strings get aligned. Afterwards, the strings are compared from right to left, beginning at the end of the signature. While doing so, two different heuristics are used. On the one hand the *Bad-Character-Heuristic* is used and on the other hand the *Good-Suffix-Heuristic*

### Bad-Character-Heuristic
Both strings will be compared until a mismatch occurs. If a mismatch occurs, there are two cases:

a. the mismatched letter of the input is not existent in the signature, which leads to a right shift of the signature equals to its length

b. the mismatched letter of the input exists in the signature, therefore, it will be aligned with the rightmost same letter of the signature

This heuristic will be called *Case 1* in the following example.

### Good-Suffix-Heuristic
In this heuristic both strings will be compared as before, but on mismatch the current suffix of the signature will be inspected for further occurrences in the signature. If there is any, the signature can be shifted right in order to align this suffix with the input.

While using the Boyer-Moore algorithm both heuristics will be evaluated. This can end up in different extents of rightshifts. The heuristic with the greater rightshift will be chosen in such case. This heuristic will be called *Case 2* in the following example.

#### Example 5 (Searching for a string in an input using the Boyer-Moore algorithm)

*Let*

\[ \text{IGHE.UCKWOFGA.SNENLEARNABLE} \]

*be our input and we are supposed to find the string LEARNABLE in it.*

\[ \downarrow \]

**IGHE.UCKWOFGA.SNENLEARNABLE** = Input

**LEARNABLE** = String to be found

**LEARNABLE**

*Case 1a: W does not occur in the string*

\[ \downarrow \]

**IGHE.UCKWOFGA.SNENLEARNABLE** = Input

**LEARNABLE** = String to be found

**LEARNABLE**

*Case 1b: N occurs in the string, Case 2: E is a good suffix*
Example 5 demonstrates the use of the original Boyer-Moore algorithm. A slightly different version is used in ClamAV, but no paper exists about this. However, in ClamAV Case 1a of the Bad-Character-Heuristic aligns the next occurrence left of the current position instead of the rightmost occurrence in the string.

### Runtime

Both heuristics are saved in tables. On the one hand, the distance of the last occurrence of each letter in the string, on the other hand, the suffix locations are saved. If we assume that the accesses to the tables lie in $O(m)$, it is possible to reach a worst-case complexity of $O(m + n)$, where $n$ is the size of the string and $m$ the size of the input and the string does not occur in the input. Otherwise, the complexity is in $O(m \cdot n)$, if we assume that every occurrence has to be found in the input.

### 2.1.3 Static Analysis

In this section some heuristic methods are presented, that try to classify a file on a basis of limited knowledge about the file, as described by J. Aycock [1]. Certain fragments of the file can be classified. On one side a so called **booster** shows a presence of viral behaviour and on the other side a **stopper** points away from such behaviour.

Typical contents of **booster** sections are:

1. loops, that decrypt parts of the file
2. selfmodifying code
3. usage of undocumented API-calls
4. change of certain jump adresses
5. usage of instructions, that cannot be created by a compiler
6. strings that include viral information
CHAPTER 2. MALWARE SIGNATURES

In contrast a stopper is for example the creation of a dialog box. Furthermore, there are other features that can be used to classify the file. For example, if there is an uniformly distributed spectrum of used bytes a higher possibility of reviewing an encrypted file can be assumed. However, at the end there has to be some kind of calculation of a score. If it reaches a certain threshold, the file may be classified as malicious.

2.2 Smart Scanning

The presented algorithms need a lot of performance, due to the fact, that there are thousands of files to be scanned on a single run. To reduce the scanned amount, there are two different heuristics. Firstly, classifying the file without scanning it completely and secondly, scanning only a fraction of all files.

2.2.1 Reducing Scanned Amount of a File

Normally, viruses append themselves to the beginning \((=\text{top})\) or end \((=\text{tail})\) of a file. Accordingly, it is possible to scan these parts. This approach is called \textit{top and tail scanning}. Additionally, viruses of small size are more common. Therefore, only some bytes have to be inspected at these points. This heuristic can be transferred to executable files. In this case, e.g., the entry point of the file is a main target of viruses. A further approach is the deletion of all irrelevant passages in the buffered filecontent. For example \textit{no operation}-instructions or line breaks may be deleted or disregarded as they can be easily added by the virus author.

2.2.2 Reducing Amount of Scanned Files

This heuristic avoids scanning files multiple times if unnecessary. Additional data is stored for each file, for example, the last time the file was edited. Hence, the scanner can ignore the file, if it was not modified since the last time scanned. Of course, this information could be faked by the virus to circumvent such mechanisms and therefore, this additional data should be located and stored safely. Furthermore, one has to think about the definition of a modification of a file and how to check it. If more performance is needed to complete these checks than needed to rescan the file, the use of mechanisms like this makes no sense at all.

2.3 Cryptographic Methods

Hash functions are often used in cryptographic algorithms. These are build to map a unique value to an input whilst reducing the ratio of the size of input and output dramatically. The outputs of
these hash functions are called checksums or hashes and are often used for integrity checks of files. If a hash for a malicious file is created and saved in a DB it can be easily compared to hashes of other files. Due to the uniqueness of the hash a match should be only found, if the exact same input was hashed. Therefore, the false positive rate is very low. But if only one bit of the malicious was changed, the whole hash varies. Accordingly, a hash can be easily evaded by adding operations, that are not changing the functionality but change the hash of the program, for example no operation instructions. However, there are certain approaches that can be used in AV scanners such as piecewise hashing and rolling hashes. The former one relies on creating several subhashes for a file by splitting it into parts and generating a hash for each of them. The latter approach creates a checksum for a certain window of the file and saves e.g. hashes at certain points. There are many different ways of using these approaches. Therefore, they are not in focus of this thesis.

2.4 Excursion: ClamAV Database

To get a better feeling of the particular design of signatures and how they will be used in the main part of this thesis, some examples can be seen in the following Figures 2.4 and 2.5.

![Figure 2.4: Extract of the byte-based signature DB.](http://www.clamav.net/doc/latest/signatures.pdf)

These signatures have the following structure: **Name of the virus**:datatype:offset:bytes. The datatype is assigned by a simple number, which can be found in the signature creation documentation of ClamAV.

**Definition 1 (Nibble)**

A nibble represents half a byte and therefore, consists of 4 bits. Hence, each byte is composed of a high and a low order nibble, or left and right nibble.

The **bytes** relate to regular expressions. It may be extended by the use of nibble wildcards (Definition 1), e.g. \(??\) or \(?2\), wildcards, e.g. \(??\), or boolean features like \((aa|bb|22|\ldots)\), which allow matching more than one byte at a certain position. Additionally, wildcards may have a whole interval of bytes, as seen above. \([1–20]\) in this case equals the possibility of matching one to twenty arbitrary bytes. The length of the gap may also be arbitrary in length (*) or only have

\[\text{shortened for visualisation}\]
one boundary as in e.g. [–10] or [10–]. The offset may be used for smart scanning purposes. EP references the entry point in Figure 2.4. Sections may also be used as reference points.

87040:ec0b983ad08e727b5260af375fc0aeb2:Worm.Gaobot-325
88420:a15c6ee5c4fda4fe96606d464ebc9ad4:Trojan.Spy-763

Figure 2.5: Extract of the hash-based signature DB.

The hash-based signatures however use a specific offset and a md5-hash. The hash will be calculated from the offset to the end of the file. The number $n$ in front of the hash equals the length of the input of the hashfunction in bytes. Hence, the offset is calculated by $(\text{file size}) - n$. These two types of signatures build the core of the ClamAV database with over 99% of all signatures. Next to them, there are certain boolean-based signatures, which are based on boolean equations and subsignatures as shown in Figure 2.6.

HTML.Exploit.C99-1;Engine:51-255;Target:3;\((0\&1\&2\&3);66...;2e...;6c73;2d...\)
JS.Exploit.CVE_2011_1997-1;Engine:51-255;Target:3;\((0\&1|2));3c...;3d20...;3d6e...\)

Figure 2.6: Extract of the boolean-based signature DB.

The first one boolean-based signature shows a simple signature, consisting of 3 subsignatures. As conjunctions are the only logical features, a derivation is potentially possible. The second one, however, shows a disjunction combined with conjunction. This virus family is therefore detected by two separate signatures (0|1 or 0|2). The later presented approaches try to generate a single signature for one virus family. As only a fraction of the signature database capacity is used to store these signatures, we will focus on hash- and byte-based signatures.

2.5 Conclusion

There are many methods that can be used to detect malicious files. AV scanners try to have a low false positive rate while classifying files. Some methods can guarantee a lower false positive rate than others.

However, we had a look at dynamic approaches. In this case, a classification is done by analysing the behaviour of the file. The behaviour of a file could change with the time and only express malicious activies at certain events, for example if run outside a virutal machine. Some kind of threshold has to be implemented to decide whether an activity is malicious or not. Therefore, a classification based on dynamic analysis is potentially not combined with a low false positive rate. The static analysis in contrast analyses the file based on the static content. The file is not executed and just viewed from the outside. Of course, this is only possible if the file is not encrypted, or encrypted and an appropriate decrypter is available. However, static analysis can only guarantee a
2.5. CONCLUSION

low false positive rate, if the signatures are crafted well. As an additional feature, these pattern matching algorithms may even include wildcard gaps in order to classify for example whole virus families with a single signature. Crafting signatures needs a lot of time and therefore, one might think of using a hash signature for a quick ensured classification of a single virus. Hash signatures imply a low false positive rate due to the fact, that there is only a low chance of occurring collisions. This of course depends on the used hash function. However, if a virus is found and no time exists to craft a byte-based signature, a hash signature is a good substitution until a better signature can be crafted. Hash signatures are easy avoidable due to the fact, that if only one bit is changed in the signature region of the file, the whole hash changes. In such cases it might be better to use piecewise hashing, as it tries to avoid something like this. Summarized one can say, that AV scanners potentially use byte- and simple hash-signatures for classifications as ClamAV does as well. These might use some smart scanning features as seen in Section 2.4.
Chapter 3

Deriving Malware Signatures

This chapter includes several aspects of signatures that have to be taken care of and how to approach them. The objective is to automatically derive signatures from a particular AV scanner given a set of malicious samples. A total of four scanners is tested, including the open source AV scanner ClamAV. For some methods the filetype of the virus sample is relevant: Smart scanning, for instance, relies on reference points which are only defined in certain filetypes. Therefore, the different approaches are covered in two separate sections: Firstly, approaches that do not depend on the filetype (Section 3.1) and secondly, those that do (Section 3.2).

3.1 Filetype Independent Approaches

As explained in Chapter 2 most useful signatures are based on bytes. Therefore, it is necessary to test the bytes of malicious files for significance for the signature. There are certain other features of signatures to be considered, though. As a first step, relevant bytes have to be found. Additionally, each nibble (cf. Definition 1) of those relevant bytes are flipped in order to determine the relevance of the nibble for the signature.

Definition 2 (Nibble flip)

Let \( b \) denote a byte, \( b_1b_2 = b \) the nibbles of the byte and \( n \in \{1, 2\} \) the nibble to be flipped. The nibble flip is defined as follows.

\[
\text{NibbleFlip}(b, n) = \begin{cases} 
(b \land 0x0F) \lor (\neg b \land 0xF0), & \text{if } n=1 \\
(b \land 0xF0) \lor (\neg b \land 0x0F), & \text{otherwise}
\end{cases}
\]
3.1.1 Byte and Nibble Classification

With respect to malware signatures a malicious file can be classified into three groups:

1. **Important bytes.** Bytes that lead to benign classification of the scanner when changed.

2. **Wildcard bytes.** Bytes that may have any value and exist between important bytes.

3. **Irrelevant bytes.** Bytes that are not between important bytes.

Since important and wildcard bytes are the most significant parts of a signature, these have to be identified first. The straightforward attempt is bruteforcing them by copying the malicious file multiple times and replacing one byte in each sample. If a sample is not malicious anymore according to a particular scanner it can be concluded, that the replaced byte is relevant to the signature. The replacement is done by substituting the existing byte with another one.

**Example 6 (Multiple sample generation)**

In this example each byte is replaced by 0x84, but replaced with 0xcc if the byte equals 0x84 already. Furthermore, let

\[ f = 4d\ 5a\ 90\ \ldots\ f3\ 84\ fa \]

be the bytewise hexadecimal content of a malicious file and \( n \) be the size of the file in bytes.

The file \( f \) is copied \( n \) times to sample files \( f_1, f_2, \ldots, f_n \) and each index will determine which byte was replaced.

\[
\begin{align*}
    f &= 4d\ 5a\ 90\ \ldots\ f3\ 84\ fa \\
    f_1 &= 84\ 5a\ 90\ \ldots\ f3\ 84\ fa \\
    f_2 &= 4d\ 84\ 90\ \ldots\ f3\ 84\ fa \\
    f_3 &= 4d\ 5a\ 84\ \ldots\ f3\ 84\ fa \\
    \vdots &\quad \vdots \\
    f_{n-2} &= 4d\ 5a\ 90\ \ldots\ 84\ 84\ fa \\
    f_{n-1} &= 4d\ 5a\ 90\ \ldots\ f3\ cc\ fa \\
    f_n &= 4d\ 5a\ 90\ \ldots\ f3\ 84\ 84 \\
\end{align*}
\]

However, this method needs \( O(n^2) \) bytes disk space. A malicious file can easily reach up to 5 MB which would end up in 25 TB sample files. Therefore, this attempt requires a lot of time to create and scan all files. Due to the fact that byte-based signatures consist of a small fraction of a file, a divide and conquer principle can be implemented to speed up the process. Depending on the file size a window size is defined, which determines not only a single byte but a whole sequence of bytes that are replaced at once. The whole window is replaced with a sequence of replacement bytes. Only if the whole window equals this sequence already, other replacement bytes are chosen.
3.1. FILETYPE INDEPENDENT APPROACHES

As shown in Figure 3.1, large parts of the file can be dropped without further inspection (white regions in the figure). If a virus is detected, only irrelevant bytes were replaced and the inspection for this fraction is completed.

![Figure 3.1: Scanning for important bytes, only showing first 4 splits.](image)

3.1.2 Byte-Replacement Strategies

For replacing parts of a file, various strategies are possible. Either bytes are replaced with random values or two constant values might be selected (Example 6). Either way, the uppermost priority to actually alter the bytes in question.

1. Random Value. The first approach is to fill regions randomly. In this case, it could happen that a single byte is replaced with the same value as before with a probability of \( \frac{1}{16} \). If this happens, a nibble flip is used on it to guarantee that the byte actually altered.

2. Constant Value. As a second approach, bytes are replaced with two different constant values. If the byte equals the first constant value before replacement, the second constant value is selected. This can be randomly considered before starting the derivation. However, counting all bytes in the ClamAV DB showed, that 0xa7 and 0xa9 are the least used bytes. These will be the used replacement bytes therefore.

A detailed evaluation is presented in Section 4.2.

3.1.3 Gap Sizes

The derived signatures will contain the bytes that are important for the signature and possible wildcards between them. As a further step the wildcard gaps have to be checked for their size, because they may vary. This is done by using a binary search for a lower and an upper limit of the gap.
CHAPTER 3. DERIVING MALWARE SIGNATURES

Example 7 (Binary search on gaps)

Let

\[ f = 4d\ 5a\ 90\ \ldots\ \text{f3}\ \text{84}\ \text{fa} \]

be the content again and

\[ s = 4d\ 5a\ \{30\}\ 84\ \text{fa} \]

be the signature found with the approach of chapter 3.1.1.

Each gap of the signature \( s \) needs to be checked. We know that exactly 30 wildcard bytes are detected by the scanner. Therefore, the first binary search will try to estimate the lower limit. The lower limit is in the interval \( I = [0; 30] \). Thus the middle of the interval equals \( \frac{0+30}{2} = 15 \). This division is done with integers. A file \( g \) is created and scanned.

\[ g = 4d\ 5a\ \{15\}\ 84\ \text{fa} \]

If the file is classified malicious by the scanner then the search continues in the lower half of the interval, in this case \( I = [0; 15] \), otherwise \( I = [15; 30] \). Note, that the upper limit of this interval was checked already and is therefore part of the signature, the lower one might not. After some steps the lower limit is found and both boundaries of the interval equal each other.

As gaps without any upper limit exist, an upper limit has to be determined. 400.000 is a good upper limit, because there are only 6 out of over 69000 signatures with bigger gaps used in ClamAV and no further insight into other signature databases is given. Most of the wildcard gaps found in signatures are small in size. Therefore, a heurisitic can be applied for gaps smaller than 400. In this case, an upper limit of 400 is tested at first. If the heurisitic succeeds, the binary search is done on the interval \( I = [\text{gap size}; 400] \), otherwise on \( I = [\text{gap size}; 400000] \). The overall procedure is similar to the search for the lower limit, as seen in Example 7. This time however, the lower limit of the interval is always part of the signature, the upper one might not.

3.2 Filetype Dependant Approaches

Approaches described previously rely on the content of the file. The approach explained in this section is additionally dependent on the file format. In the scope of this thesis, only Portable Executables (PE) are analysed. Particular important in this case are, for instance, entry points, which are defined in the PE header. AV scanners may only scan the region around these points, as covered in Chapter 2.2. Therefore, the importance of an entry point or beginning of a section within the file has to be tested. Those points are called reference points from now on.
3.2. FILETYPE DEPENDANT APPROACHES

3.2.1 Relevance of the Entrypoint and Sections of PEs

Detecting the relevance of a certain reference point can be accomplished by shifting the content of the file at the point by one byte. This is achieved by deleting or inserting a byte, as shown in Figure 3.2. If the file is not detected anymore, the relevant bytes are fixed in position and some sort of smart scanning is used.

Note, that from this point of view it is not possible to conclude the point that the signature is relevant to, because adding a byte to the first section of the file will also shift the following sections by one byte.

![Figure 3.2: Testing the relevance of the entry point.](image)

3.2.2 Exact Position of the Signature

Up to this point the following information about the virus family has been derived: (1) relevant bytes and nibbles, (2) gap sizes and (3) relevance of reference points and possible offsets. In the case of PE files, the relevant bytes before the first section are important for the scanner, to find the right reference points and offsets. The signature is potentially located somewhere behind the first section. If these bytes are now copied to another file, it is possible to check the relevance of a specific reference point. This approach heuristically uses only the bytes behind the beginning of the first section. Furthermore, it selects the most occurring signature of the virus family. If there is more than one possibility, the longer one is chosen in regard to the amount of relevant bytes and gaps. These heuristics are evaluated in Section 4.3.

However, if the position of the relevant bytes is fixed, the reference point might be one of the following three possibilities: (1) the beginning of the file, (2) the entry point and (3) a beginning of a section. The offset to a certain reference point may be zero, meaning that the signature starts at the exact same point as the reference point. This however, is not essential.

One possible way of finding these relevant points is to look for the signature in the sample directly. The offset can be found by calculating the distances to each reference point and simply chose the closest one. Nevertheless, viruses may modify the section table and there may exist overlapping sections.
Therefore, an injection of the signature pattern with the corresponding offset to each reference point into a file with a big section table and no other content is done one by one as seen in Figure 3.3. Same colours illustrate same offsets. The offset to the reference points is calculated after deriving the relevant bytes (Section 3.1.1). Due to the fact, that there is a different section table it is possible to test different locations but same offsets. If a file is detected by the AV scanner, the offset with the corresponding reference point has been successfully found. Furthermore, they are saved in a database for further evaluation.

There is a slight difference in the result, if varying file sizes are allowed. As said before, hash-based signatures are calculated until the end of the file. The offset for them can only be verified if the file ends after the injection. Due to this, further injections represent the end of the file.
Chapter 4

Evaluation

The derived signatures are evaluated in terms of correctness after presenting the set of malicious file samples in Section 4.1. This is done by using different approaches:

1. The ClamAV signature database is compared to the derived signature set of the ClamAV scanner. This is done by calculating the Levenshtein distance \[5\] in Section 4.3.

2. An injection of the derived signature pattern into a file is performed in Section 4.4. Afterwards, it is scanned with the corresponding scanner. If the file is classified as malicious, we have derived the signature correctly. A cross validation with other scanners is done as well to see if there are intersecting signatures.

3. Some malicious samples have a unfavourable structure, with the result that only a fraction of important bytes can be found. Therefore, a higher number of samples for the same virus can allow a more specific signature derivation in some cases. This is evaluated by plotting a success rate over the sample amount in Section 4.4 as well.

Furthermore the runtime of the whole process is evaluated in Section 4.5 using the strategies presented in Section 4.2.

4.1 Sample Set

As mentioned earlier, a malicious file has to be given to derive the signature for a certain scanner. The derivation was tested with a set of 3000 malicious files. These files are the 3000 most recent uploads to VirusTotal[https://www.virustotal.com/1] and have been classified as malicious by 30 or more scanners on March 24th 2015.

ClamAV classifies 1275 of 3000 as malicious as of March 31st 2015 as seen in Table 4.1. In total, 191
different virus families were found, from which 75 exist in the byte-signature based DB, 111 in the hash DB, 4 in boolean-signatures, and the last family is detected by a heuristic.

<table>
<thead>
<tr>
<th>last update</th>
<th>ClamAV</th>
<th>AV1</th>
<th>AV2</th>
<th>AV3</th>
</tr>
</thead>
<tbody>
<tr>
<td>detected</td>
<td>31/03/2015</td>
<td>26/03/2015</td>
<td>31/03/2015</td>
<td>14/04/2015</td>
</tr>
<tr>
<td>different virus families</td>
<td>1275</td>
<td>2846</td>
<td>≈ 2828</td>
<td>2958</td>
</tr>
<tr>
<td></td>
<td>191</td>
<td>151</td>
<td>151</td>
<td>138</td>
</tr>
</tbody>
</table>

Table 4.1: Scan results of various AVs.

All 3000 files are executables in order to test the whole deriving process. However, if the malicious files would not be executable, only the approach involving reference points would have been dropped.

### 4.2 Evaluation Strategy

Due to runtime and memory limitations it is necessary to use specific parameters. Otherwise a vast amount of time and disk space could be spent on the calculation of, for example, a hash signature.

#### 4.2.1 Runtime Limitations

In some cases it happens, that a single scan needs a lot of time. The inspection is stopped after 3 minutes. In such cases a hash signature is possibly used to detect the virus present and therefore, many bytes of the file appear relevant. A hash signature normally starts at some offset and is calculated until the end of the file. The presented approach only derives the byte-based signatures and hash-bases signatures, that contain a small amount of relevant bytes. If a sample is infected, the scanner returns the name for the encountered virus. Therefore, the signatures of the same virus family can be easily stored in the same file.

Furthermore, byte replacement strategies have to be selected. Testing both approaches mentioned in Section 3.1.2 shows very few differences. While using random replacement strategies a byte was replaced with one, that is included in a logical feature of a signature. Only if these features or nibble wildcards are included in the signature, both strategies imply a possibility of replacing a byte such that a gap will be concluded (Example 3).  

\footnote{files including more than one virus were counted more than once and have been substracted}
4.2. EVALUATION STRATEGY

Example 8 (Byte replacement strategies)

Let \( s = (ab|a7) \) be a signature. The sample with content \( c = ab \: fc \: fc \) is tested. The \( ab \) will be replaced by \( a7 \) in the constant value approach. Therefore, it will fulfill the signature criteria and the AV scanner detects it. This leads to an assumption of a wildcard byte at this position. However, the random value approach could change the second byte to, e.g., \( \text{fd} \) and fulfills the signature as well. There is a 1/16 possibility of replacing with a wrong byte while using the random approach, because there is a nibble wildcard.

Constant values are selected for ClamAV and AV1 due to better runtime of used methods as seen in Figure 4.1. As the amount of relevant bytes is fixed for each sample, the same amount of bytes has to be replaced no matter which replacement strategy is used. AV2 and AV3 are tested using the random approach due to a better environment (cf. Section 4.5). If enough time is available, deriving multiple times with different replacement strategies would potentially lead to the best result.

```
import time
import os

total=[0,0]
for TESTSIZE in range(1,10000000,10000):
    start=time.time()
    _=os.urandom(TESTSIZE) #random
    total[0]+=time.time()-start

    start=time.time()
    _=b'\xFF'*TESTSIZE #constant
    total[1]+=time.time()-start

print "Random: ",total[0]
print "Const.: ",total[1]
```

Output

```
Random: 0.00399994850159
Const.: 0.0
...
Random: 0.0360000133514
Const.: 0.000999927520752
...
Random: 21.7019970417
Const.: 1.58700299263
```

Figure 4.1: Timing random and constant approaches in Python

4.2.2 Memory Limitations

Another limiting factor is the free space on the drive. Therefore, I limited the amount of created samples to 20 to 40 GB by calculating a certain window size as follows: If the size of the sample is below 4472 bytes (\( \approx \sqrt{20.000.000} \)) the file is divided into approximately 1000 parts by using the following window size

\[
\text{window}_{\text{approx}} = \frac{\text{filesize}}{1000}
\]
Otherwise the window size is calculated by
\[
\text{window}_{\text{approx}} = \frac{\text{filesize}^2}{20,000,000}
\]
in order to stay in the interval of 20 to 40 GB. The exact window size is chosen to represent a power
of 2 to guarantee an easy division by 2 and is calculated by
\[
\text{window}_{\text{exact}} = 2^\left\lfloor \log_2(\text{window}_{\text{approx}}) \right\rfloor
\]
This window size is then applied for the divide and conquer principle presented in Figure 3.1.

4.3 Derived Signatures vs. ClamAV Database

Only the database of ClamAV is publicly available. Therefore, the Levenshtein distance [5](Definition 3) is computed between signatures derived using ClamAV and those stored in the correspond-
ing signature DB.

Definition 3 (Levenshtein distance)
The Levenshtein distance takes two strings \( s_1 \) with length \( m \) and \( s_2 \) with length \( n \) as input. It then calculates
the number of operations needed, to transform \( s_1 \) into \( s_2 \). This can be done by adding, replacing or deleting
a character of \( s_1 \). A distance matrix \( D \) is calculated therefore, which is done by dynamic programming:

\[
\begin{align*}
D_{0,0} &= 0 \\
D_{i,0} &= i, 1 \leq i \leq m \\
D_{0,j} &= j, 1 \leq j \leq n \\
D_{i,j} &= \min \left\{ \begin{array}{l}
D_{i-1,j-1} + 0 \text{ (equality)} \\
D_{i-1,j-1} + 1 \text{ (replacing)} \\
D_{i,j-1} + 1 \text{ (adding)} \\
D_{i-1,j} + 1 \text{ (deleting)}
\end{array} \right.
\end{align*}
\]

Afterwards, \( D_{m,n} \) represents the Levenshtein distance.

Before comparing both signature sets, each wildcard-gap is substituted with a single gap character
in order to create a penalty of 1 for missing gaps. Afterwards, the virusname is searched for in
the ClamAV byte-based database. The distance is calculated between the signature and derived
signature. There might be a variety of different signatures derived for each virus family. On the
one hand, the most occurring signature but on the other hand, the longest signature in regard to
the amount of relevant bytes can be selected. Both heuristics were evaluated and only in one case,
different signatures have been selected as seen in Figure 4.2. In this case, two signatures were derived for one family. The amount of occurrences of each of them equals one. One of them is more accurate due to an additional gap, which reduces the distance by one. For following approaches both heuristics are used successively. While doing so, it is possible to find such small differences and select the better signature.

A total of 79 different signatures has been derived. As there are only 75 byte-based signatures in the set, other signature types have been derived, too. 67 of those 79 can be found in the byte-based signature database of ClamAV whereof 31 were without difference (Figure 4.2). Two of those 67 signatures are used before the first section of the file. The heuristic that signatures lie behind the beginning of the first section failed, as all derived relevant bytes before the first section were without difference. The other 12 signatures were not compared and can be grouped into different signature classes: \textbf{2 heuristics, 3 boolean-based signatures and 7 hash signatures}. These do not appear in the byte-based database. A manual lookup showed, that the signatures for the hash signatures included the complete fraction of the file, which is used to generate the hash for the specific virus. Both, heuristics and boolean based signatures, will not be inspected due to low significance for the whole comparism and derivation. Additionally, boolean based signatures are based on logical trees consisting of subsignatures and are therefore not easily comparable.

Further inspections show, that the signature differences originate from five different sources, as seen in the following Figures 4.3 to 4.7 (differences are marked in red).
CHAPTER 4. EVALUATION

Figure 4.3: Nibble wildcards can be included in a gap.

Figure 4.4: PE-Header included in signature.

Figure 4.5: Additional wildcard gaps.

Figure 4.6: Logical features which were replaced with *.

Figure 4.7: Same content before and after a gap.

The first source is avoidable by parsing through the derived signature and replacing such occurrences. The second source only exists due to a signature which is not fulfilling ClamAV signature DB standards and includes information from the PE header. The other sources of differences will be explained further in Chapter 5 (Limitations). However, calculating the Levenshtein distance without replacing the gaps showed, that all derived gaps are equal in size. An increase in distance is only observable, when additional gaps exist, as seen in Figure 4.5. Each underived gap increases the distance by a minimum of three (Figure 4.8). The 31 signatures without distance however show, that the binary search for gap sizes works very well, as they include multiple gaps.
4.4 Injection of the Derived Signature into a File

Now that a DB of derived signatures is created, we inject them into benign PE files. At least signatures that are not using reference points allow an injection without breaking file. Depending on the structure of the benign file others might work as well.

4.4.1 Testing the Corresponding Scanner

In this section a simple injection of the derived pattern is performed. The file is scanned with the corresponding scanner afterwards. If the file is classified as malicious, the signature has been successfully derived. To show that the injection does not break the file, a signature is injected into the scanner itself. If the scanner is able to scan itself and classify itself as malicious, we assume flawless injection (Example 9).

Example 9 (Flawless injection of a signature)

The signature is injected into the `clamdscan.exe` in this example. Afterwards, it is scanned with the infected `clamdscan.exe` itself. The signature has to be appendable at any position, in this case it is injected at the end of the file. The following signature pattern has been used: 8a 07 66 31 c8 88 07 83 c7 01. The scanner classifies itself successfully as malicious.

The final result can be seen in Figure 4.9. The most important relation is the ratio between "Families Inspected" and "Detected on Injection", because it shows how many signatures were successfully derived, for each family inspected. However, uninspected families consisted mostly of hash-based signatures in the case of ClamAV (cf. Section 4.1). "Injectable without Flaw" represents signatures, which can be used to mark arbitrary executable files as malicious for the corresponding scanner.
CHAPTER 4. EVALUATION

Figure 4.9: Injecting derived signatures.

Signatures, that are not relying on the PE header may be injected into arbitrary files without regard to the file format.

As a next step, the relation between correct derivations and amount of samples for a virus family is evaluated. All signatures that were not added to the derived signature database or were not detected on injection as seen in Figure 4.9 count as not derived correctly.

Figure 4.10: Relation for ClamAV signatures.
Figure 4.11: Relation for AV1 signatures.

The relations for ClamAV and AV1 – AV3 are illustrated in Figures 4.10 to 4.13. The rate of success increases with the increase of samples for a virus family. A very high amount of classifications for one virus family points to a use of a heursitic. For ClamAV the heuristic ended up in 172 different signatures with same length each time and was therefore not correctly derivable. For AV3 the highest number of samples for a single virus family was 1201. Many samples with only one sample were not derivable either.

Obviously, a smaller number of samples for one virus family potentially leads to a higher failure rate. On the contrary, a very high number potentially points to a heuristic, which is possibly not derivable.
4.4. INJECTION OF THE DERIVED SIGNATURE INTO A FILE

4.4.2 Cross-AV Detection

The previously modified files of Section 4.4.1 are now scanned with other anti-virus scanners while using the smallest possible gap size except for gaps of arbitrary length. In such case 5000 bytes will be inserted. This cross-AV detection is done to see how many AV signatures are possibly used equally in other AVs. If a signature is shared amongst multiple scanners, it does not have to be the same signature and can be a subordinate signature as seen in Example 10.

Example 10 (Intersecting Signature)

AV1 uses the signature

\[ s_1 = 4d\ 5a\ ff\ ac\ \{1-4\}\ ac \]

to detect malicious files. AV2 uses

\[ s_2 = ac\ \{1-3\}\ ac \]

to detect malicious files. Therefore, every file that is classified as malicious by AV1 with \( s_1 \) automatically is detected by AV2 as well when using a gap size of one. If both signatures are equal, AV1 will detect the file with the injected signature \( s_2 \) as well. If this is not the case, one signature is a subordinate signature and therefore less specific. However, varying gap sizes may occur. In this case, \( s_2 \) is a subordinate signature of \( s_1 \) only in some cases (as seen in this example).

As seen in Table 4.2 there are some intersecting signatures. If signatures are detected by another scanner but not vice versa, signatures of one scanner might be more specific or overfitted than the signatures of the other one.
CHAPTER 4. EVALUATION

<table>
<thead>
<tr>
<th>detected by</th>
<th>ClamAV</th>
<th>AV1</th>
<th>AV2</th>
<th>AV3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClamAV DB</td>
<td>48</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>AV1 DB</td>
<td>1</td>
<td>25</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>AV2 DB</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>AV3 DB</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 4.2: Cross-AV detection result.

4.5 Runtime

The runtime is evaluated for each stage of the derivation. There might be some outliers due to the fact, that ClamAV for example sometimes reloads the signature database. In this time, the scanning process is paused. Therefore, the median of all times is calculated. Additionally, the mean is specified for completeness.

<table>
<thead>
<tr>
<th></th>
<th>total</th>
<th>relevant bytes</th>
<th>relevant nibbles</th>
<th>ref. point relevance</th>
<th>gap sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>median</td>
<td>122.3s</td>
<td>39.1s (23.7%)</td>
<td>10.1s (6.1%)</td>
<td>3.5s (2.1%)</td>
<td>69.6s (42.1%)</td>
</tr>
<tr>
<td>mean</td>
<td>180.1s</td>
<td>49.2s (27.3%)</td>
<td>28.3s (15.7%)</td>
<td>4.1s (2.3%)</td>
<td>98.5s (54.7%)</td>
</tr>
</tbody>
</table>

Table 4.3: Runtimes per sample for ClamAV including gap sizes.

Keep in mind, that signatures without wildcard gaps exist. Still, the binary search needs a lot of time for all samples because of the high amount of initiated scans. Because of this, the calculation of the gap size is skipped for other scanners as it only consumes time and is not necessary for current evaluations.

<table>
<thead>
<tr>
<th></th>
<th>total</th>
<th>relevant bytes</th>
<th>relevant nibbles</th>
<th>ref. point relevance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClamAV median</td>
<td>52.7s</td>
<td>39.1s (74.2%)</td>
<td>10.1s (19.2%)</td>
<td>3.5s (6.6%)</td>
</tr>
<tr>
<td>ClamAV mean</td>
<td>81.6s</td>
<td>49.2s (60.3%)</td>
<td>28.3s (34.7%)</td>
<td>4.1s (5.0%)</td>
</tr>
<tr>
<td>AV1 median</td>
<td>109.7s</td>
<td>88.9s (81.0%)</td>
<td>9.3s (8.5%)</td>
<td>11.5s (10.5%)</td>
</tr>
<tr>
<td>AV1 mean</td>
<td>126.6s</td>
<td>97.9s (77.3%)</td>
<td>16.1s (12.7%)</td>
<td>12.6s (10%)</td>
</tr>
<tr>
<td>AV2 median</td>
<td>89.1s</td>
<td>60.3s (67.7%)</td>
<td>6.4s (7.2%)</td>
<td>22.4s (25.1%)</td>
</tr>
<tr>
<td>AV2 mean</td>
<td>123.3s</td>
<td>86.9s (70.5%)</td>
<td>12.5s (10.1%)</td>
<td>23.9s (19.4%)</td>
</tr>
<tr>
<td>AV3 median</td>
<td>394.9s</td>
<td>258.2s (65.4%)</td>
<td>34.2s (8.7%)</td>
<td>102.5s (26%)</td>
</tr>
<tr>
<td>AV3 mean</td>
<td>397.8s</td>
<td>258.9s (65.1%)</td>
<td>38.7s (9.7%)</td>
<td>100.2s (25.2%)</td>
</tr>
</tbody>
</table>

Table 4.4: Runtimes per sample for all scanners.
The four different AV scanners were tested on two different environments. ClamAV and AV1 were derived on a notebook with less performance than the computer, on which AV2 and AV3 were derived. Therefore, percentages of each passage are more useful for comparisons between scanners. The total amount of time needed for a sample is related to the amount of gaps as shown in Table 4.3. Approximately 50% of all time was spent on this step. If this calculation is skipped, the runtime is related to the amount of relevant bytes as seen in Table 4.4. Around 70% of all time was used in this step. All files that were not inspected in more detail due to exceeding the time limit are potentially stored as hashes as there is a high amount of relevant bytes.
Chapter 5

Limitations

The results presented in Section 4.3 are summarized to show the limits of this automatic derivation process. While calculating the Levenshtein Distance between derived signatures and the ClamAV signature DB, the main sources of mistakes and limitations became clear and are described in Section 5.1 to 5.3. Another possible limitation regarding dependencies within the file is presented in Section 5.4.

5.1 Additional Wildcard Gaps

The first source of deviation comes into existence due to gaps, that are not existent in the sample file. This can be seen in Figure 5.1. A sample file, that will lead to such an error is shown in Example 11.

\[
\text{Derived: } 8a440500 \ 3007 \ 80e901*5e4e*0f85 \\
\text{ClamAV : } 8a440500*3007*80e901*5e4e*0f85
\]

Figure 5.1: Additional wildcard gaps.

Example 11 (Underrived wildcard gaps)

Let \( s_{\text{correct}} = AB \) be our signature. We apply our derivation on the sample \( f = \text{CABB} \). At the end of our calculation we have derived \( s_{\text{derived}} = AB \).

Obviously, an additional byte can be inserted at every position to find additional gaps, which are not existent in the inspected sample. This however consumes significantly more time and is left for future work.
5.2 Boolean Operations within the Signature

Another source of errors is caused by boolean-based signatures. As shown in Figure 5.2, the union of the two bytes extends the amount of possible signatures for one virus family. In this case, 40 different signatures are covered in one family. This leads to a possible error, when replacing the related byte with for example another byte, that is included in the union. If the byte is replaced in such way, the derivation will assume a wildcard byte at this position. The presented derivation procedure is therefore not able to find all signatures. However, this mistake can be avoided if enough time and disk space are available. Each byte can be replaced with any other byte and only if all $2^8$ possible samples are classified as malicious, the byte is a wildcard. This may be applied to wildcard gaps as well. Wildcard gaps are assumed to lie in an interval, but more complex wildcard gaps might be used, for example:

\[
4d \ 5a \ fc \ (-5|10-20) \ fc
\]

which would allow zero to five or ten to twenty arbitrary bytes. If used, the binary search would not be applicable anymore and more time consuming methods have to be used to determine the exact size of the gap.

5.3 Multiple Occurrences of Substrings

As a third limitation a sample file may include multiple equal substrings. If such a substring and sufficient gaps are included in the signature a derivation is not possible. If the first substring is replaced, the scanner is able to match the virus with the second string and vice versa. Therefore, the virus is detected each time and it is not possible, to find all relevant bytes. Figure 5.3 shows multiple occurrences of the same substring within the signature and Example 12 shows a simple illustration, how this error occurs.

Derived: 000000005b636f6e666967446174615d  
ClamAV : 000000005b636f6e666967446174615d*000000005b636f6e666967446174615d

Figure 5.3: Same content before and after a gap.
5.4. FILEFORMAT DEPENDENCIES

Example 12 (Bad sample file)
Let $s_{correct} = A \cdot B$ be our signature. We apply our derivation on the sample $f = CABBC$. At the end of our calculation we have derived $s_{derived} = A$. While deriving, specific bytes are replaced. In this case, it does not matter which $B$ of our sample is replaced, the signature will still match. This leads to an assumption, that both $B$s of our sample are wildcards in the signature, which is not the case.

The amount of possible multiple occurrences will grow exponentially with increasing file size as the number of substrings increases exponential as well. When increasing the file size by only one byte, each substring is able to create a new substring using this byte. Finding all such occurrences and replacing them at the same time takes more time and is left for future work as well.

5.4 Fileformat Dependencies

Anti-virus scanners may analyse the structure of the file as well. For example, a malicious HTML file might only be considered malicious, if the structure is all right. For example, if the derivation process replaces a closing bracket ">" with an opening one "<", the file may be not conforming standards and therefore remain unclassified.
AV1 classifies executables with a flaw in the PE-Header as "damaged". While replacing bytes, flaws like this could occur and more relevant bytes will be assumed, as the initial virus is not detected anymore.
Going hand in hand with this, an overfitting is possible, because more relevant bytes are assumed than necessary. A derived signature could work at the entry point, but cause a "damaging" of the file at certain other places. Then the derivation will assume, that this signature is only matched at the entry point, but it could be, that the signature is also tried to be matched at all other positions.
CHAPTER 5. LIMITATIONS
Chapter 6

Conclusion

The evaluation of the presented derivation approaches show, that it is possible to automatically derive signatures of AV scanners. Signatures for ClamAV were derivable very well. Around 63% of all inspected virus family showed a successful signature derivation. The approach worked for AV2 as good as for ClamAV. In this case, 57% are derived effectively. AV1 reached a derivation ratio of 48%. In contrast to them, AV3 reached only 18%. This value highly depends on the given set of samples but as the used set of this thesis is up to date, it is a very good measure. A sample to a certain virus family has to exist in order to be able to derive a signature for it. If there are more samples of a virus family, a better signature quality can be achieved in most cases as seen in Section 4.4. However, in some cases it is not possible to derive a signature due to the structure of the signature. In such cases even a great amount of samples will not be able to guarantee a better signature quality. Due to the fact, that byte-based signatures only represent a small fraction of a file it is possible to derive signatures for them without great runtimes. However, hash-based signatures contain many relevant bytes and are therefore not easily derivable. If enough time is available, they are even better derivable than byte-based signatures, as no wildcards may be included. Thereby, limitations mentioned in Chapter 5 are invalid for those. Heuristics can even help to find the offset and stop the derivation early. Additionally, it is possible, to find more than one signature from a single sample. This can be accomplished, by having a sample which will lead to another virus family, if certain parts are replaced. A sample can include more than one signature or the included signature is a subsignature of another one.

All in all we can conclude, that a derivation of the used signature for a sample is possible in many cases as limitations exist. Approaches to a better signature quality are shown and left for future work. Heuristics represent the core of undervivable signatures, as they are not specified as a single signature. Additionally, some enhanced classification methods are potentially used in AV3 that prevented a successful derivation as many different signatures are found for separate virus families.
Bibliography


