YARIX: Scalable YARA-based Malware Intelligence

Michael Brengel
michael.brengel@cispa.saarland
CISPA Helmholtz Center for Information Security

Christian Rossow
rossow@cispa.de
CISPA Helmholtz Center for Information Security

Abstract

YARA is the industry standard to search for patterns in malware data sets. Malware analysts heavily rely on YARA rules to identify specific threats, e.g., by scanning unknown malware samples for patterns that are characteristic for a certain malware strain. While YARA is tremendously useful to inspect individual files, its run time grows linearly with the number of input files, resulting in prohibitive performance penalties in large malware corpora.

We present YARIX, a methodology to efficiently reveal files matching arbitrary YARA rules. In order to scale to large malware corpora, YARIX uses an inverted n-gram index that maps fixed-length byte sequences to lists of files in which they appear. To efficiently query such corpora, YARIX optimizes YARA searches by transforming YARA rules into index lookups to obtain a set of candidate files that potentially match the rule. Given the storage demands that arise when indexing binary files, YARIX compresses the disk footprint with variable byte delta encoding, abstracts from file offsets, and leverages a novel grouping-based compression methodology. This completeness-preserving approximation will then be scanned using YARA to get the actual set of matching files.

Using 32M malware samples and 1 404 YARA rules, we show that YARIX scales in both disk footprint and search performance. The index requires just ≈74% of the space required for storing the malware samples. Querying YARIX with a YARA rule in our test setup is five orders of magnitude faster than using standard sequential YARA scans.

1 Introduction

As a core part of their threat intelligence, the security industry closely monitors both known and new malware samples. To this end, anti-virus and threat intelligence companies heavily leverage their long-term malware archives to inspect malware threats. These gigantic malware databases quickly span hundreds of millions or billions of samples [3]. The security industry uses these archives to search for patterns of known malware *variants*. While not following a strict definition, a malware *family* usually groups together malware files that follow the same semantics, e.g., because they share the same code basis. By monitoring these variants, we learn when a certain threat has been active, what it aimed for, and ultimately, indications that manifest the responsible actors. Obtaining a complete malware picture is fundamentally important to create accurate threat intelligence reports, and gives valuable insights into both consumer malware and Advanced Persistent Threats (APT) [10, 15, 21].

The inherent challenge in this process is to classify malware samples into variants. To close this gap, YARA [2] has matured to the community and industry de facto standard to express patterns that are characteristic for a malware variant. YARA rules capture structural and semantic patterns of a malware variant. Such rules determine if a malware sample matches the known variant, and hence, are a vital driver for automated malware inspection. YARA signatures help to identify and classify malware samples based on arbitrary binary patterns specified in a YARA-specific syntax. Analysts create YARA rules as part of their exploratory threat intelligence and apply them to sample archives. YARA signature repositories, both free [33, 40] and commercial, are frequently updated to keep up with newest threats. To keep up with the rapidly growing malware ecosystem, analysts also use signature generators [8, 12] to create these rules. Consequently, a diverse and ever-changing set of YARA rules has to be (re-)applied to large and continuously growing sample databases.

Unfortunately, the growing number of new malware samples (e.g., $\approx 10^5$ of new malware samples daily [3]) is a great challenge to YARA users. While YARA has a reasonable runtime performance on small data sets, it does not scale to larger data sets. For example, applying a standard YARA signature on 32M malware samples takes multiple days, and this runtime increases linearly with the number of samples. While parallelism mitigates the problem to a certain extent, it does not solve the need to scan each and every sample regardless of the YARA rule. This is stark contrast to the need of malware analysts, who want to *efficiently* scan malware samples using

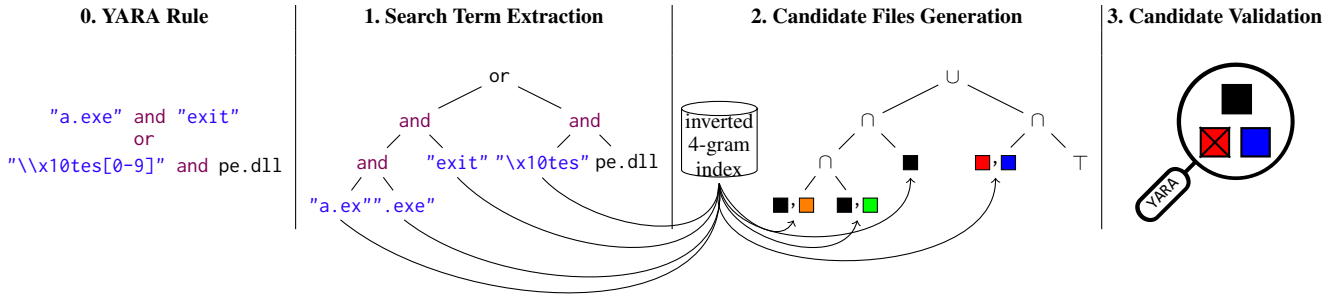

Figure 1: YARIX queries an inverted malware file index with search terms extracted from a YARA rule.

arbitrary YARA rules. Analysts regularly need to (i) adapt and re-evaluate YARA signatures on large malware data sets, (ii) efficiently search for known rules in new malware corpora, or (iii) apply and fine-tune ad hoc YARA signatures, e.g., to quickly reveal the existence of emerging threats. They thus have to scan for patterns that are not known *a priori*, showing the need for an *efficient yet generic* search methodology.

In this paper, we propose YARIX, the first generic YARA search engine that is both scalable and space efficient. YARIX finds files that match a YARA rule several orders of magnitude faster than off-the-shelf YARA scans. This performance optimization is achieved by transforming YARA rules into search terms that can be efficiently searched for using an inverted n-gram index. YARIX parses the YARA rule and extracts all the n-grams (n consecutive bytes in a search string) contained in the strings of the rule. YARIX then uses the index to enable for efficient and sub-linear searches for these terms.

The index is independent from the YARA rules. That is, the index supports arbitrary new YARA rules without requiring an update. Index updates are only required when adding new malware files to the corpus—a one-time effort per indexed file. YARIX is *fully compatible* to YARA and does *not* require modifications to the YARA standard or rules. It significantly optimizes the YARA search and can handle advanced features such as regular expressions and library import specifications.

Furthermore, YARIX returns *sound and complete* results. Our exact YARA rule transformation guarantees that queries return complete results, i.e., all files that match the search criteria. To retain soundness, i.e., to reliably rule out false positives, as a final step, we scan the set of candidate files returned by querying the inverted index—a superset of all actually matching files—with YARA signatures. YARIX thus reliably replaces traditional YARA scans with an efficient search strategy, which significantly reduces the set of malware samples that have to be scanned.

YARIX's back-end, the inverted file index, extends the general idea of an inverted n-gram index [4]. The index consists of $2^{8 \cdot n}$ *posting lists*, i.e., sets of malware sample IDs that contain a given n-gram. To reduce the disk footprint of the index, we employ (i) variable-length encoding, (ii) compress posting lists using delta encoding, and (iii) propose an over-approximating grouping-based compression methodology.

We evaluate YARIX by building an index with 4-grams over 32M malware samples. Our compression methods reduce the index disk footprint effectively. While a naive inverted index would have a high overhead (e.g., 400% for 2^{32} indexed files, due to 4B-wide file IDs) compared to the sample size, variable length and delta encoding shrink this overhead to 149.5%. Grouping reduces this even further to about 74% overhead, showing a significant gain over standard compression. At the same time, YARIX reduces the runtime to search for YARA signatures significantly. To assess the search performance, we process 1 404 publicly available YARA rules [33] with YARIX. On average, YARIXis five orders of magnitude faster than full sequential YARA search.

To summarize, our contributions are as follows:

- We present YARIX, a fully YARA compatible search engine that uses an inverted file index to efficiently reveal all samples matching a given YARA rule.
- We provide a fully-automated methodology that transforms off-the-shelf YARA signatures into search terms that can be used to efficiently query the index, resulting in *sound and complete* results.
- We evaluate the effectiveness of space compression algorithms and are able to shrink the index's disk footprint to less than the size of the malware samples being indexed.
- We evaluate our prototype based on 32M malware samples and 1 404 YARA rules.[1]. YARIX reduces the search time by five orders of magnitude.

2 Extracting YARA Search Terms

YARIX receives a YARA rule as input and utilizes a preprocessed inverted (malware) file index to efficiently search for all indexed files matching this rule. At the same time, YARIX

[1]The YARIX reference implementation can be obtained at https://github.com/mbrengel/yarix

```
rule foo{
  strings:
   $str_a = "calc.exe"
   $str_b = "IsDebuggerPresent"
   $str2_a = /[\/a-zA-Z0-9\.]{0,64}\.png/
   $str2_b = /\xC7\x45\xC3\x41[A-Za-z-_\/\\ ]/
   $op_a = {01 01 01 01 ?? 5? [2-8] C1 (E?|F?) 02}
   $op_b = {F3 AB 88 12 83 (E?|F?) 03 F3 AA}

  condition:
   $op_b
   or(2 of $str_* and 1 of $str2_* and 1 of $op_*)
}
```

Figure 2: YARA Rule Example.

retains the strong capabilities of the feature-rich YARA language. To this end, we automatically extract *n*-grams from the strings provided by the YARA rule and use the index according to the logic provided by the rule to find a set of candidate files that will then be scanned using YARA. This optimization is done in a completeness-preserving manner, i.e., we do not miss files that would be found by standard YARA. At the same time, we want to extract as much prefilter information as possible to minimize the set of candidate files that will be scanned with YARA to improve performance. This optimization leads to large speedup factor, as the number of files that need to be scanned by YARA can be reduced by multiple orders of magnitude.

Figure 1 shows the general system overview of YARIX using an exemplary YARA rule. In the first step, YARIX parses the rule to extract the condition expression and its 4-grams (substrings of size four) of all strings. In the second step, these 4-grams are searched for in the inverted 4-gram index of YARIX. For instance, `"a.exe" and "exit"` is processed by feeding the 4-grams `"a.ex"` (found in file ■ and ■), `".exe"` (found in ■ and ■) and `"exit"` (found in ■) to the index. The regular expression `"\x10tes[0-9]"` is—without losing completeness—simplified to `"\x10tes"` (found in ■ and ■). Some YARA features, such as the expression `pe.dll`, are not indexed and thus cannot be further optimized ($\top$, i.e., found in all files)—again, without loosing completeness. The resulting sets of files are intersected according to the logic dictated by the rule ($\cup$ for `or` and $\cap$ for `and`), which yields the candidate files ■, ■ and ■. In the third and final step, these candidate files are validated with the standard YARA tool which eliminates file ■ and returns files ■ and ■ as the final results.

In the following, we will describe the details of this optimization process and outline how YARIX supports all YARA features. We will start with a short introduction into YARA rules.

2.1 YARA Rules Overview

The YARA framework [2] features complex and feature-rich file searches. To this end, YARA defines its own pattern matching language that relies on common constructs such as strings, regular expressions and context-sensitive grammars. In addition, YARA features semantic capabilities, such as checking the list of exported functions of a Portable Executable file. Optimizing all these features is an impossible endeavor, as they partially require heavy file parsing (e.g., imported/exported functions) or ad hoc computations (e.g., checksums over parts of the file). Having said this, YARIX optimizes the most common YARA language constructs, and supports the remaining features by considering the full set of indexed files without filtering. For a detailed list of the individual YARA language features and module extensions that YARIX leverages for its optimizations, we refer to the technical documentation of YARIX[2].

Figure 2 shows an example of a YARA rule named `foo`. YARA rules consist of strings and a condition that dictates the searching logic. The three basic types of strings in YARA are normal plain strings (`$str_a` and `$str_b`), regular expressions (`$str2_a` and `$str2_b`) and hex strings (`$op_a` and `$op_b`). Hex strings support wildcard and grouping mechanisms. For example, `??` matches any byte, whereas `5?` matches any byte where the upper nibble is 5. The wildcard expression `[2-8]` matches an arbitrary sequence of at least 2 and at most 8 bytes. The group expression `(E?|F?)` matches any byte where the upper nibble is either 14 or 15.

The *condition* of the rule governs the matching logic. Conditions are expressions of boolean type and support standard logic operators such as `and`, `or` and `not`. The most simple boolean expression is a string identifier. For example, `$op_b` in Figure 2 evaluates to `true` if the string pointed to by the identifier matches. As seen in the example, conditions also support `x of strs` expressions, where `x` is a number and `strs` is a set of strings. The notion `x of y` means that at least `x` of the strings contained in `y` should match. The set of strings inside such an expression can also be expressed using a wildcard expression such as `$str2_*`. Our example rule triggers if `$op_b` matches, or if both plain strings, at least one regular expression and at least one hex string match.

2.2 Processing Strings

In order for YARIX to work with an arbitrary YARA rule, we must be able to automatically process all types of strings and all types of condition expressions. That is, given a string of a YARA rule, we have to feed it in some form to the index to find all files for which that string would be a potential match. As previously stated there are three types of strings, each of which we handle separately.

[2] https://github.com/mbrengel/yarix

Plain Strings This is the easiest type of string, as it can be broken up into its n-grams which can then be used to query the index. Formally, let s be a plain string of length l consisting of the bytes $b_1, \ldots, b_l$. Then we use each n-gram $x_i \in \{b_j b_{j+1} \ldots b_{j+n-1} \mid 1 \leq j \leq l-n+1\}$ and query the index to get a set of file IDs F_i in which x_i is contained. Finally, the intersection $C = \bigcap_{x_i} F_i$ is returned as a set of candidates that potentially match the plain string s.

Regular Expressions To handle a regular expression, we identify the plain strings that will be contained in every string that will be matched by the regular expression and then proceed as with plain strings. In detail, given a regular expression r, we first construct a DFA from it. Then, we compute the dominators of the final state of the DFA, i.e., all states that any accepting word will visit when the DFA is executed. For each of these dominator states we then proceed as follows: We check if the state has only one outgoing edge. If this is the case, we collect the label (the character) of the edge and continue with the target state of that edge. This is repeated until we reach the final state or a state with more than one outgoing edge is discovered. The concatenation of the collected characters along the path form a plain string that will be contained in every string that matches r. This process will give us a set of plain strings S that we can proceed with as before to get a set of file IDs F_i for each $s_i \in S$. Given that all of these plain strings must be contained in every match of r we can return $C = \bigcap F_i$ as the final set of candidates. In Figure 2, the plain strings of `$str2_a` and `$str2_b` are `".png"` and `"\xC7\x45\xC3A"`, respectively.

Hex Strings Similar to before, we handle hex strings by identifying plain strings. In detail, we start at the beginning of the hex string and scan byte by byte to collect a plain string. We stop collecting the current plain string whenever we encounter a wildcard or a grouping expression and start collecting a new plain string when we encounter the next fixed byte. In Figure 2, the sets of plain strings of `$op_a` and `$op_b` are `"\x01\x01\x01\x01"`, `"\xF3\xAB\x88\x12"` and `"\xAB\x88\x12\x83"`.

We may fail to extract *any* n-gram. For example, the regular expression `"[0-9]+"` does not contain any plain strings that will be contained in every match. Similarly, a hex string might consist only of wildcards and/or grouping expressions such as `{ (A?|B?) (C?|D?) }`. Because of performance reasons, we also neglect strings smaller than n bytes, where n is the size of the n-grams. In principle, we could query the index for all n-grams where the search string is a prefix or a suffix. However, this quickly imposes a significant overhead even if we are just a single byte short. For example, if we have $n = 4$ and we want to create the posting list of the 3-gram `"abc"` we would need to create the union of all posting lists of `"`x`abc"` and `"abc`x`"` where x is an arbitrary byte, which would be a slowdown factor of $2 \cdot 256 = 512$ and involve more costly set operations.

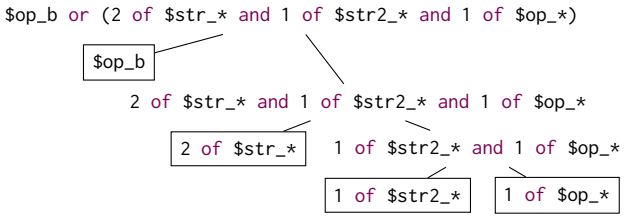

Figure 3: Abstract Syntax Tree (AST) of the condition of the YARA rule in Figure 2.

This also applies to the plain strings yielded by processing regular expressions and hex strings, i.e., a regular expression is not optimizable if all of its plain strings are smaller than n bytes. YARA also supports the `nocase` modifier for case-insensitive searches, which we optimize by numerating all 2^n different options for each n-gram. If we cannot find any long enough plain strings, we consider the whole string not optimizable and return $C = \top$, i.e., the whole universe of files.

2.3 Processing the Condition

We need to parse the condition of a rule to understand how we should combine the search results of the individual strings. Figure 3 shows how we create the abstract syntax tree (AST) of the condition of our example rule in Figure 2. The leaf nodes are expressions of boolean type that do not contain any of the standard logical operations (`and`/`or`/`not`). For each of these expressions, we define an index search operation that captures the semantics of the expression. After that, we combine the search results according to the logic operation in the tree, i.e., set union ($\cup$) for `or` and set intersection ($\cap$) for `and`[3]. In case of a logical negation (`not`), we check if the expression only contains plain strings which are exactly n bytes long. In this case, we compute the file IDs, apply a set minus and proceed upwards in the tree. Otherwise, the expression is considered unoptimizable and $\top$ is returned. For example, consider the expression `not $s` where `$s = "abcde"`. Using an index for $n = 4$, we can identify posting lists for `"abcd"` and `"bcde"`. However, any combination of these posting lists only gives us a set of candidate files that might contain `"abcde"`. A negation of such a set is not the set of files that do definitively not contain `"abcde"`. For example, it is perfectly valid for all files in the intersection of both posting lists to contain `"abcde"`.

2.4 Processing Individual Expressions

We need to handle the individual expressions, i.e., the leaf nodes of the AST. Handling a simple string expression such as `$op_b` is obvious as we just use the index as described in

[3] We apply the standard set operation simplifications in case we have to deal with $\top$, i.e., $\top \cup A = \top$ and $\top \cap A = A$

Section 2.2. The `x of strs` expression can be captured with the index as follows: for each string contained in `strs` we query the index to find all file IDs that match the string. For each file ID we count how many strings match, which we use to return all file IDs that match at least `n` strings. In the case that searching some of the strings of `str` is not optimizable for reasons described in Section 2.2, we first create the expression `x' of strs'` where `strs'` is the optimizable subset of `str` and `x'` is `x` minus the number of unoptimizable strings. If `x'` is smaller or equal to 0, then we consider the expression not optimizable and return $\top$. Otherwise, we proceed as before, i.e., index search and keeping track of the number of matches. For any other type of expression e, we simplify the expression to a `x of strs` expression. In detail, if `s1`, ..., `sl` are the string identifiers that an expression e contains, we create the simplified expression $e' =$ `1 of (s1, ..., sl)` and proceed as before, i.e., we want to match all strings contained in e.

2.5 Optimization Limitations

The simplification we apply to expressions to ensure that we can optimize them, comes with a loss of precision. For example, an expression of the form `str at o` requires positional information that we cannot capture. By simplifying this expression to `1 of str` we do not lose completeness, as any file where `str` appears at `o` must fulfill the precondition that `str` appears at all. However, we accept a loss in semantic information that could potentially blow up the set of candidate files. While this is not a problem in terms of soundness of the search as we will use YARA as a post-tool to filter out non-matching files, it could still lead to performance issues.

Another problem that could occur is optimizability. We consider a rule not optimizable if evaluating it returns $\top$, which happens if there are too many unoptimizable expressions and this problem propagates through the set logic dictated by the condition. In this case, YARIX is equivalent to a sequential YARA scan. We have seen that expressions cannot be optimized if they contain too many unoptimizable strings. Apart from the examples we discussed, i.e., strings being too short, there are also expressions that we consider unoptimizable because they do not use any strings at all. For example, YARA contains the PE module which allows for complex expressions such as `pe.imphash() == "..."` or `pe.rva_to_offset(0x40000) == ...`. Such expressions reflect strong semantic constraints that we cannot optimize with YARIX. We thus have to ignore such expressions and consider them unoptimizable. In case a rule contains too many of such expressions, the whole rule becomes unoptimizable and a full sequential traditional YARA scans is required. However, the PE module also contains features that YARIX can cope with. For example, the expression `pe.checksum == "\xDE\xAD\xBE\xEF"` puts constrains on the checksum field of the PE header. We can abstract from this by simplifying this to `1 of "\xDE\xAD\xBE\xEF"`, i.e., we only require the string `"\xDE\xAD\xBE\xEF"` to be present at all, which preserves completeness.

It is worth noting that a rule can be optimizable even if it contains unoptimizable strings. Consider, for example, a rule `e1 and e2`. Even if `e1` cannot be optimized, the whole rule can still be optimized if `e2` is optimizable. In this case the logical `and` operator translates to a set intersection and remedies the fact that `e1` translates to the set of all file IDs.

3 File Index Design

We now describe the design and implementation of the binary file index used by YARIX. We start our discussion by showing the commonalities and differences between traditional indexes used for full-text search and our setting. After that, we discuss the decisions we take to compress the index to reduce its disk space footprint.

3.1 Background

YARIX requires a generic index design, i.e., supports arbitrary YARA rules without requiring updates upon rule changes and/or additions. The underlying data structure of YARIX is thus an inverted n-gram index [4]. We borrow this idea from the domain of full-text search. There, an inverted index maps n-grams of tokens, i.e., n consecutive words or characters in a document, to sets of IDs of documents containing these tokens – so called *posting lists*. This mapping allows to find all documents that contain a given n-gram. Moreover, a posting list of an n-gram usually stores the position(s) at which the n-gram occurs for each document ID of the list. This allows to find the exact position of the n-gram within the file.

To search for a sentence in all documents, one would break down the sentence into its n-grams and look up the document identifiers in the posting lists of those n-grams. For example, assume we search for the phrase `The five boxing wizards` with $n = 3$. Here, we first use a sliding window to split the phrase into the two possible 3-grams, namely `(The, five, boxing)` and `(five, boxing, wizards)`. We then look up the posting lists for each of the four n-grams. These lists are then intersected to get a list of candidate documents that *potentially* contain the desired phrase. To verify if the candidate documents *actually* contain the entire phrase, we can verify the positional information stored in the posting lists. If the n-grams' offsets of a candidate document are in sequential order, we can be certain that the document contains the word or sentence; if they are not, the phrase is not contained. With reference to our example, assume a file that contains the sentence `The five boxing frogs are similar to five boxing wizards`. While this candidate document indeed contains all searched n-grams, their positions within the document (1, 8) are not consecutive, and hence, the search results excludes it.

3.2 Inverted n-Gram Malware Index

The general idea of an inverted index seems to translate nicely to this use case. Here, the documents are malware samples, and an n-gram represents a sequence of n consecutive bytes within a file. There are, however, a few notable challenges that we have to tackle to apply the idea of an inverted n-gram index to malware samples.

First, the number of possible byte sequences in a binary file quickly explodes for larger n. For example, choosing $n = 4$ already yields a set of $2^{4 \cdot 8}$ potential posting lists that need to be maintained. The number of words or characters in the case of full-text document search is orders of magnitudes smaller than in our case. We have found that this space is quickly saturated by approximately 10^5 malware samples, i.e., every possible 4-gram occurs in at least one of those samples. This is different in a text setting, where the case-insensitive character set is tightly constrained. Moreover, the language grammar dictates strong relationships between particular words (e.g., (1) article, (2) adjective, (3) noun), resulting in an overall smaller number of actual combinations.

Second, malware executables lack a natural word delimiter. While texts contain whitespaces that can be used to infer tokens, we cannot infer any meaningful boundary in malware samples that contain a mostly unstructured blob of arbitrary code and data. Due to the lack of reliable tokenization of malware samples, we thus have to fall back to fixed-size byte sequences within the document.

Both observations impose interesting challenges for a space-efficient malware index. We aim to index large collections of potentially billions of files, which quickly leads to formidable space requirements. It becomes particularly challenging as we aim for a *complete* search, i.e., search results must not dismiss any documents that match the search criteria in favor of efficiency. In the following, we will thus present and discuss methodologies that compensate the lack of space-efficiency, while preserving completeness.

3.3 Space Optimization Strategies

The size of an inverted index is largely determined by two factors: (a) the number of n-grams in the index, and (b) the size of the posting lists of each n-gram. Both represent suitable angles to heavily reduce the space required to store an index. In the following, we will survey the general options to reduce storage costs for either angle.

3.3.1 Optimizing the Set of Considered n-Grams

An obvious first optimization point is to set n to a small value. For $n = 1$, there are at most $2^8 = 256$ n-grams, and for each increment, the number of n-grams multiplies by eight. Choosing an efficient n that is still characteristic enough for searches is a trade-off. While smaller n clearly reduce the number of posting lists, shorter n-grams are less characteristic and have a higher chance to be present in a large fraction of indexed files. We defer this discussion to Section 4.5, in which we evaluate and choose an appropriate n.

If we knew the search criteria when building the index, n-grams that are never searched for could be ignored. This would represent a significant reduction of index space. However, for this optimization to work, we need *a priori* knowledge of the search criteria, and the criteria must be static over time. One can quickly see that this is not a fair assumption in most settings. Malware analysts regularly define new search criteria ad hoc to explore malware files as part of their threat analysis. Any new n-gram not covered by the index yet requires a costly rebuild of the index. Consequently, in this work, we do not assume such *a priori* knowledge of search criteria, which allows to apply arbitrary searches.

Another optimization strategy would be to ignore n-grams that do not serve as discriminative part of any search criteria. In the setting of text files, one could ignore words that frequently occur (like articles), and likewise ignore them during search. In our setting, lacking knowledge of the search criteria, we could stop maintaining posting lists of those n-grams that are shared by "too many" files in the index. This follows the intuition that such n-grams would not help to distinguish between malware families, each of which makes up only a smaller portion of the overall index. However, a general risk of this strategy is that the index can no longer used as filter for large classes of files in the index. For example, in principle, the n-gram of the Windows PE header allows to search for all Windows executables. Yet, given that it is shared by "too many" files, it will not be part of the index. Furthermore, particular malware families may be overrepresented in malware collections [34], e.g., due to polymorphism. Neglecting popular n-grams would thus render it infeasible to search for those families. Given that we want YARIX to be generic, complete, and compatible to large search results, we do not further follow this strategy.

3.3.2 Optimizing the Posting Lists

A completely orthogonal approach to shrinking the set of n-grams is to reduce the size of the posting lists. For the reasons mentioned before, for YARIX, we follow only optimization strategies of this kind. In particular, YARIX (i) operates without storing offset, (ii) deploys optimal delta encoding to represent posting lists, and (iii) groups files to shorten the identifier space that has to be stored. All of these methods do not break our completeness guarantees, i.e., they retain that all files matching a certain criteria will be returned.

3.4 Offset-Free Index

As a first measure to shrink the posting lists, we remove any positional information from them so that they only contain the file IDs. Typically, posting lists contain the file ID where an

n-gram was found, plus its offset within the file. Neglecting offsets saves a large amount of data. Apart from not storing the offsets themselves, we also save space as we do not count n-grams occurring multiple times. For instance, if a malware sample f contains the 4-gram `00 00 00 00` 1 000 times, a traditional inverted n-gram index would store 1 000 (f, o_i) pairs in the posting list of `00 00 00 00` where the o_i values are the offsets at which the 4-gram occurs in f. In contrast, YARIX will only store f once in the posting list. The negative aspect of ignoring offsets is that it complicates the search process. After intersecting the posting lists, a position-aware index ensures that the offsets contained in the posting lists are in sequential order to eliminate any wrong candidates among the set of all possible candidates. Given that this is not possible anymore if we remove offsets from the posting lists, we solve this is by performing a normal sequential search on the candidates. However, this is not a problem in our context as the YARA expression optimizations that we described in Section 2 require us to do a sequential YARA scan anyway. We will show in Section 4 that acceptable real-world performance is maintained with this solution.

3.5 Variable Delta Encoding

We aim to create an index for up to 2^{32} malware files. Such a bound nicely sets an upper limit for the size of one ID. In particular, we have the guarantee that a file ID always fit into 32 bits. Note this restriction is not a limitation. In fact, if the index is saturated, we can create further indexes and search through all indexes combined.

Despite the upper bound for file IDs, storing 32 bits per entry is wasteful. In particular, when using sorted posting lists, we can store ID differences instead of their absolute values. In well-populated lists, such deltas would be significantly smaller than 32 bits. That is, in a sorted posting list, we compute $\delta_i = f_{i+1} - f_i$ and we store the list $f_0, \delta_0, \delta_1, \ldots, \delta_{\ell-2}, f_\ell$. We store the smallest file ID in the beginning to ensure that we can reconstruct the original posting list. Similarly, we store the last file ID f_ℓ in an absolute representation to ease incremental index updates (see Section 3.7 for details).

To leverage the space gain of delta encoding, we store the deltas using a variable-length s-bit encoding instead of fixed-size deltas. s-bit encoding uses chunks of $s+1$ bits, where the most significant bit of a chunk indicates that another chunk follows. For example, consider the number 6 743, which is `11010 01010111` in binary. In an uncompressed form, we would naively require 32 bits to store this number. With 7-bit encoding, the number would be encoded as `11101001 00010111`, i.e., 16 bits only. Similarly, with 5-bit encoding it would be `11101 01010 00111`, i.e., 18 bits. For $s = 3$, the encoding would be `111 011 010 100 111 010 001`, i.e., 20 bits.

Given that the optimal value for s depends on the size of the posting list, YARIX uses a hybrid s-bit encoding that chooses the optimal s. During the initial index build, the optimal value of s is determined per posting list by comparing different choices. Header bits encode this optimal choice.

3.6 Grouping

We previously removed offsets from our index to save space. Doing so made pure index searches unsound as the returned file IDs were an overapproximation and soundness could only be gained back by using the index as an optimizer for sequential search. In a situation where even more compression is required, we can apply a more aggressive overapproximation to the posting lists, which we call grouping.

General idea. The basic idea is as follows: we randomly assign each file ID in a posting list to a group and store a group ID instead of the file ID. Storing group IDs instead of file IDs brings two optimization benefits. First, by choosing the possible number of groups small enough it further decreases the footprint given that storing a group ID requires fewer bits than a file ID even in an uncompressed form. Second, the random mapping from file IDs to group IDs creates collisions, as multiple file IDs in a posting list may belong to the same group and thus also saves space.

Formally, given a posting list mapping an n-gram x to a list of file IDs $f_1, \ldots, f_\ell$, we compute a group ID g_i for each f_i as follows:

$$g_i = f_i \mod \mathsf{gn}, \tag{1}$$

where gn is the number of groups. We could have $g_i = g_j$ for $i \neq j$, i.e., collisions can occur as previously discussed. This means that the index search for a single n-gram x will now yield a of set of group IDs $G_x = \{g_1, \ldots, g_\ell\}$ instead of a set of file IDs. It will be necessary to revert G_x to a set of file IDs F_x, which can be done by inverting Equation (1). Formally, let $f_{\max}$ be the largest file ID currently indexed by YARIX, then F_x is the union of a finite subset of the congruence class of g_i modulo gn for each $g_i \in G_x$, i.e.,:

$$F_x = \bigcup_{g_i \in G_x} \{g_i + k\mathsf{gn} \mid k \geq 0, g_i + k\mathsf{gn} \leq f_{\max}\}. \tag{2}$$

When searching for a string there will be one such set for each distinct n-gram of the string, i.e., sets $F_{x_1}, \ldots, F_{x_\ell}$. To get the final set of file IDs F, that will then be searched sequentially, we perform a set intersection, i.e.:

$$F = \bigcap_{x_i \in \{x_1, \ldots, x_\ell\}} F_{x_i}. \tag{3}$$

Varying moduli reduce over-approximation. Note that the immense overapproximation created by Equation (2) (because of the set union) is compensated by the set intersection in Equation (3). As previously mentioned, the choice of gn influences the disk footprint as smaller group IDs require less space and trigger more collisions. For example, by ensuring

that $\mathsf{gn} \leq 2^{16}$ we can guarantee that a group ID will never occupy more than 2 bytes. However, to minimize the overlap between different posting lists in Equation (3) we want gn to be prime and most importantly to differ per n-gram $x_i \neq x_j$. That is, instead of a fixed gn, we want a variable gn_x. This is obviously not possible in most situations, e.g., consider the case $n = 4$ where 2^{32} prime numbers would be required, which does not work if $\mathsf{gn}_x \leq 2^{16}$. However, by inspecting Equation (3) it becomes evident that a problem only occurs if all gn_{x_i} are the same, because then the intersection will be all the true file IDs plus all their congruence classes modulo gn_{x_i}. As soon as one gn_{x_i} is different this is not the case as the congruence classes do not all overlap anymore. Therefore, we simply define gn_x as a uniform hash function that maps n-grams to the list of the largest m prime numbers smaller than gn. By choosing the m largest prime numbers we ensure that the groups stay close to our desired number of groups gn in order to not overapproximate too much. For a search string consisting of ℓ different n-grams, the probability of all n-grams sharing the same modulus is therefore given by $m^{-\ell}$. In our evaluation in Section 4, we will use $m = 256$, which will make that probability sufficiently small even for short search strings. For example, for a search string consisting of 2 distinct n-grams, we have $m^{-\ell} = 0.0015\%$. From a computational point of view it is also worth noting that Equation (2) and Equation (3) can be computed and at the same time most of the costly set operations can be avoided. For instance, let G_i be the smallest set of group IDs. We then iterate over each $f \in F_i$ and check for each n-gram $x_j \neq x_i$ if $G_j \ni g = f \mod \mathsf{gn}_{x_j}$. Only if this is the case for all n-grams x_j we know that $f \in F$. This avoids the set union in Equation (2) by generating F_i element by element and also avoids the set intersection in Equation (3) by performing a cheap set element check $g \in G_j$ of an already constructed set.

Selective Grouping: It is advisable to not apply grouping to every posting list. For example, consider 16-bit group IDs and a posting list consisting of 300 000 file IDs. In this case almost all group IDs would be occupied after grouping and using these posting lists during search would reduce the compensatory effects of the intersection. To account for this, we can determine a threshold τ for the size of posting lists up until which they will be considered for grouping. We will evaluate different choices for the grouping threshold τ as well as the general overall effectiveness of grouping in Section 4.

Example. In the following, we will give a brief example for our grouping methodology. Consider a case where $N = 100$ malware samples need to be indexed, i.e., we can assume 7 bits per file ID for simplicity. Assume we use $\mathsf{gn} = 8$ groups, i.e., we can use 4 btis per group ID and we use the primes $11, 13, 17, 19, \ldots$. Furthermore, for the sake of simplicity assume we use 1-grams instead of 4-grams. We want to search for all files containing the bytes A, B and C, which we assume are matched by the file IDs **30** and **98**.

Let the corresponding posting lists look as follows:

$$A \mapsto \{18, 30, 33, 39, 40, 49, 98, 99\}$$
$$B \mapsto \{10, 30, 31, 53, 98\}$$
$$C \mapsto \{25, 30, 33, 52, 83, 98\}$$

If no grouping is used, the intersection of those posting lists yields the optimal result, i.e., $\{\mathbf{30}, \mathbf{98}\}$.

If we use the simple grouping mechanism with the fixed modulus gn as described in Section 3.6, the posting lists look as follows:

$$A \mapsto \{18 \bmod 8, \mathbf{30} \bmod 8, \ldots\} = \{0, 1, \mathbf{2}, 3, \mathbf{6}, 7\}$$
$$B \mapsto \{10 \bmod 8, \mathbf{30} \bmod 8, \ldots\} = \{\mathbf{2}, 5, \mathbf{6}, 7\}$$
$$C \mapsto \{25 \bmod 8, \mathbf{30} \bmod 8, \ldots\} = \{1, \mathbf{2}, 3, 4, \mathbf{6}\}$$

A set intersection of those posting lists yields the group IDs 2 and 6 which can be restored to the following 25 file IDs, i.e., all file IDs that are congruent to 2 or 6 (mod 8):

$$\{2, 6, 10, 14, 18, 22, 26, \mathbf{30}, 34, 38, 42, 46, 50,$$
$$54, 58, 62, 66, 70, 74, 78, 82, 86, 90, 94, \mathbf{98}\}$$

If we instead use different moduli per posting list, we get the following groups:

$$A \mapsto \{18 \bmod 11, \mathbf{30} \bmod 11, \ldots\} = \{0, 5, 6, 7, 8, 10\}$$
$$B \mapsto \{10 \bmod 13, \mathbf{30} \bmod 13, \ldots\} = \{1, 4, 5, 7, 10\}$$
$$C \mapsto \{25 \bmod 17, \mathbf{30} \bmod 17, \ldots\} = \{1, 8, 13, 15, 16\}$$

To intersect the lists, we first need to revert the grouping process as previously described and intersect the resulting lists of file IDs. This yields the following set of 7 candidate files:

$$\{18, \mathbf{30}, 33, 49, 66, 83, \mathbf{98}\}$$

As we can see, the optimized version with different moduli allowed us to eliminate 18 candidate files.

3.7 Incremental Index Updates

Our file index is sufficiently generic and does therefore not require updates for new YARA rules. However, if new files have to be indexed, our design allows adding further samples to the index in a non-costly manner. Ignoring grouping for the moment, adding a sample to the index can be trivially done by computing its n-grams and updating the necessary posting lists. Even with delta-encoding, updating the posting list is cheap, because we store the last file ID of the posting list in an absolute representation (additionally to the relative representation). This means we do not need to reconstruct the whole posting list before adding the new file ID.

When considering grouping, special care has to be taken during incremental index updates. Adding new files to the

index involves extending posting lists. If the size of a grouped posting list surpasses the threshold τ after adding new files, we would need to revert the grouping which would potentially involve scanning billions of files in the worst case. This is not a viable option. Instead, we can group every posting list, and if a list surpasses the threshold, we can store file IDs instead of group IDs. Such a posting list will then consist of file IDs *and* group IDs. As we will show in Section 4.7, the distribution of posting list sizes of a set of indexed samples can be extrapolated to learn the distribution of posting list sizes for the desired number of samples. This means that a reasonable value for τ can be chosen *a priori* to ensure predictable grouping behavior and to minimize the number of posting lists surpassing the threshold τ.

4 Evaluation

We evaluate YARIX and benchmark the performance of YARIX regarding index build time, disk footprint and search performance.

4.1 Dataset

We build an inverted 4-gram index over $N = 32$M malware samples with a total uncompressed size of 13.79 TiB. We will discuss the choice of $n = 4$ in Section 4.5. The samples stem from VirusShare.com, non-public repositories and AV vendor feeds, and include well-known malware families for all popular operating systems. We retrieved anti-virus labels for 900k samples and found more than 19k Microsoft labels and more than 21k Kaspersky labels, indicating high diversity. The feeds are updated on a daily basis and thus represent a real-world excerpt of the malware ecosystem. In order to evaluate the YARA search, we use all 1 404 YARA rules from the Malpedia project [33] repository[4]. About 286k of the 32M samples matched at least one of the 1 404 YARA rules.

4.2 Index Build Time

We developed a prototype in C++ and Python 3 that implements YARIX as described in Section 3 and Section 2 for $n = 4$. The prototype ran on a system using 2 Intel® Xeon® Processor E5-2667 v4 CPUS utilizing 24 threads. We additionally used an NVME drive as an intermediate fast storage for storing parts of the index that were merged and moved to a traditional HDD setup (7.2K RPM SAS-12Gb/s). This setup was capable of indexing 10^6 samples in 15 hours, showing that YARIX can process over 1M samples per day on just a single system. If we wrote the index directly on the HDD, the operation took 39 hours. Writing directly to the HDD is slower than using the NVME as intermediate storage, because the index creation requires many IO operations. Note that all subsequent experiments do not use parallelism to foster comparability to standard (non-parallel) YARA.

[4]Revision: `0b7d57251cd0fecf149d47d9c5564617c9fa7978` (Tue Nov 26 09:19:08 2019 UTC)

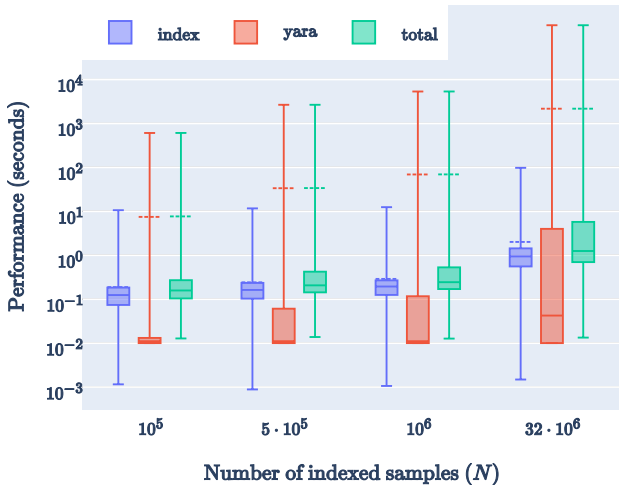

Figure 4: Search performance breakdown for all YARA rules and different numbers of indexed samples. Whiskers denote $\pm 1.5 \cdot$ IQR, dashed line denotes mean. Lower is better.

4.3 Correctness

To empirically validate the correctness of our approach, we compared the results of YARIX with sequential YARA scans. We scanned all 32M samples with every YARA rule and checked if the candidates yielded by the YARA optimization of YARIX for that rule is a superset. This was the case for all the rules, which confirms the completeness of YARIX. Given that YARIX leverages standard YARA to refine the candidate files in the final step, YARIX also guarantees soundness. In total, 37 out of the 1 404 (2.64%) rules can not be optimized by YARIX for reasons described in Section 2.2. That is, these rules contain too many expressions that we cannot handle or they contain too many unoptimizable strings.

4.4 YARA Search Performance

For evaluating the search performance of YARIX, we first created an index for $N \in \{10^5, 5 \cdot 10^5, 10^6, 32\text{M}\}$ samples each. Then we queried all 1 404 YARA rules with each of these indexes with YARIX as described in Section 2 and measured the elapsed time. By choosing different numbers of indexed samples, we can see how YARIX scales. The result of this experiment is depicted in Figure 4. We also break down the search time into the time spent using the index for narrowing down the set of candidate files and the time spent sequentially scanning this optimized set with YARA. The execution time largely depends on the number of indexed samples and grows sub-linearly due to the fast index lookups. In the case

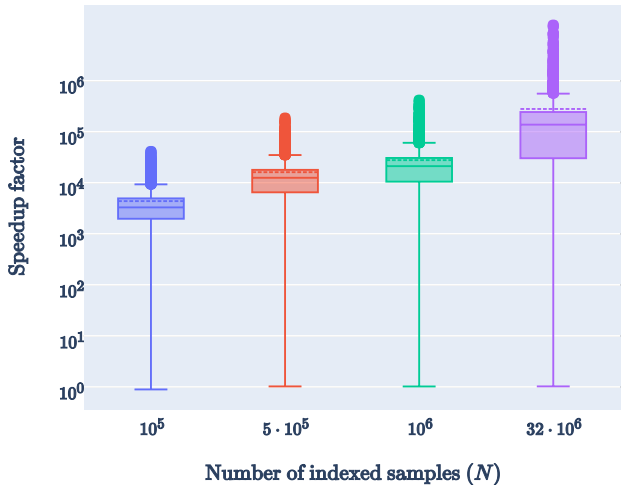

Figure 5: Search performance speedup of YARIX compared to sequential YARA scanning. Higher is better.

#groups \ τ	200	400	5000	10000	∞
2^{14}	1	1	18.49	32.03	$\approx 5.72 \cdot 10^5$
2^{15}	1	1	7.8	11.56	$\approx 2.77 \cdot 10^5$
2^{16}	1	1	4	5.13	$\approx 9.77 \cdot 10^4$

Table 1: Average number of search candidates relative to non-grouping for different number of groups and grouping thresholds τ.

of $N = 32$M samples, querying the index with a rule takes 12.47 seconds in total, 9.38 seconds for querying the index and 0.42 seconds for sequential YARA scanning in the median. The high mean values for the total time and the YARA scan time are the result of a few outliers where the rule consisted of expressions that yielded large overapproximations. In these cases, we had to perform a sequential YARA scan on nearly all samples. The index operation takes longer for larger indexes, as the posting lists become larger. As a result, the decoding of those lists takes longer. Also, as the resulting sets become larger, the set intersections and unions become more costly.

These experiments were carried out in a single threaded workload in order to minimize caching and scheduling effects and ensure reproducibility. In a real-world deployment it would be trivial to distribute the workload among different threads and/or machines to significantly improve the performance.

To get a better understanding of how much faster YARIX is compared to sequential scanning, Figure 5 depicts the speedup factor. In the case of 32M samples, YARIX is five orders of magnitude faster than sequential YARA scanning on average. The speedup factor gradually increases with the number of indexed files.

Finally, we use the search performance to evaluate different values for the grouping threshold τ and the influence of the number of groups. To do so, we compute the number of candidates, i.e., the file IDs that are yielded by the YARA optimization of YARIX and used by the sequential YARA scan. We compare the number of candidates in the case of grouping to the number of candidates in the case where no grouping takes place to understand how much accuracy is lost. The results of this analysis are depicted in Table 1. A first intuitive observation here is that $\tau = \infty$ is not a viable option as it blows up the number of candidates. For example, if we have 2^{14} groups, using $\tau = \infty$ yields approximately $5.72 \cdot 10^5$ more candidates on average than the non-grouping version. Note that this means that the sequential YARA scan that would follow will be 5 orders of magnitude slower, which is not a viable option. However, if we use a grouping threshold, this slowdown is drastically reduced. Consider, for example, the case of 2^{16} groups and a grouping threshold of $\tau = 10000$, which leads to only 5.13 times more candidates than the non-grouping case on average. Even in the worst case of 2^{14} groups this threshold would only slow down the YARA search by a factor of 32.03. We will see in Section 4.6 that such a threshold would group more than 98% of all posting lists. If the n-grams of the strings of YARA rules were randomly distributed, this would mean that almost all resulting posting lists would be subject to grouping. As a result the overapproximation would become too large and the actual slowdown factor would be closer to that of $\tau = \infty$, i.e., in the order of 10^5. However, as this result shows the n-grams of strings inside YARA rules do now follow a normal distribution and there are enough non-grouped posting lists in practice to ensure good performance. We can use this to our advantage if we later discuss the disk footprint in Section 4.6. For example, choosing 2^{15} number of groups with a threshold of $\tau = 10000$ would decrease the relative disk footprint from 149.5% to 65.48% while only slowing down the search by a factor of 11.56 on average.

4.5 Choosing n

When choosing a suitable value for n we wanted to pick the smallest n that has a reasonable search performance. We empirically validated that $n = 3$ is an unsuitable choice by comparing it against $n = 4$, because it delivered 1421.83 times more candidate files during YARA search. Given that smaller n would only make this worse, we identified $n = 4$ as a lower bound. Regarding larger values for n, we verified that choosing $n = 5$ would almost double the number of unoptimizable rules from 37 to 73. Additionally, the disk footprint would suffer from such a choice. First, there are more unique 5-grams than 4-grams per file that need to be indexed. Second, the posting lists would become more sparse and as a result the delta encoding would not be as efficient. Given that all of this

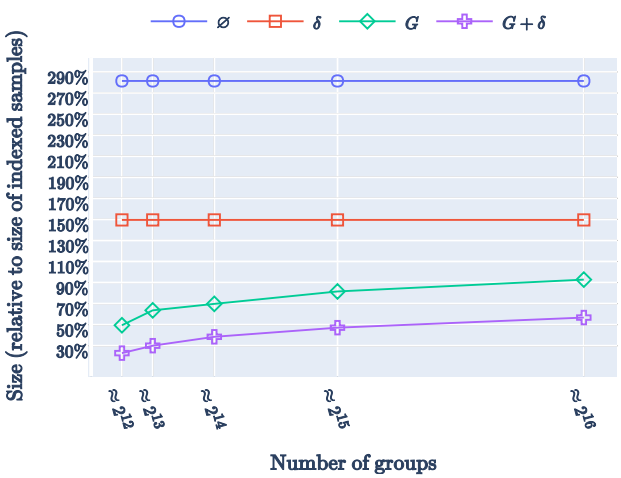

Figure 6: Disk footprint for different number of groups for $N = 32$M indexed samples ($\varnothing$ = uncompressed, G = grouping was used, δ = delta encoding was used, $G + \delta$ = grouping and delta encoding was used). Lower is better.

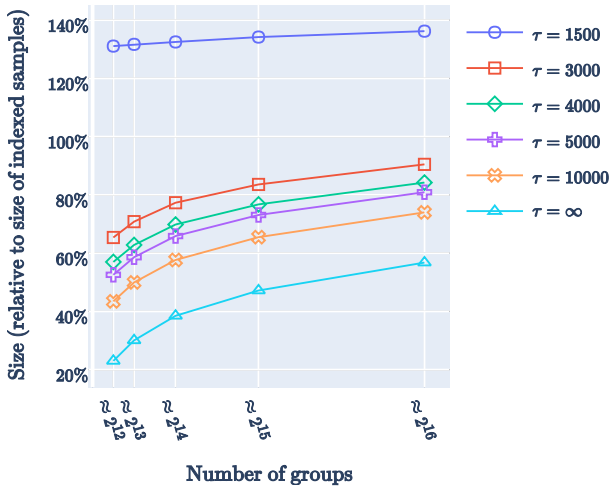

Figure 7: Disk footprint for varying number of groups and grouping thresholds for $N = 32$M indexed samples, using $G + \delta + \tau$. Lower is better.

would be further amplified by choosing even larger values, we chose $n = 4$.

4.6 Disk Footprint

To evaluate the disk footprint of YARIX, compare the size of the index for all N samples in different configurations regarding the number of groups and grouping thresholds τ as discussed in Section 3.

We define the size of the index as the accumulated number of bytes it takes to encode all posting lists. In particular, this does not include overhead introduced by the file system or the file format that is used to organize the posting lists. For example, in our test setup we used the ext4 file system and a folder structure `a/b/c` to organize posting lists. Here, `a`, `b` and `c` each represent a byte of a 4-gram and the file `c` uses a custom file format to store the 256 postings lists of all n-grams that share the `abc` prefix. The overhead introduced by this approach is both environment- and implementation specific, but certainly constant and almost negligible for larger indexes. For example, the overhead introduced by the file system is asymptotically constant as we never expect more than 2^{32} posting lists, which was already saturated in the case of the N samples. Moreover, if we used a different file system that supports more files than ext4 we could store each posting list in an individual file and thus would not have the overhead of the custom file format. By not including this overhead we thus make the analysis implementation- and environment independent.

Figure 6 and Figure 7 show the result of this analysis. The former depicts the cases where no grouping threshold was used and the latter includes cases where grouping with delta encoding and a grouping threshold was used. When grouping was applied, we use the 256 largest prime numbers that fit into a certain number of bits to optimize the disk footprint. This means that we have slightly fewer than 2^x groups, which is why we use the "$\approx$" notation in both figures. The choices for both the number of groups and the grouping thresholds present a reasonable range of options depending on the trade-off between disk footprint and search performance, which will be discussed later.

In an uncompressed form, the index requires 281.46% of the space required for all samples. Note that this is already a large improvement over offset-sensitive indexes. This is because assuming 4 byte file IDs without offsets, a file consisting of m bytes has $m - 3$ n-grams, which will add $4m - 12 \approx 4m$ bytes to the index, because the file ID will be added once for each n-gram. This would yield an overhead of $\approx 400\%$ not accounting for additional overhead required for storing file IDs several times including their positional information. Since YARIX is offset-free, we can abstract from n-grams occurring multiple times, which yields the depicted improvement.

Applying delta encoding reduces the relative size significantly to 149.5%. Grouping further shrinks the required disk space. The smaller the number of groups, the smaller the disk footprint. This can be explained by the fact that the improvement in disk footprint with grouping has two reasons: collisions by chance and less bits required to store group IDs than file IDs. The smallest footprint is 23%, which is achieved by 2^{12} groups with delta encoding. Without delta encoding, 2^{12} groups have a relative size of 49.46%. The largest footprint for grouping with delta encoding and without delta encoding is 92.8% and 56.76% respectively.

However, none of these grouping variants are useful in practice because of their poor search performance (cf. Section 4.4).

Instead, a reasonable choice of the grouping threshold τ (maximum posting list length to apply grouping on) will be required in practice. Choosing a threshold is a trade-off between disk footprint and search performance. The larger the threshold, the more posting lists will be grouped, the better the disk footprint improvement. However, the more posting lists are subject to grouping, the larger the overapproximation, which degrades search performance as we have seen in Section 4.4. The disk footprint of different choices of τ on top of grouping with delta encoding is depicted in Figure 7. We can see that a small threshold like $\tau = 1\,500$ has little effect on the disk footprint as too many posting lists are not subject to grouping. With 2^{16} groups, for example, the footprint is 136.21%, which is only a small improvement over the 149.5% of delta encoding alone. However, we see that the disk footprint is sensitive in the beginning for changes in τ and slows down as τ becomes larger. For example, doubling τ from 1 500 to 3 000 decreases the disk footprint from 136.21% to 90.48%, while doubling it from 5 000 to 10 000 decreases the footprint from 80.91% to 74%. Grouping all posting lists, i.e., $\tau = \infty$ (equivalent to $G + \delta$ cf. Figure 6), would decrease the footprint to 56.76%. By choosing a threshold of $\tau = 5\,000$ we already group 96.31% of all posting lists and by choosing $\tau = 10\,000$ this percentage increases to 98.71% (cf. Figure 10 in the appendix).

4.7 Scalability

One of the core goals of YARIX is scaling to large malware sample databases. Until now, we have evaluated YARIX on a real-world dataset consisting of $N = 32$M samples, but we want to understand how YARIX scales for 2^{32} samples by extrapolating our results. One particular challenge for such an extrapolation is to estimate the distribution of n-grams on a larger sample set. Only the uncompressed index size can be trivially extrapolated, as having kN samples will require roughly k times the space of storing/indexing N samples. However, to extrapolate the compressed disk footprint, we have to study the posting list distributions, as they influence grouping and delta encoding.

To this end, we use combinatorics to estimate the expected number of groups the file IDs of a posting list will belong to. A detailed description of this method and extrapolated figures are given in Appendix A. Overall, this extrapolation confirms that the disk footprint scales linearly also if grouping and delta encoding is applied. Following the intuition that the distribution of n-grams among samples can be extrapolated as described in Appendix A, we have reason to believe that the sub-linear trend of the search performance will continue for larger datasets as well. We already empirically confirmed this assumption in Figure 4 where we indexed differently sized subsets and observed a sub-linear progression in search performance.

5 Case Studies & Future Work

For a small subset of YARA rules, the YARA optimizer of YARIX has not eliminated enough candidates and has left too many files for a sequential YARA scan. While most rules perform well and the filtering by the index does most of the work, applying YARIX to some rules excluded almost no files from the sequential YARA scan. In the following, we study one rule that performed well and the three rules that performed the worst and had no empty result, and discuss the reasons and mitigations for their poor performance.

Case Studies. First, an example that performed well is a YARA rule for the Retefe [16] banking trojan that matches 16 indexed samples. This rule consists of a single `all of them` expression which requires 7 plain strings to be present in the sample. All plain strings are of adequate length and can thus be captured by YARIX. The longest of those is `"security add-trusted-cert -d -r trustRoot -k /Library/Keychains/System.keychain %@"`.
Using YARIX with this string already yields the 16 actual matches.

Next, an example that performed poorly is a rule for the PlugX malware [29]. In this case YARIX filtered a set of 12M candidates and a sequential YARA scan yielded merely 84 actual matches. The condition of the rule is of the form $x_0 \vee x_1 \vee x_2$ and x_0 is `{E8 00 00 00 00 58} at 0`, which requires the x86 instructions `call 0x5 ; pop eax` ("get program counter" gadget) to be present at offset 0 of the sample. Given that YARIX is offset-free and thus has to abstract from the `at` constraint, any file that contains
`E8 00 00 00`, `00 00 00 00` or `00 00 00 58` will be a candidate for sequential YARA scanning. This is the case for almost all 12M files and is thus the culprit of the problem. After reverse-engineering the malware, we found that x_1 and x_2 are used to capture the custom API importing scheme of the malware. Both x_1 and x_2 are characteristic enough to identify the malware, and hence, we removed x_0 from the condition. As a result, the index now yielded the 84 actual matches, which is a perfect optimization.

Another bad performing example is a rule for the Smokeloader malware [26] that delivered roughly 10M candidate files, of which only 2 remained after sequential scanning. The root cause of this is the hex string `53 56 57 8B 7? 0C B? [4] E8 [4] 68 [4] 5?`, which checks for a sequence of x86 instructions. The problem here is that YARIX can only handle the first 4 bytes because of the wildcards. This sequence `53 56 57 8B` is the encoding of `push ebx ; push esi ; push edi; mov ??`, i.e., a series of pushes and moves. This is a common pattern found in binaries and is responsible for almost all 10M candidates. We reverse-engineered the Smokeloader samples of our dataset and additionally acquired more samples and found that the wildcards introduced by the authors do not seem to be necessary. We thus hard-coded some of the offsets and

String Type	#	Optimizable	Optimizable w/o null bytes
Plain	3635	99.48% (3616)	99.09% (3602)
Hex	11753	99.83% (11380)	95.20% (11189)
Regex	91	93.41% (85)	92.31% (84)

Table 2: A breakdown of how removing null bytes affects string optimizability. 31.13% of the rules contain plain strings, 30.13% hex strings, and 1.28% regular expressions.

constants, which reduced the set of candidates to 2. While such hard-coding makes the rule less generic as it now can be evaded by changing an offset, this is merely a theoretical limitation as YARA rules are in general not resistant to this kind of instruction-level obfuscation. In general, any obfuscation or packing attempt to evade YARIX boils down to evading YARA, as YARIX is both sound and complete.

Last, a rule for a dropper of the SnatchLoader [28] malware has 3 matches, but YARIX only manages to narrow down the search to roughly 7M samples. The rule consist of a single `3 of them` expression where `them` refers to 4 hex strings. Two of those strings have no streak of 4 consecutive bytes, which is why YARIX simplifies the expression to `1 of them`. One of the strings has only one streak which is the hex string `00 00 ff 24`, which is responsible for all the 7M candidates. After analyzing the malware, we found that all hex strings by themselves are good and characteristic indications of the custom self-written loader of the malware. We thus decided to remove one of the hex strings that has no streak of 4 consecutive bytes, which simplified the expression to `2 of them`. This caused the number of candidates to drop to the 3 actual matches.

Future Work. Another more general problem are UTF-16 encoded (wide) strings, which are common in Windows applications. YARIX faces the problem that wide strings are mostly used in practice with code points that fit into 8-bits, which makes every second byte a null byte. Consider, for example, the string `nice` which is `6E 69 63 65` in ASCII and `6E 00 69 00 63 00 65 00` in UTF-16LE. The 4-grams of these strings suffer from entropy, as every second byte will be a null byte and is thus more likely to be present in many files as it can be compared to searching for the 2-grams of an ASCII string. This could be solved by not indexing null bytes completely. That is, during indexing and before querying a posting list, all null bytes are removed. This would solve the problem with UTF-16 strings and could potentially also improve disk space as null bytes occur often in binary files. However, stripping null bytes could also lead to cases where not enough consecutive bytes are found in a hex string, for example. We experimented with this idea on a subset of 100 000 samples. Disk footprint was reduced by 46.84% compared to ordinary indexing. Regarding the string optimizability, Table 2 breaks down the types of the strings that are contained in all rules and how the feasibility is affected. As expected, hex strings are affected the most, while the other cases remain rather unchanged by this optimization. We leave a thorough evaluation of this idea as future work.

Overall, these findings make us believe that in practice most YARA rules can be used with YARIX and it is feasible in practice to convert YARA rules to practically equivalent ones in case searches with YARIX require optimization. In particular, if YARIX gets incorporated into an existing workflow, the malware analysts can optimize a rule a priori as opposed to modifications by a third person a posteriori. In the future we plan to investigate to what extent rules can be optimized in an automated fashion. For example, if we have enough samples that match a rule, we could systematically reason about whether or not all wildcards in a string are required if all the matched samples have the same byte at the wildcard position. Similarly, YARIX could be used to automatically generate YARA signatures for a malware family. Given enough samples of the family, shared sequences of bytes could be extracted and YARIX could quickly check if these sequences do not appear in other samples.

6 Related Work

While there are other approaches tackling the problem of searching content in large malware collections [30, 31] using file index technologies, none of these solutions support YARA scanning. Additionally, these approaches show worse performance in both disk space and index build time than YARIX. Related efforts for improving YARA scalability such as KLara [22] parallelize the YARA scans with distributed computing techniques. Apart from being more resource demanding by nature, such efforts also do not offer the same magnitude of speedup in practice (multiple TiB in 30 minutes with KLara vs seconds with YARIX).

So far, academic efforts mainly aimed to improve the scalability of malware analysis and malware detection. For example, several sandboxes [7, 14, 38] provide a framework that allows observing the dynamic behavior of malware in a scalable fashion. Based on these efforts, various refinements focus on the scalability of analyzing special dynamic behaviors, such as evasive behavior [5, 25] or understanding the obfuscation techniques of malware [9, 36]. Similarly, to obtain a better picture of the malware landscape, related work proposed scalable clustering [6, 32] or malware triage [19, 35] approaches. Closer to our goal, scalable approaches to detect special types of malware [23, 41, 42] have been proposed. While there are a few scalable methodologies proposed that operate statically [18, 24], the majority of these efforts operate in a dynamic analysis setting. Dynamic analysis has, however, an inherent cost associated with it that cannot be compared to our case, as we do not perform any execution or emulation of the malware. Finally, also related are malware signature generators [1, 8, 12] that automatically try to create (YARA) rules for malware analysis. Our approach is orthogonal to

these solutions and tackles a different problem, i.e., scaling complex static malware signatures to large malware data sets.

One of our main contributions is a scalable and efficient search methodology that can be used with arbitrary rules specified in YARA, a widely-used industry standard. We therefore present the first malware search methodology that retains full compatibility to off-the-shelf YARA rules. Earlier attempts to efficiently index malware have taken different directions and lack such support. Hu *et al.* [17] propose a system called Symantec Malware Indexing Tree (SMIT) that indexes malware using their function-call graphs. Function-call graphs are a high-level abstraction of malware and thus creating and processing them is relatively costly. Jin *et al.* [20] propose BIGGREP, a file index for malware. The authors also use an offset-free inverted *n*-gram index and a similar delta encoding scheme. However, BIGGREP is restricted to plain *n*-gram string search only and does not support any sort of rule language (such as YARA) or more complex constructs (such as regular expressions). That is, our contribution goes beyond providing just an index. When looking at the index itself, there are further differences between YARIX and BIGGREP. First, we introduce file ID grouping, an effective compression methodology that reduces the disk footprint by over 45% while maintaining search performance. Second, our optimal variable length encoding is superior to the static 7-bit encoding used by Jin *et al.*. The disk footprint reported by Jin *et al.* is thus significantly higher than ours. In BIGGREP, storing the posting lists requires up to 700% of the disk space of the indexed samples. With YARIX, we have shown that the expected relative disk space for 2^{32} samples is much smaller: 149.5% without grouping and 74% with grouping. Third, the index query time of YARIX outperforms the related work. Jin *et al.* mention that querying the string `\drivers\mrxcls.sys` requires 17.38 seconds, which only takes 0.8 seconds with YARIX. When accounting for the fact that Jin *et al.* have indexed 6 million fewer samples than we did, YARIX is over 26 times faster and thus poses a major improvement.

The challenge to efficiently compress indexes has been explored in depth outside of the security community. Wang *et al.* [37] give a complete overview of the different techniques that have been developed in the past decades. Most of the presented techniques are based on a simple variable delta encoding [11] similar to our version, although we use the optimal bit encoding. The objective of these different methods is, however, usually more targeted towards micro optimizing the encoding and decoding. For example GroupVB [13] is an optimized version of variable delta encoding that aims at microarchitectural improvements to reduce branches taken by the CPU. Another example is PForDelta [43] that works by collecting blocks of deltas by choosing the smallest b in the block such that a majority of the deltas can be encoded in b bits. The optimized versions of PForDelta, i.e., NewPForDelta [39], OptPforDelta [39], and SIMDPforDelta [27] are based on the same compression principle and only aim at encoding and decoding performance. This is different to our file ID grouping method that is more targeted towards improving the disk footprint using a justifiable over-approximation. Existing lossy index compression approaches mainly rely on bloom filters, which however cannot be used to intersect and thus reduce the set of search candidates across multiple combined searches—a vital aspect of our novel grouping.

7 Conclusion

We presented YARIX, a novel YARA search engine that significantly optimizes searches for malware files with arbitrary off-the-shelf YARA rules. We introduced a methodology to convert YARA rules into search terms that can be fed to the inverted *n*-gram index of YARIX to optimize YARA searches. Our evaluation of YARIX demonstrates that its inverted *n*-gram index can drastically reduce the files that have to be scanned sequentially. At the same time, the index footprint is reasonably small due to several compression techniques used including a novel grouping-based compression scheme. That is, while optimizing YARA searches by five orders of magnitude, only 74% of the accumulated disk space of all samples is required to store the inverted *n*-gram index of YARIX.

Availability

The YARIX reference implementation can be obtained at https://github.com/mbrengel/yarix.

Acknowledgments

We would like to thank the anonymous USENIX reviewers of this paper as well as Giuliano Schneider, Benedikt Birtel and the anonymous AEC reviewers for testing YARIX. We would also like to thank VirusShare and our anonymous partners from the AV industry who supplied us with malware samples. Our thanks also goes to Veelasha Moonsamy for shepherding this paper. Finally, we would like to thank Tillmann Werner for initial brainstorming about the general problem in summer 2018. We apologize for neglecting your idea to use prefix trees to solve this problem.

References

[1] Mohannad Alhanahnah, Qicheng Lin, Qiben Yan, Ning Zhang, and Zhenxiang Chen. Efficient Signature Generation for Classifying Cross-Architecture IoT Malware. In *Conference on Communications and Network Security (CNS)*, 2018. doi:10.1109/cns.2018.8433203.

[2] Victor Manuel Alvarez. YARA – The pattern matching swiss knife for malware researchers, 2020. URL: https://virustotal.github.io/yara/.

[3] AV-TEST. Malware Statistics & Trends Report, 2020. Last accessed at: June 19, 2020. URL: https://www.av-test.org/en/statistics/malware/.

[4] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[5] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

[6] Ulrich Bayer, Paolo Milani Comparetti, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2009.

[7] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. A View on Current Malware Behaviors. In *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (USENIX LEET)*, 2009.

[8] Felix Bilstein and Daniel Plohman. YARA-Signator: Automated Generation of Code-based YARA Rules. *The Journal on Cybercrime & Digital Investigations*, 2019. doi:10.18464/CYBIN.V5I1.24.

[9] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, 2018. doi:10.1145/3243734.3243771.

[10] CrowdStrike. Meet The Threat Actors: List of APTs and Adversary Groups, 2019. Last accessed at: June 19, 2020. URL: https://www.crowdstrike.com/blog/meet-the-adversaries/.

[11] Doug Cutting and Jan Pedersen. Optimization for Dynamic Inverted Index Maintenance. In *Proceedings of the Annual International Conference on Research and Development in Information Retrieval (SIGIR)*, 1990. doi:10.1145/96749.98245.

[12] Omid E. David and Nathan S. Netanyahu. DeepSign: Deep learning for automatic malware signature generation and classification. In *International Joint Conference on Neural Networks (IJCNN)*, 2015. doi:10.1109/ijcnn.2015.7280815.

[13] Jeffrey Dean. Challenges in Building Large-scale Information Retrieval Systems: Invited Talk. In *Proceedings of the International Conference on Web Search and Data Mining (WDSM)*, 2009. doi:10.1145/1498759.1498761.

[14] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, 2008. doi:10.1145/1455770.1455779.

[15] FireEye. Advanced Persistent Threat Groups, 2020. Last accessed at: June 19, 2020. URL: https://www.fireeye.com/current-threats/apt-groups.html.

[16] Jaromír Hořejší. The evolution of the Retefe banking Trojan, 2016. Last accessed at: June 19, 2020. URL: https://blog.avast.com/the-evolution-of-the-retefe-banking-trojan.

[17] Xin Hu, Tzi cker Chiueh, and Kang G. Shin. Large-Scale Malware Indexing Using Function-Call Graphs. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, 2009. doi:10.1145/1653662.1653736.

[18] Xin Hu, Kang G. Shin, Sandeep Bhatkar, and Kent Griffin. MutantX-S: Scalable Malware Clustering Based on Static Features. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2013.

[19] Jiyong Jang, David Brumley, and Shobha Venkataraman. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, 2011. doi:10.1145/2046707.2046742.

[20] Wesley Jin, Charles Hines, Cory Cohen, and Priya Narasimhan. A Scalable Search Index for Binary Files. In *Proceedings of the International Conference on Malicious and Unwanted Software (MALWARE)*, 2012. doi:10.1109/malware.2012.6461014.

[21] Kaspersky. Targeted cyberattacks logbook, 2018. Last accessed at: June 19, 2020. URL: https://apt.securelist.com/#!/threats/.

[22] Kaspersky. Klara, 2020. URL: https://github.com/KasperskyLab/klara.

[23] Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2016.

[24] Dhilung Kirat, Lakshmanan Nataraj, Giovanni Vigna, and B. S. Manjunath. SigMal: A Static Signal Processing Based Malware Triage. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2013. doi:10.1145/2523649.2523682.

[25] Dhilung Kirat and Giovanni Vigna. MalGene: Automatic Extraction of Malware Analysis Evasion Signature. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, 2015. doi:10.1145/2810103.2813642.

[26] Lastline. An Analysis of PlugX Malware, 2013. Last accessed at: June 19, 2020. URL: https://www.lastline.com/labsblog/an-analysis-of-plugx-malware/.

[27] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, may 2013. doi:10.1002/spe.2203.

[28] Malpedia. SnatchLoader, 2019. Last accessed at: June 19, 2020. URL: https://malpedia.caad.fkie.fraunhofer.de/details/win.snatch_loader.

[29] Malpedia. SmokeLoader, 2020. Last accessed at: June 19, 2020. URL: https://malpedia.caad.fkie.fraunhofer.de/details/win.smokeloader.

[30] Andrei Mihalca and Ciprian Oprişa. Full Content Search in Malware Collections. In *IOSec 2018*, 2019. doi:10.1007/978-3-030-12085-6_12.

[31] Andrei Mihalca, Ciprian Oprisa, and Rodica Potolea. Hunting for Malware Code in Massive Collections. In *International Conference on Automation, Quality and Testing, Robotics (AQTR)*, 2020. doi:10.1109/aqtr49680.2020.9129948.

[32] Roberto Perdisci, Davide Ariu, and Giorgio Giacinto. Scalable fine-grained behavioral clustering of HTTP-based malware. *Computer Networks*, 2013. doi:10.1016/j.comnet.2012.06.022.

[33] Daniel Plohmann, Martin Clauß, Steffen Enders, and Elmar Padilla. Malpedia: A Collaborative Effort to Inventorize the Malware Landscape. *The Journal on Cybercrime & Digital Investigations*, 2017. doi:10.18464/CYBIN.V3I1.17.

[34] Christian Rossow, Christian J. Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. In *Proceedings of the Symposium on Security and Privacy (S&P)*, 2012. doi:10.1109/sp.2012.14.

[35] Shanhu Shang, Ning Zheng, Jian Xu, Ming Xu, and Haiping Zhang. Detecting Malware Variants via Function-call Graph Similarity. In *Proceedings of the International Conference on Malicious and Unwanted Software (MALWARE)*, 2010. doi:10.1109/malware.2010.5665787.

[36] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *Proceedings of the Symposium on Security and Privacy (S&P)*, 2015. doi:10.1109/sp.2015.46.

[37] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. An Experimental Study of Bitmap Compression vs. Inverted List Compression. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2017. doi:10.1145/3035918.3064007.

[38] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy Magazine*, 2007. doi:10.1109/msp.2007.45.

[39] Hao Yan, Shuai Ding, and Torsten Suel. Inverted Index Compression and Query Processing with Optimized Document Ordering. In *Proceedings of the International Conference on World Wide Web (WWW)*, 2009. doi:10.1145/1526709.1526764.

[40] YaraRules. YaraRules Project, 2018. Last accessed at: June 19, 2020. URL: https://yararules.com.

[41] Lun-Pin Yuan, Wenjun Hu, Ting Yu, Peng Liu, and Sencun Zhu. Towards Large-Scale Hunting for Android Negative-Day Malware. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2019.

[42] Ziyun Zhu and Tudor Dumitras. FeatureSmith: Automatically Engineering Features for Malware Detection by Mining the Security Literature. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, 2016. doi:10.1145/2976749.2978304.

[43] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *International Conference on Data Engineering (ICDE)*, 2006. doi:10.1109/icde.2006.150.

A Extrapolation to Larger Datasets

Given a posting list of k out of N file IDs and P number of groups, the expected number of occupied groups within that posting list is given by:

$$P - P\left(1 - \frac{1}{P}\right)^k \tag{4}$$

Note that Equation (4) is slightly inaccurate as it does not account for the different prime numbers P that are actually used during grouping. This problem is minor, however, as the difference between the smallest and the largest group modulus becomes negligibly small relative to P.

Then, to account for delta encoding, we estimate the distribution of the pairwise differences to approximate the expected encoding costs. For example, if we use 7-bit encoding, we can encode the differences between 1 and $2^7 - 1 = 127$ (inclusive) with 8 bits, the differences between 2^7 and $2^{2\cdot7} - 1$ with 16 bits, and so on. We only need to know how many differences we expect in these ranges. Formally, this can be tackled as follows: Let a (sorted) posting list of an index over N samples consist of ℓ file IDs between 0 and N, i.e.: $0 \leq f_0 < f_1 < \ldots < f_{\ell-1} < N$. Furthermore, let $\delta_i = f_{i+1} - f_i$ for $0 \leq i < l - 1$ be the pairwise differences that will be encoded. The distribution of those pairwise differences, i.e., the probability that some δ_i equals some fixed difference $\delta \in [1, N-1]$ is given by:

$$\Pr(\delta_i = \delta) = \frac{\binom{N-\delta}{\ell-1}}{\binom{N}{\ell}}. \tag{5}$$

Given that Equation (5) is independent of i and that there are $\ell - 1$ pairwise differences, the expected number of δ_i such that $\delta_i = \delta$ is $(l-1)\Pr(\delta_i = \delta)$. This can be used to compute the expected number of deltas between 1 and some upper limit k:

$$R(\ell, k) = \sum_{\delta=1}^{k} (\ell - 1)\Pr(\delta_i = \delta) \tag{6}$$

$$= \frac{(1-\ell)(N-k)!(N-\ell)!}{N!(N-k-\ell)!} + \ell - 1 \tag{7}$$

$$= (\ell - 1)\left(1 - \prod_{i=1}^{a} \frac{N - (a+b-i)}{N-(a-i)}\right), \tag{8}$$

where $a = \min(k, \ell)$ and $b = \max(k, \ell)$. Now, $R(\ell, k)$ can be used to estimate the costs. For example, there will be $R(\ell, 2^7 - 1)$ pairwise differences that can be encoded with 8 bits and $R(\ell, 2^{2\cdot7} - 1) - R(\ell, 2^7 - 1)$ that can be encoded with 16 bits using 7-bit encoding.

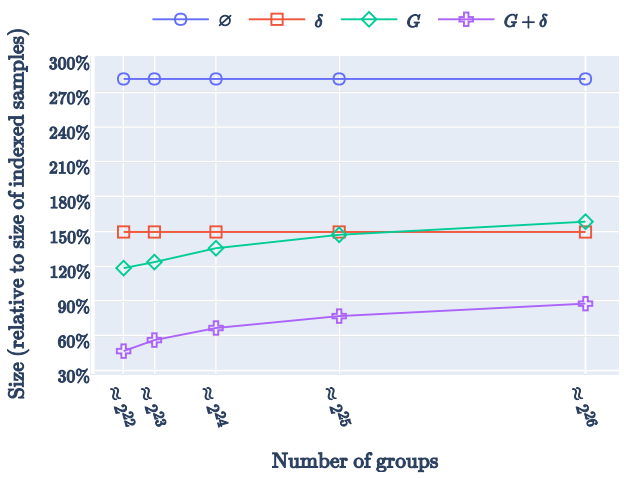

Figure 8: Disk footprint for different number of groups for $N = 2^{32}$ indexed samples ($\varnothing$ = uncompressed, G = with grouping, δ = with delta encoding, $G + \delta$ = with grouping + delta encoding). Lower is better.

Finally, we apply Equation (8) and Equation (4) on the expected distribution of posting list sizes. To do so, we extrapolate the distribution shown in the CDF in Figure 10. That is, if we have y posting lists of size x for N samples, then we will have y posting lists of size kx for kN samples. We verified that this extrapolation is reasonable by applying it to 10^6 samples to estimate 32M samples.

The results of this extrapolation are depicted in Figure 8 and Figure 9, which are the counterparts of Figure 6 and Figure 7, respectively, extrapolated to $N = 2^{32}$ samples. The number of groups and the grouping thresholds are also extrapolated accordingly. As previously stated, the relative overhead of the uncompressed footprint does not change. The delta encoding footprint also does not change. The footprint of grouping alone is much larger, e.g., 158.36% for 2^{26} groups compared to the 92.8% for 2^{16} groups in Figure 6. This is because of the fact that larger number of groups implies wider group IDs, i.e., storing 2^x groups requires x bit per group. Interestingly, this is overcompensated by applying delta encoding, as it decreases the disk footprint by about 9% compared to the case of $N = 32$M samples. Regarding grouping plus delta encoding, the relative disk footprint of choosing 2^{26} groups yields a relative disk footprint of 87.65%. The same trend can be witnessed in Figure 9 which plots the footprint for different values of τ. Choosing $\tau = 10^7$ yields a footprint of 93.71% for 2^{26} groups.

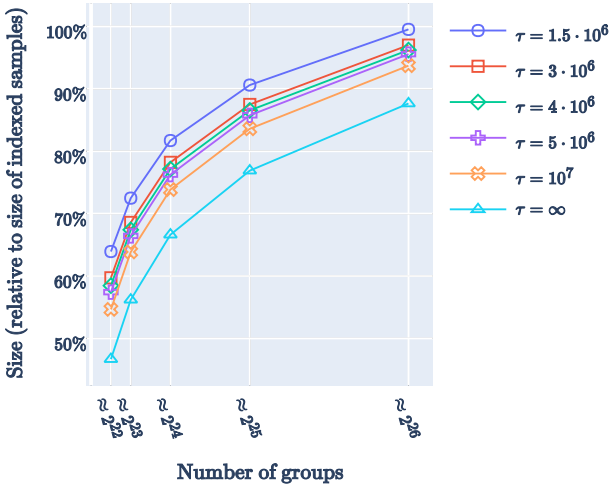

Figure 9: Disk footprint for different number of groups and different grouping thresholds for $N = 2^{32}$ indexed samples. In all experiments $G + \delta + \tau$ was used. Lower is better.

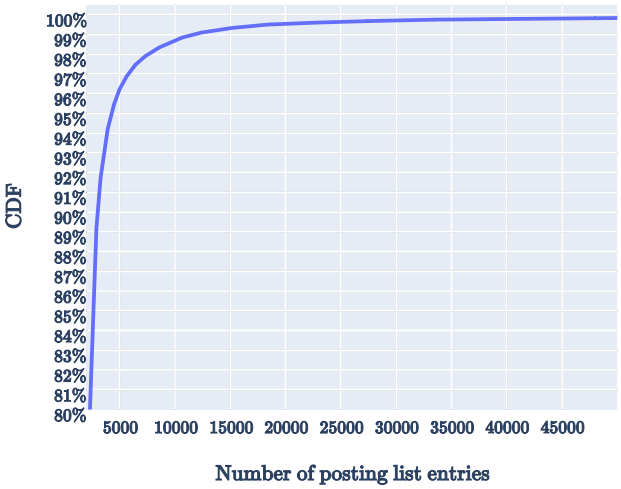

Figure 10: Excerpt from the CDF for the number of posting list entries for $N = 32$M indexed samples.