A static, packer-agnostic filter to detect similar malware samples

Grégoire Jacob[1,3,4], Paolo Milani Comparetti[2,4], Matthias Neugschwandtner[2], Christopher Kruegel[1,4], and Giovanni Vigna[1,4]

[1]University of California, Santa Barbara, [2]Vienna University of Technology, [3]Télécom SudParis, [4]LastLine, Inc.
`gregoire.jacob@gmail.com`, `{pmilani,mneug}@seclab.tuwien.ac.at`, `{chris,vigna}@cs.ucsb.edu`

Abstract. The steadily increasing number of malware variants is a significant problem, clogging the input queues of automated analysis tools. The generation of malware variants is made easy by automatic packers and polymorphic engines, which produce by encryption and compression a multitude of distinct versions. A great deal of time and resources could be saved by prioritizing samples to analyze, either, to avoid the repeated analyses of variants and focus on innovative malware, or, on the contrary, to re-analyze variants and have better insights on their evolution. Unfortunately, indexing in malware analysis tools and repositories relies on executable digests (hashes) that strongly differ for each variant.
In this paper, we present a robust filter to quickly determine when a malware program is similar to a previously-seen sample. Compared to previous work, our similarity measure does not require the costly task of preliminary unpacking, but instead, operates directly on packed code. Our approach exploits the fact that current packers use compression and weak encryption schemes that do not break, in the packed versions, all the similarities existing between the original versions of two programs. In addition, we introduce a packer detection technique that is able to distinguish between different levels of protection, such as unpacked, compressed, encrypted, and multi-layer encrypted code. This allows us to optimize the sensitivity of the similarity measure accordingly. We evaluated our approach on a large malware repository containing 795,000 samples. Our results show that the similarity measure is highly effective in filtering out malware variants, even after re-packing, and can reduce the number of samples that need to be analyzed by a factor of 3 to 5.

1 Introduction

Malware authors release an ever-increasing number of malware samples. Overwhelmed by the quantity (up to several thousands per day), malware analysts cannot rely on manual analysis to examine the characteristics and behavior of new malware samples. As a result, analysts use automated dynamic analysis tools such as *Anubis* [1], *CWSandbox* [2], *Norman Sandbox* [3], or *ThreatExpert* [4]. These tools monitor the execution of malware samples in a controlled environment and provide a detailed report of their activity (*e.g.*, interactions with processes, files, the registry, or the network). The drawback of the dynamic approach mainly lies in the execution time, especially considering that the instrumented environments used to confine malware are usually slower than “real” execution environments. A minimal execution time is hard to determine, but, usually, it takes several minutes before a malware sample performs enough suspicious operations to allow for a correct characterization of its behavior, plus the time necessary to revert the instrumented environment in a clean state.

Throwing more hardware at the problem offers only temporary relief, considering the growing number of variants produced by malware authors. Moreover, this approach is wasteful, as a majority of the released malware samples are simple variations of existing ones. It would be preferable to manage analysis priorities and spend the available resources, either on previously unseen malware, or, on similar variants to obtain insights on their evolution, such as finding new control servers for bots [19]. To this end, a technique is needed to quickly determine whether a submitted sample is similar to one that was analyzed before.

Different static approaches have been explored to address the problem of malware similarity. In [10], the authors introduce a distance-based approach that uses the edit distance between instruction sequences, whereas the approaches described in [13] and [26] rely on the cosine vector distance over n-gram distributions of instructions. Other approaches replace the one-to-one distance function with more complex classification algorithms [18, 21, 25]. In [7] and [11], the authors introduce a graph-based approach, which compares graph representations extracted from the disassembled code. Some of these systems are computationally expensive. More importantly, all these previous approaches require that the malicious code is *unpacked* and *disassembled* first. Unfortunately, existing generic unpackers rely on a dynamic instrumentation of executables [12, 17, 22], and thus also suffer from performance limitations due to the code execution.

In this paper, we present an efficient, static technique that can identify samples that are similar to those previously analyzed, without the need to execute them. That is, our similarity measure is directly computed over packed and encrypted samples. This is possible because existing packers and their compression/encryption algorithms retain some of the properties present in the original code. Thus, two packed executables, produced by a certain packer, are likely to remain similar (in certain ways) if they were originally similar. The work closest to our proposed filter is *peHash* [27], a system that also attempts to detect duplicate malware samples without executing them. To this end, *peHash* leverages the structural information extracted from malware samples, such as the number, size, permission settings, and Kolmogorov complexity of the sections in a PE executable.While this approach makes *peHash* efficient, its reliance on ephemeral features is not robust, and it can be trivially confused. Our similarity measure, on the other hand, is based on properties directly derived from the code (content) of the malware program, and hence, is more tamper resistant.

To summarize, our contributions are the following:

– We introduce an efficient and robust similarity measure for malware samples. The measure operates directly on the packed code section(s) of the program.

– We present a packer detection method that can also identify the type of algorithm: compression, encryption, multi-layer encryption. The detected type is used to automatically configure the sensitivity of the similarity measure.
– We discuss prefiltering methods to select samples candidate for comparison. Prefiltering relies on efficient heuristics to quickly discard irrelevant samples.
– We have evaluated our techniques over a large malware repository (795,000 samples). Our experiments demonstrate that our similarity measure is effective in filtering out packed variants obtained from the an original malware. The system reduces the number of samples to analyze by a factor of 3 to 5.

2 Similarity and packing

Techniques to compute the static code similarity between malware samples face the same problems as static malware detection techniques: the packers and mutation engines that are widely used by malware writers to evade signature detection also blur the similarity between malware variants. According to [17], the percentage of malware that is packed has grown steadily, up to more than 80% of the samples currently found in the wild. Packers were first used to reduce the size of executables by compression. To hinder reverse engineering further, encryption was soon combined to compression. Encryption makes unpacking more difficult and, from the point of view of the malware authors, it increases the number of variants that can be generated by simply changing the encryption key. Recently, protections based on virtualization were introduced, where the original program is translated into virtual instructions that are then executed by an embedded virtual machine. In this work, we did not try to address virtualization-based packers, such as *Themida* or *VMProtect*, because they have complete control over the mapping between real and virtual instructions. In these conditions, code similarities at the binary level are hard to preserve.

In general, compression and encryption severely hinder any similarity computation on executables because the content of code sections is modified, both in terms of byte sequences and statistical properties. To better understand the ways in which current packers modify the body of an executable, we manually examined (reverse engineered) a number of popular tools frequently used by malware authors. A first, important observation is the limited number of algorithms that are at the core of current packers. For compression, dictionary-based approaches are the most widely used (mostly *LZ77*), sometimes combined with entropy and range encoders. For encryption, reversible arithmetic operations (such as *add/sub*, *rol/ror*, *xor* with 8-bits or 32-bits keys) are the most commonly used techniques. The use of stronger cryptographic algorithms (such as *RC4*, *DES*, or *AES*) is rare because these algorithms are much slower and need to be reimplemented to avoid using easily-detected cryptographic APIs.

Table 1 presents the key algorithms used by packers, as well as their impact on the data contained in the sections of the program: ‘alignment" indicates whether a byte-aligned data block remains aligned after packing, "sequences" discusses the effects of packing on the order of bytes (or regions) in the original program,

Table 1. Impact of the different packing algorithms on the binary content.

Dictionary Compression **e.g.** *LZ77* is used in *LZO*(*PolyEnE*), *NRV*(*UPX*).	Principle	Most-frequent bytes (or blocks) are replaced by relative references to previous occurrences.
	Alignment	Byte alignment is preserved by uncompressed blocks and references.
	Sequences	Order of incompressible blocks is preserved. Relative references are interleaved in between; their inter-space is bound by the size of the reference window.
	Distribution	Distribution is flattened because frequent blocks are replaced by references that tend to introduce infrequent byte values.
Entropy Encoding **e.g.** *Huffman* is combined with *LZ77* for *deflate*(*gzip*).	Principle	Most-frequent bytes (or blocks) are encoded by symbols (bit strings) of smaller size.
	Alignment	Byte alignment is destroyed by the shortened symbols.
	Sequences	Byte blocks are replaced by shorter symbols but their sequence is preserved.
	Distribution	Distribution is destroyed due to lost alignment, but can be reconstructed over the encoded symbols.
Range Encoding **e.g.** Encoding is combined with *LZ77* for *LZMA*(*NsPack*).	Principle	Most-frequent bytes (or blocks) are replaced by a single integer range representation.
	Alignment	Byte alignment is destroyed by the shortened integer ranges.
	Sequences	Sequences are shortened, but order remains.
	Distribution	Distribution is destroyed due to lost alignment, but can be reconstructed over the encoded ranges.
Arithmetic Encryption **e.g.** *PolyEnE* uses 32-bit xor, add/sub, rot/rol for encryption.	Principle	Blocks of bytes are independently encrypted using reversible arithmetic operations.
	Alignment	Byte alignment is preserved by the key and blocks.
	Sequences	Sequences are preserved, except that blocks are replaced by their encrypted values.
	Distribution	n-gram distribution is permuted, where n is the size of the encryption block.
Key/Operation Variation **e.g.** *Yoda's Cryptor* uses a cycle of xor, add/sub, rot/rol.	Principle	A different encryption key/operation is used for each new block.
	Alignment	Byte alignment is preserved by the key and blocks.
	Sequences	If variation is cyclic, repeated blocks at the same relative position in the cycle have the same encrypted value, otherwise, sequences are lost by the variable encryption.
	Distribution	If variation is cyclic, the effect is identical to encryption of larger blocks (encryption block length is equal to the cycle length).
Multi-layer Encryption **e.g.** *tElock* uses multiple layers of 8-bit xor encryptions.	Principle	The entire input is encrypted multiple times.
	Alignment	Byte alignment is preserved by the key and the byte blocks.
	Sequences	If layers are aligned with same size of encryption blocks, effect is equivalent to a single encryption, otherwise, overlaps are equivalent to key variations.
	Distribution	If layers are aligned with same size of encryption blocks, effect is equivalent to a single encryption, otherwise, overlaps are equivalent to key variations.

"distribution" characterizes how the distribution of bytes (or n-grams) in the original binary is altered by the packing process. By combining compression and encryption, packers tend to destroy all similarity between an original executable and its packed version. However, looking closer at Table 1, we observe that some information is preserved both by compression and weak encryption algorithms.

Compression Algorithms. Dictionary-based packers preserve certain incompressible parts of the original program, while they compact other parts by replacing entire sub-sequences with references (relative offsets) to previously-seen, uncompressed occurrences of the same sub-sequence. If two executables are similar before packing, their incompressible parts will be mostly similar. As for the compressed parts, one can expect that most of the relative offsets point to similar positions. Dictionary-based algorithms align both uncompressed regions and offsets on byte boundaries. Entropy encoders operate by replacing frequent bytes (or blocks) with shorter symbols. The lengths of these symbols are typically not a multiple of a byte, and hence, the byte alignment is destroyed. This also significantly alters the byte distribution. However, when considering the encoded symbols, their distribution is identical to the byte (block) distribution of the data before encoding. Similar considerations hold for range encoders.

Encryption algorithms. The reversible arithmetic operations used by a majority of crypters only achieve a simple substitution of the byte blocks in the original code. Arithmetic encryption results in a permutation of the distribution

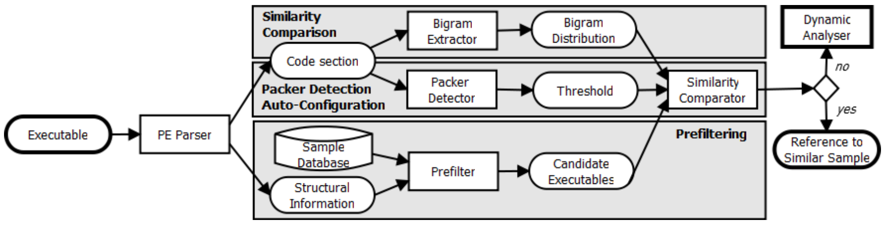


Fig. 1. Architecture of our similarity filter.

of the original n-grams but the alignment of bytes is not affected. Most of these crypters do not implement any chaining to strengthen their algorithm. Based on the packers we have studied, only a few crypters were actually offering position-dependent encryptions: these crypters apply a short cyclical variation of the key or the arithmetic operation but no chaining.

The important conclusion that can be drawn from the previous observations is that packers preserve certain properties of the original code. Compressors tend to alter the byte alignment. However, when considering the compressed symbols, some sequences are incompressible by dictionary-based compression and references to compressed sequences are deterministically determined. Compressors thus preserve similarity because originally similar programs result in similar compressed data, and, consequently, similar symbol distributions. Crypters do not alter the byte alignment. However, they create a permutation of the distribution of bytes (blocks) in the original program, where the permutation depends on the encryption key. In the next section, we will discuss how we can leverage these insights to perform similarity computations directly on packed code.

3 A similarity measure for packed executables

The filter that we introduce in this paper is designed to detect malware programs that are similar to previously-seen samples. Leveraging this filter, we can prioritize submissions to dynamic analysis systems according to the samples novelty.

To compute the similarity between two (malware) programs, we compute the distance between their *code signals*. A code signal is a bigram distribution over the raw bytes of the code section, but extracted in a way to compensate for the modifications (noise) introduced by packers (Section 2). An overview of the system is presented in Fig. 1. We keep a database of previously-analyzed malware programs that stores, for each sample, its code signal. Whenever a new sample arrives, its code signal is extracted. We then compute the distance of this signal with respect to those stored within the database (Section 3.1). If this distance is below a certain threshold, a similar sample already exists in the database, and no further analysis is performed. If the distance is above the threshold, the sample is submitted for further analysis and the database is updated accordingly.

To increase the speed and precision of the system, two additional steps are introduced. First, we use a packer detector that automatically configures the distance sensitivity, based on the type of packing used (Section 3.2). Second, the distances are not computed for all samples in the database. Instead, a prefilter selects likely candidates to reduce the number of comparisons (Section 3.3).

3.1 Extracting and comparing code signals

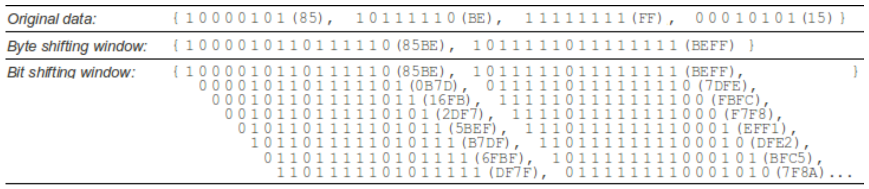
We employ *code signals* to characterize the executable section of a binary, and to determine the distance of binaries, from one to another. A code signal is a distribution of byte bigrams (pairs of subsequent bytes), extracted in a particular way from a program's code segment. One reason for operating directly on the raw bytes of the malware code is speed. Neither disassembly nor any other interpretation of the bytes is required. A second reason is that the similarity measure must be packer-agnostic, meaning that the measure should work directly on the packed code, which cannot be disassembled. To handle packed code, we introduce two specific transformations during the code signal extraction.

Extracting code signals. As discussed in Section 2, when similar programs are compressed or encrypted by current packers, the resulting binaries share certain similarities that "shine through" the packing process. We exploit these similarities using two transformations to respectively address the previously-identified problems of alignment destruction and distribution permutation.

1) Bit-shifting window: To recover from the destruction of the byte alignment, a bit-shifting window is used to extract bigrams, instead of the traditional byte-shifting window. The process is shown in Fig. 2. Using a byte-shift, any local difference between two similar streams of compressed data is likely to result in a disalignment because compressed symbols have sizes that are not byte-aligned. The importance of the bit-shift thus lies in its capacity to resynchronize two similar compressed streams with the correct alignment.

2) Sorted distribution: Once all bigrams are extracted from a malware's code section, we compute the bigram frequencies. Their distribution is then normalized by dividing these frequencies by the total number of bigrams in order to obtain a probability distribution. To address the possible, additional encryption of the code by simple arithmetic operations, the distribution is sorted by decreasing order of probability values. As mentioned in Section 2, for simple block encryption algorithms (without chaining), the n-gram distribution of the encrypted code is simply a permutation of the original distribution; in these cases, sorting the bigram distribution can perfectly recover the similarity between samples that was obscured by encryption. This technique was originally introduced in anomaly detection to detect similar attack payloads, possibly encrypted [15].

In our case, only a partial recovery of the distribution is possible because of the bit-shifting window used to extract bigrams: the bit-shifting is required to handle compression (and the alignment issues compression introduces). However, looking at the extracted bigram distributions, we find that only a small fraction of bigrams are frequent enough to contribute significantly to the code distribution (these are predominant bigrams). The remaining bigrams have a very small



```
Original data:          { 10000101 (85),  10111110 (BE),  11111111 (FF),  00010101 (15) }
Byte shifting window:   { 1000010110111110 (85BE),  1011111011111111 (BEFF) }
Bit shifting window:    { 1000010110111110 (85BE),  1011111011111111 (BEFF),                 }
                           0000101101111101 (0B7D),  0111110111111110 (7DFE),
                            0001011011111011 (16FB),  1111101111111100 (FBFC),
                             0010110111110101 (2DF7),  1111011111111000 (F7F8),
                              0101101111101011 (5BEF),  1110111111110001 (EFF1),
                               1011011111010111 (B7DF),  1101111111100010 (DFE2),
                                0110111110101111 (6FBF),  1011111111000101 (BFC5),
                                 1101111101011111 (DF7F),  0111111110001010 (7F8A)...
```

Fig. 2. Bigram extraction by bit shifting window.

Let us consider a 4-byte value $X = X_1X_2$. X is encrypted by a function E as follows:
$X' = E(X, K)$ with $K = K_1K_2$ and $X' = X'_1X'_2$.

E **is xor:** Relation between encrypted values and inputs:
$X'_1 = X_1 \oplus K_1$ and $X'_2 = X_2 \oplus K_2$
No diffusion between bits of of X_1 and bits of X_2.
E **is addition:** Relation between encrypted values and inputs:
$X'_1 = X_1 + K_1 + carry$ and $carry, X'_2 = X_2 + K_2$
If $carry = 0$, no diffusion between upper bits of X_1 and lower bits of X_2.
If $carry = 1$, only the rightmost bits of X_1 are impacted by the encryption of X_2.
E **is rotation:** Rotation diffuses overflowing bits from one side to the opposite side.
Still, particular keys do not properly achieve diffusion:
If $K = \alpha 16$ and α is even then: $X'_1 = X_1$ and $X'_2 = X_2$
If $K = \alpha 16$ and α is odd then: $X'_1 = X_2$ and $X'_2 = X_1$

Fig. 3. Diffusion between upper and lower bytes for arithmetic encryption operations.

probability compared to these frequent (predominant) bigrams, and they are in the long tail of the distribution. As expected, the predominant bigrams are those bigrams that are aligned on instruction boundaries or on the boundaries of compressed/encrypted symbols. The bigrams with small frequencies typically correspond to bigrams that overlap adjacent instructions or symbols.

In our experiments, we found that only about 7% of the bigrams are frequent enough to contribute to the code distribution. If we restrict our view of the bigram distributions to these predominant bigrams, the sorting process is still efficient in recovering the significant part of the distribution. In cases where the size of the encryption blocks is equal to or smaller than the size of the bigrams, the predominant bigrams are simply permuted. If blocks are of a larger size than bigrams, the quality of the recovery for these predominant bigrams depends on the extent to which the encryption operation on a large block can be approximated as separate (independent) encryption operations on sub-blocks. When this approximation holds, the original probability of a bigram is divided between a limited number of encrypted bigrams, depending on the relative position of the bigram to the key. For example, the original probability of a bigram X, after encryption by xor with a 32-bit key $K = K_1K_2$, will be always divided between $X \oplus K_1$ and $X \oplus K_2$. The approximation in separate encryptions actually depends on the diffusion achieved by encryption between the bits of different sub-blocks. Fig. 3 discusses different conditions under which certain arithmetic operations do not achieve diffusion.

This technique is designed to address the simple encryption algorithms used by current packers. On the other hand, it does not the address standard encryption algorithms, such as *AES* or *RSA*, used in contexts where the security of the data is critical and stronger cryptography is required.

Comparing code signals. The comparison between code signals is performed using Pearson's χ^2 test:

$$\chi^2 = \sum_{i=0}^{2^{16}-1} \frac{(o_i - r_i)^2}{r_i} \; where\; r_i > 0 \tag{1}$$

where o_i are elements of the distribution extracted from the submitted sample, and r_i are elements of the reference distribution from the candidate samples. The cases where $r_i = 0$ were ignored since, as previously seen, they correspond to negligible bigrams that are not contributing significantly to the distribution.

We did consider a number of other similarity measures (such as the cosine vector distance), but we found that the χ^2 test yielded the best results in terms of precision and performance. Moreover, we investigated weighting mechanisms, such as the *inverse document frequency*. Unfortunately, such mechanisms do not improve the results since compression and encryption make any *a priori* hypothesis about the statistical frequency of bigrams unreliable.

The χ^2 measure is computed between the distributions of the submitted sample and the first candidate. If the test value remains below a given threshold τ, the two samples are considered similar. Otherwise, the test is repeated with the next candidate. Whenever a similarity is found with one of the candidate samples, the comparison process is stopped, and the reference to the existing sample is returned. If no similarity is found, the comparison process is continued until the set of candidate samples is exhausted.

The actual value for the threshold τ is selected based on two factors. First, the threshold provides a mechanism to adjust the sensitivity of the filter, and hence, to control the trade-off between false negatives and false positives. In our use case, a false negative (failing to recognize that a similar sample is already in the database) is much less problematic than a false positive (incorrectly concluding that a similar sample is already in the database). This is because, in case of a false negative, a duplicate sample is analyzed, which results in a small waste of resources. In case of a false positive, a new, and possibly interesting sample, is incorrectly discarded. The second factor is the output of the packer detector (discussed in the next section). We use a set of different thresholds that are optimized according to the packing level of the tested program.

3.2 Packer detection

As explained in Section 2, packers modify the byte distribution of the code. In particular, packing often leads to a "flatter" distribution. In case of compression, frequent values are replaced by references or short symbols. In case of encryption, the same, frequent byte value might be mapped to different, encrypted values. Flatter distributions can lead to false positives, because the similarity values returned by the χ^2 test decrease (compared to unpacked samples). The similarity threshold should thus be reduced accordingly when checking packed executables.

Fig. 4. Statistical tests for packer detection.

T_1: *Uncertainty test.* Code entropy.
T_2: *Uniformity test.* χ^2 test between the code and an equiprobable distribution.
T_3: *Run test.* Longest sequence of identical bytes in the code.
T_4: *1st-order dependency test.* Autocorrelation coefficient of the code at lag 1.

Type	Test series	Detection criterion
Packers	$T_1 : H \geq t_1$	`packed` if T_1 = true
Compressors	$T_2 : U < t_2$ $T_3 : lgth(run) \leq t_3$ $T_4 : \|ACC\| < t_{4a}$	`compressed` if `packed` $\wedge$ no more than one of T_2,T_3,T_4 = true
Crypters	$T_2 : U < t_2$ $T_3 : lgth(run) \leq t_3$ $T_4 : \|ACC\| < t_{4a}$	`encrypted` if `packed` $\wedge$ two or more of T_2,T_3,T_4 = true
Multi-layer crypters	$T_4 : \|ACC\| < t_{4b}$	`multi-layer` if `encrypted` $\wedge$ T_4 = true

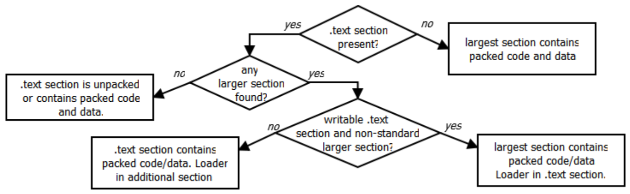
To detect packed executables, we leverage the fact that a flattened distribution makes packed code similar to random data. Thus, the statistical properties used to assess random generators can be used to detect packed executables, and classify their type of protection: compression, single and multi-layer encryption.

Packer detection and classification. To detect packers and to identify the type of protection, we introduce four statistical tests in Fig. 4. These tests are performed over the raw bytes in the actual code section or the packed code section (depending on whether the sample is packed).

The entropy-based test T_1 is the traditional test used to detect packed executables. A high entropy value constitutes a significant sign of randomness. Thus, whenever T_1 yields a code entropy value above an experimentally determined threshold, the sample is considered packed. For all packed samples, we use three additional tests T_2, T_3, and T_4 to determine more precisely the type of packing. These tests were originally designed for assessing random number generators [23]. Here, we apply them in a novel context.

The uniformity-based test T_2 and the run-based test T_3 are primarily employed to distinguish between compressed and encrypted code. When an encryption algorithm uses input blocks that span multiple bytes, one particular (byte) value in the original code is likely mapped to several different, encrypted values in the packed code, depending on the relative positions of the bytes in the encrypted block. Thus, the distribution of encrypted code is closer to a uniform distribution than compressed code (a larger specter of observed bigrams with a levelled frequency), a property checked by T_2. Moreover, some compression algorithms (especially dictionary-based approaches) can produce sequences (runs) of identical bytes, something that is unlikely for crypters. As a result, the presence of longer runs of identical bytes is an indication of compression.

Finally, executable code is known to have a first-order dependency [21]. This dependency between consecutive bytes is partially destroyed by compression and encryption. In the case of multi-layer crypters, the boundaries between different layers introduce additional discontinuities. These discontinuities are detected by testing the autocorrelation coefficient (ACC) of the code T_4. Fig. 4 explains how our four tests are combined to identify the level of packing. Thresholds t_1, t_2, t_3, t_{4a}, t_{4b} are experimentally determined in the evaluation section.



.text section present?
yes
no
largest section contains packed code and data
any larger section found?
no
yes
.text section is unpacked or contains packed code and data.
writable .text section and non-standard larger section?
no
yes
.text section contains packed code/data. Loader in additional section
largest section contains packed code/data Loader in .text section.

Fig. 5. Finding the plain/packed code section.

Locating packed code. The four statistical tests have to be performed on the "normal" text (code) segment for unprotected executables or on the section that holds the packed code. Since packers modify the sections of executables, the risk is to perform the test over the section that contains the loader. Fig. 5 shows our heuristic to find the section that contains the packed data. Notice that the identified section is later used to extract the code signal.

3.3 Fast prefilter

Computing the similarity of a new malware sample with respect to those already stored in the database is potentially costly when the number of samples increases. To reduce the necessary similarity computations, but also to reduce potential false positives due to random collisions between code signals, we apply a prefilter to select only a subset of candidate samples for further consideration. This prefilter step uses fast heuristics to discard non-similar samples based on straightforward observations. More precisely, the prefilter uses two sequential heuristics: a first heuristic based on the size of samples, and a second heuristic based on the structural information contained within the programs' PE headers.

Size-based filtering. An immediate criterion of similarity between PE executables is their size. When malware writers produce variants of their original code, these variants tend to be of similar size. Of course, the size of samples derived from the same original source code might change because of compilation parameters, small modifications to the code, and, most importantly, because of packing. Taking into account these factors, we compute, for a new sample, a range with limits that are a fixed percentage above and below this sample's size. The prefilter then selects candidate samples whose size falls within this range. If no candidate is found, the sample is considered new.

PE-characteristic-based filtering. Further criteria of similarity between executables are their structural characteristics. In the PE format, the header contains important information about the executable's layout, both on disk and in memory, and meta-information about the compilation process. However, only a subset of these features is useful for prefiltering: we only consider features that provide sufficient differentiation between executables while being robust to packing (that is, features that are not modified by packers). The 16 features we selected are presented later in Table 5. The prefilter computes the Hamming distance between the PE features of a new (incoming) sample and the features of all candidates that were selected by the first heuristic. When the distance is larger than a threshold, the corresponding database sample is discarded. All remaining samples become candidates for the similarity measure computation.

4 Evaluation

The filter presented in previous sections was implemented and used to process samples submitted to an automated, dynamic malware analysis system. The evaluation was carried out in two steps. For the first step, we used our filter on known samples for which ground truth was available (Section 4.1). The goal of this first step was to establish the similarity thresholds and configure the packer detector as well as the prefiltrer. For the second step, we applied the filter to a large collection of malware samples that were provided to us by the authors of *Anubis* [1] (Section 4.2). The goal of this second step was to verify that the precision is maintained in real-world conditions, when the filter is exposed to a large number of diverse malware samples and packers. We also took advantage of this second step to study the scalability and the robustness of our approach.

4.1 Experiments on known samples

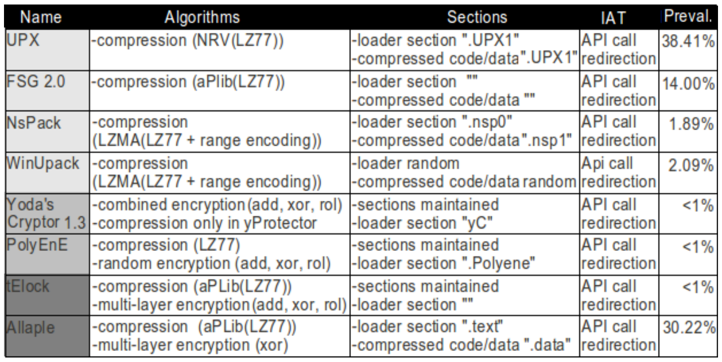
We started our experiments with two data sets. The first set, S_1, contained 384 PE executables, mostly taken from the system directory of a *Windows XP* installation. It also contained open-source software, such as *OpenOffice*, and free shareware, such as *mIRC*. All programs in S_1 were unpacked and served as examples of dissimilar (unrelated) binaries.

The second set, S_2, contained 65 bots, whose source code was made available to us. These bots belong to two malware families: *SdBot* (23 samples) and *rBot* (42 samples). The *SdBot* samples were further classified as versions 4 and 5, while the *rBot* samples span five versions ranging from 3 to 7. Since the samples in S_2 are related to various degrees, we could leverage this data set as labeled ground truth to study the precision of our similarity measure. Any other malware family with a version history could have been used for this configuration.

Packer detection. To assess our packer detection technique, we selected seven packers, based on their popularity with malware writers: *UPX*, *FSG*, *NsPack*, *WinUPack*, *Yoda's Cryptor*, *PolyEnE* and *tElock*. We also added instances of the *Allaple* worm as a representative example for polymorphic malware; its engine uses techniques similar to packing. Table 2 provides an overview of these packers, covering the compression and encryption algorithms they implement: four compressors, two crypters, and two multi-layer crypters. Looking at prevalences, these eight packers cover 86% of the packed samples from the *Anubis* data set.

We first packed each of the 384 executables from S_1 with the 7 packers, and then added 120 *Allaple* samples. This set of packed executables was used to verify the rate of False Negatives (FN) of the technique. The unpacked versions of these executables were then included to the data set in order to verify the rate of False Positives (FP). Training over a small subset of this data set, we obtained the

Table 2. Specifications of the tested packers.



Name	Algorithms	Sections	IAT	Preval.
UPX	-compression (NRV(LZ77))	-loader section ".UPX1" -compressed code/data".UPX1"	API call redirection	38.41%
FSG 2.0	-compression (aPlib(LZ77))	-loader section "" -compressed code/data ""	API call redirection	14.00%
NsPack	-compression (LZMA(LZ77 + range encoding))	-loader section ".nsp0" -compressed code/data".nsp1"	API call redirection	1.89%
WinUpack	-compression (LZMA(LZ77 + range encoding))	-loader random -compressed code/data random	Api call redirection	2.09%
Yoda's Cryptor 1.3	-combined encryption (add, xor, rol) -compression only in yProtector	-sections maintained -loader section "yC"	API call redirection	<1%
PolyEnE	-compression (LZ77) -random encryption (add, xor, rol)	-sections maintained -loader section ".Polyene"	API call redirection	<1%
tElock	-compression (aPLib(LZ77)) -multi-layer encryption (add, xor, rol)	-sections maintained -loader section ""	API call redirection	<1%
Allaple	-compression (aPLib(LZ77)) -multi-layer encryption (xor)	-loader section ".text" -compressed code/data ".data"	API call redirection	30.22%

Table 3. Detection/classification of packers.

Name	Unpacked	Packed	Compr.	Crypt.	MLCrypt.
Unpacked	99.74%	00.26%	00.26%	00.00%	00.00%
FSG	18.18%	81.82%	81.55%	00.00%	00.27%
UPX	03.04%	96.94%	96.10%	00.56%	00.28%
NsPack	12.11%	87.89%	87.63%	00.26%	00.00%
WinUPack	13.84%	86.16%	83.55%	02.09%	00.52%
Compressors	11.80%	88.20%	87.21%	00.72%	00.27%
YodaCryptor	17.99%	82.01%	06.08%	74.87%	01.06%
PolyEne	06.01%	93.99%	28.98%	62.14%	02.87%
Crypters	12.00%	88.00%	17.53%	68.51%	01.96%
tElock	04.84%	95.16%	00.57%	70.94%	23.65%
Allaple	00.00%	100.0%	00.00%	72.22%	27.78%
Multi-layers	02.42%	97.58%	00.28%	71.58%	25.72%
Packed	08.74%	91.26%	N/A	N/A	N/A

following thresholds that optimize the trade-off between FN and FP: $t_1 = 4.73$, $t_2 = 0.0012$, $t_3 = 2$, $t_{4a} = 0.005$, $t_{4b} = 0.002$ *c.f.* Fig. 4, Section 3.2.

The detection results for the remaining samples (test set) are presented as a confusion matrix in Table 3. One can see that the detector is able to distinguish very well between unpacked and packed executables: the detection rate for unpacked samples is 99.74%, while it is over 91% on average for packed programs. Furthermore, our statistical tests were able to correctly distinguish, in more than 80% of the cases, between compressors and crypters. The lowest classification rate was achieved for multi-layer crypters. The reason is that encrypting the same executable multiple times does not necessarily result in stronger encryption. In particular, several layers of *xor* encryption are basically equivalent to a single layer. It is important to observe though that a misclassification only leads to the use of a suboptimal threshold, but it does not prevent the system from computing correct similarity results.

When putting our detection results into the context of related work, our technique provides fine-grained distinctions between different types of packing without making use of packer-specific signatures or features. Systems such as [16] relying on pure entropy, or [20] relying on structural properties of the executable, only distinguish between packed and regular code. More advanced systems such as [8] or [24] can precisely classify packers by name using randomness profiles. However, these systems need to be trained for each individual packer that should be recognized, something that our system, providing a coarser-grained distinction, does not require because it relies on information theoretic metrics that extend to any other packer that uses similar algorithms.

Tuning the filter granularity. The goal of the next experiment is to select suitable filter thresholds. For this, we turned our attention to S_2, the set of 65 classified bot samples. More precisely, to build our configuration set, these 65 bots, together with all benign 384 programs from S_1, were packed with all seven packers and submitted to the filter.

$$TH = \frac{nb\ similar\ samples\ flagged\ as\ similar + nb\ unique\ samples\ flagged\ as\ unique}{nb\ submitted\ samples}$$

$$FH = \frac{nb\ dissimilar\ samples\ flagged\ as\ similar}{nb\ submitted\ samples}$$

$$M = \frac{nb\ similar\ samples\ flagged\ as\ dissimilar}{nb\ submitted\ samples}$$

Granularity levels:

$(f)-$ *two samples are similar if they belong to the same family*

$(v)-$ *two samples are similar if they belong to the same family and have the same version*

To measure the filter precision, we use the following metrics: (i) the rate of True Hits, TH, which correspond to the cases where the filter successfully discards similar samples, or forwards new, unique samples to the analysis tool; (ii) the rate of False Hits (or false positives), FH, which correspond to the cases where new samples are discarded even though they are novel (these errors are critical, because they may result in a loss of interesting information); (iii) the rate of Misses (or false negatives), M, which correspond to cases where samples are forwarded to further analysis even though they should have been discarded (these errors are less severe, because they only result in unnecessary analyses).

In Table 4, we present the results of our experiments for two different sets of thresholds. The first set of thresholds corresponds to what we refer to as *family granularity*. That is, the thresholds are set with the aim of recognizing as similar two samples when they belong to the same malware family. That is, a sample that belongs to *rBot* version 5.0 should be considered similar to an *rBot* version 6.0. The thresholds were found by an optimization process that maximizes the rate of true hits while maintaining the rate of false hits under 0.5%. The rate of false hits is the most critical factor because it eventually corresponds to the potential loss of information we tolerate by not running a unique sample.

With family granularity, we observe 95.2% of true hits on average, with only 4.5% misses and, more importantly, only 0.3% of false hits. The rate of true hits indicates to which extent similarity is preserved by packers, even after the minor modifications brought to the code of the different versions. Unsurprisingly, the best results are observed for compressors, because their packing process is deterministic. On the other hand, the filter does not achieve 100% detection in the case of crypters because the size of the encryption key is typically 32-bits, which is twice the bigram size. In this case, as we have seen, the similarity preservation depends on some bias of the encryption algorithm. The worst results are obtained for Yoda's Cryptor, because this crypter uses a cycle of different en-

Table 4. Precision of the similarity measure for various packers.

	Family granularity thresholds							Version granularity thresholds			
Packer	Thrsh.	TH(f)	FH(f)	M(f)	TH(v)	FH(v)	M(v)	Thrsh.	TH(v)	FH(v)	M(v)
None	0.0020	99.8%	00.2%	00.0%	94.2%	05.8%	00.0%	0.0012	98.0%	00.2%	01.8%
FSG	0.0018	99.6%	00.4%	00.0%	91.5%	08.5%	00.0%	0.0008	94.2%	00.4%	05.4%
UPX	0.0018	91.8%	00.2%	08.0%	89.9%	02.1%	08.0%	0.0008	91.1%	00.4%	08.5%
NsPack	0.0018	99.4%	00.2%	00.4%	93.6%	06.0%	00.4%	0.0008	94.7%	00.2%	05.1%
WinUPack	0.0018	99.2%	00.4%	00.4%	93.6%	06.0%	00.4%	0.0008	94.7%	00.2%	05.1%
YodaCryptor	0.0015	89.3%	00.0%	10.7%	90.4%	00.2%	09.4%	0.0006	90.2%	00.0%	09.8%
PolyEne	0.0015	90.0%	00.4%	09.6%	90.6%	01.2%	08.2%	0.0006	89.8%	00.4%	09.8%
tElock	0.0013	96.1%	00.6%	03.3%	95.1%	02.9%	02.0%	0.0004	91.8%	00.2%	08.0%
Allaple	0.0013	92.2%	00.0%	07.8%	82.2%	10.0%	07.8%	0.0004	76.6%	00.0%	23.4%
Average	-	95.2%	00.3%	04.5%	91.3%	04.7%	04.0%	-	91.3%	00.2%	08.5%

Table 5. PE Header characteristics selected for comparison.

Location	Name	H	*card*	Name	H	*card*
DOS Header	`AddressNewExeHeader`	1.87	13			
NT Header	`Characteristics`	0.67	7			
Optional Header	`(min/maj)LinkerVersion`	0.68	6	`CodeBase`	0.93	6
	`ImageBase`	0.44	5	`(min/maj)OSVersion`	0.43	4
	`(min/maj)ImageVersion`	0.46	4	`(min/maj)SubsystemVersion`	0.45	4
	`Subsystem`	0.22	2	`DllCharacteristics`	0.75	7
	`SizeStackReserve`	0.31	4	`SizeStackCommit`	0.44	5

cryption operations and keys per block. The cycling operations make encryption position-dependent, thus explaining the higher rate of misses.

Depending on the desired level of granularity, it might be preferable to analyze different versions of the same malware family. In this case, the family granularity thresholds are too loose. This can be seen by looking at the false hit rates for malware versions, denoted as $FH(v)$, which reaches 4.7% when using family granularity thresholds. To differentiate between different malware versions, we created a second set of tighter thresholds (referred to as *version granularity*). It can be seen that, using these thresholds, $FH(v)$ drops to 0.2%. However, we also have to accept that the rate of misses increases. Notice that misses remain tolerable because they only imply re-running an existing sample, without potential loss of interesting information.

We also examined the precision of our system when analyzing the polymorphic worm *Allaple*, which was a major issue in 2007-2008, polluting malware repositories with thousands of mutated variants. The experiments have been run over two versions, namely *Allaple.b* and *Allaple.e*. The results are also given in Table 4. The worm variants are accurately detected in more than 92% of the cases, with a good distinction between versions.

Configuring the prefilter. The size range that constitutes the first heuristic of the prefilter was configured so that the variants of a given program fall within this range, while it remains tight enough so that the number of irrelevant candidates remain minimal. Considering the packed versions of the bot variants from S_2, the maximum size variation that we observed was 4.4%, which corresponds to a lower bound of 95.6% and a higher bound of 104.4% of the original size.

To find the best structural features to constitute the second heuristic of the prefilter, we again examined the original and packed versions of the samples from S_1. For all PE header fields, we verified that they were resilient to packing, and that they were distinguishing enough (sufficient number of different values, $card$, and high entropy, H). Table 5 provides the list of 16 selected features that are compared by Hamming distance with a threshold of 0.

4.2 Large scale experiments

The experiments with known samples allowed us to analyze the accuracy of our filter, tune detection thresholds, and configure the prefilter. In the next step, we performed a large-scale experiment with 794,665 malware samples that were submitted to the *Anubis* analysis tool in 2009. For each of these samples, we had at our disposal behavioral information (execution traces) and a reference clustering [6]. This clustering partitioned the malware programs into 91,522 different groups sharing similar runtime activity.

Precision and scalability. We applied our filter to the entire data set of almost 795 thousand malware samples. To evaluate the filter precision, we use the aforementioned metrics: True Hits (TH), False Hits (FH), and Misses (M).

A problem for this experiment was the fact that we did not have ground truth available (such as source code or reliable malware labels). To address this, we introduced a reference classification based on the behavioral and structural information of executables. More precisely, we leveraged the behavioral clusters [6]: we considered two samples as similar when they produced similar behaviors, and hence, ended up in the same behavioral cluster. The behavior similarity was computed using the Jaccard distance between their execution traces. Unfortunately, the execution of malware programs is not deterministic and can change depending on the environment, time, or the availability of network resources (such as C&C servers). As a result, similar samples might end up in different behavioral clusters. Thus, to improve the reference clustering, we also considered structural characteristics of the malware programs. More precisely, we checked whether the executable sections of two programs share the same name, size, position in memory, and hash of the sections' contents. We considered two samples as similar when at least 90% of their structural information is identical and they share more than 70% of their behavior.

Precision. Table 6 shows the filter precision for three sets of thresholds. The first two correspond to the thresholds for *family* and *version* granularities, respectively, while the third is an extra set with more conservative thresholds. These three sets represent different trade-offs between reducing unnecessary analysis runs (TH) and the risk of discarding potentially interesting samples (FH).

For the first thresholds, the filter achieves a true hit rate of more than 90%. That is, more than 90% of similar (irrelevant) samples are correctly discarded. This leads to a reduction of the amount of overall analysis runs by a factor of almost five – saving a significant amount of valuable resources. This is paid for by a false hit rate of 0.7%. When the thresholds are more conservative, the number of incorrectly discarded samples (FH) is reduced to 0.3%. This, however, also lowers the hit rate, and thus, the reduction factor that can be achieved.

Table 6. Filter accuracy for selected thresholds
(U-Unpacked, C-Compressed, E-Encrypted, MLE-Multi-Layer Enc.)

Similarity Thresholds				Family accuracy		Version accuracy		Misses	Reduction
U	C	E	MLE	TH(f)	FH(f)	TH(u)	FH(u)	M	Factor
0.0020	0.0018	0.0015	0.00130	91.1%	00.7%	89.8%	02.0%	09.2%	4.84
0.0012	0.0008	0.0006	0.00040	84.6%	00.5%	83.8%	01.3%	14.9%	3.79
0.0005	0.0003	0.0002	0.00008	74.4%	00.3%	74.0%	00.7%	25.3%	2.71

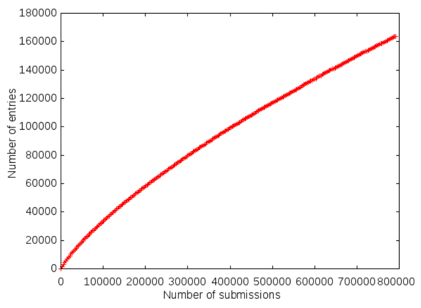


Fig. 6. Database growth.

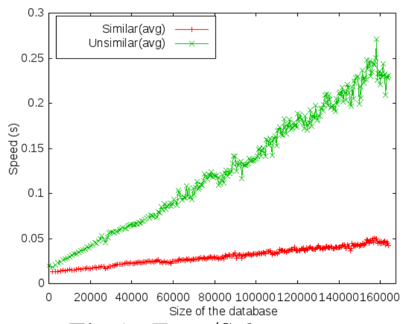


Fig. 7. Time/Submission.

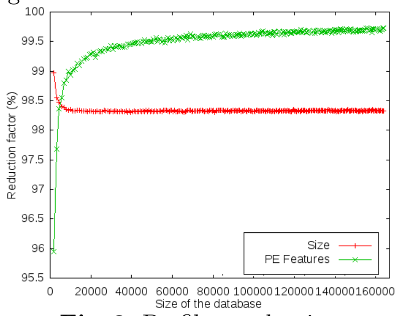


Fig. 8. Prefilter reduction.

We then analyzed the False Hits produced by our filter in more detail. We found that incorrect similarities can be explained either by the failure of the heuristic to find the section containing the packed code (∼10% of FH), or, in most cases, by the misclassification of samples that, although they belong to different families, are part of the same class of malware (∼90% of FH). The heuristic failed mainly on very small executables where the packed code was negligible compared to the loader code. The misclassification mainly happened for fake anti-virus software and IRC bots, probably because they share substantial portions of code. For this analysis, we used the malware labels produced by more than 40 AV scanners (run by *VirusTotal* [5]). We declared a false hit every time less than 5 scanners would agree on the family name.

With respect to misses, we found that most cases were caused by similar samples that exhibit similar dynamic behavior but were protected by different packers. For a given executable, the filter tends to create a new database entry for each different packer (type) used to protect this binary.

Scalability. To understand the scalability of our approach, we first examined the growth of the sample database. According to Figure 6, the database size increases sub-linearly with the number of submissions. Figure 7 shows a linear increase of the computation time with the number of entries. The computation time for similar samples is lower because, as soon as a similar entry is found, the computation stops. In the worst case, for unique samples, the filter takes no more than 300ms. This is 1,200 times faster than the 6 minutes required to execute a sample within *Anubis*. Considering the observed slowdown in the increase of the database size, the system should scale at least to tens of millions of samples.

The prefilter plays an important role in these performances. In Figure 8, it can be seen that the two heuristics reduce the candidate set to less than 1% of the database samples. Moreover, the figure shows that the prefilter maintains its effectiveness independently of the size of the database.

Robustness of the filter. In the next step, we compare our approach to existing techniques that aim to detect similarities between malware binaries without analyzing their runtime behavior. To this end, we reimplemented *peHash* [27]. This tool operates mostly on structural characteristics of malware samples, and hence, does not require unpacking or disassembling the code beforehand. To understand how much the precision of our filter suffers because it has to operate on packed code (bytes) instead of disassembled instructions, we implemented a second version of our filter, where the bigram distribution (code signal) is not computed over the raw (and possibly packed) bytes, but over bigrams of disassembled instructions. This technique is similar to *Vilo* [26]. Finally, to compare with an alternative approach to detect malware similarity, we used an existing tool that operates on control flow graphs [14].

In the following, we refer to the four systems under examination as: *Filter* for our tool, *peHash*, *Disasm* for the disassembled version of *Filter*, and *Graph*. To experiment with these systems, we selected a subset of 18,645 samples from our real-world data set, where the corresponding unpacked binaries were available, as a byproduct of their execution in a dynamic analysis environment.

Attacker model. Here, we assume an attacker who develops a packer that operates directly on executables. That is, the packer is given a binary, and it has to output variants that cannot be recognized as similar. This is realistic because malware authors typically distribute their malware programs as binaries, using third-party packers to produce new variants on the fly.

The need to operate on binaries imposes certain constraints; in particular, *the memory layout of the executable must be preserved.* Otherwise, addresses in the code or data sections would not resolve properly, and the program would crash. To work around this problem, the attacker would have to perfectly disassemble any input binary, which is extremely difficult in practice. Given this limitation, the attacker can perform structure-based operations leaving the original code untouched (1−4), and content-based operations that modify the code (5):

(1) Modifying access permissions of sections.
(2) Changing the size of sections on disk only.
(3) Injecting random data within the padding spaces.
(4) Appending sections at the end of the memory image.
(5) Compressing and/or encrypting code/data sections.

Table 7. Compared robustness to structural modifications.

Modifications	*peHash*	*Filter*
Section permissions	7.8%	99.8%
Size of sections	42.5%	98.4%
Random data	37.8%	80.8%
Appended sections	0.0%	84.6%

Table 8. Compared precision and runtime.

Systems	TH	FH	M	Time
No prerequisite on the code				
Distance-based(*Filter*)	80.8%	00.7%	18.5%	6 min
Hash-based(*peHash*)	81.1%	00.6%	18.3%	9 min
Unpacked and disassembled code (* without unpacking)				
Distance-based(*Disasm*)	84.3%	00.5%	15.2%	239 min*
Graph-based(*Graph*)	83.4%	00.4%	16.2%	847 min*

Table 9. Compared robustness summary.

Modifications	*Disasm*	*Graph*	*peHash*	*Filter*
Modifying section permissions	✓	✓	×	✓
Changing section sizes	✓	✓	×	✓
Injecting data in sections	✓	✓	×	*
Appending new sections	✓	✓	×	*
Compression	×	×	✓	✓
Arithmetic encryption	×	×	✓	✓
Chained encryption	×	×	×	×
Strong encryption	×	×	×	×

Structure-based robustness. To examine the techniques robustness to structure-based modifications, we developed an obfuscation tool that can apply all four structural modifications defined within our attacker model. Using this tool, we generated four kinds of variants for the 18,645 samples in our test set, and submitted them to *peHash* and our *Filter*. We did not test *Disasm* and *Graph* against the obfuscated binaries since these tools ignore structural information.

Table 7 presents the percentages of similar variants correctly identified for each type of modification. Overall, our approach is significantly more robust than *peHash*. This is not surprising since *peHash* focuses on structural information, which is easy to tamper with. Our *Filter*, on the other hand, relies on the statistical properties of the code, which are harder to change.

Table 7 also shows that our system considered as different a number of samples that should have been recognized as similar. The first, and main, reason was that the sizes of the binaries were changed by the obfuscator so that they exceeded the size range of the prefilter. To handle this issue, we can increase the size range that the prefilter accepts, at the expense of a small performance penalty. The second reason, far less frequent, was that our heuristic to identify the packed code section (see Figure 5) was misled.

Content-based robustness. All packers apply some form of content-based obfuscation, by compression or some simple form of encryption. Since *Disasm* and *Graph* work only on unpacked samples, such simple transformations would already be sufficient to render them useless. However, in this section, we explore the precision of these systems when operating on unpacked binaries, compared to our *Filter* and *peHash* that operate on the corresponding packed versions.

For this experiment, we submitted the packed and unpacked versions of our 18,645 samples to all four systems. Table 8 compares the results, both in terms of precision and runtime. *PeHash* performs quite similarly to our approach, but at the significant expense of structural robustness, as was discussed previously. *Disasm* and *Graph*, which operate on unpacked executables, do not achieve a significantly better accuracy; in fact, the overall differences are minimal. This is encouraging because it shows that our *Filter*, working on packed code, produces almost the same results as tools that require unpacking and disassembling the malicious code. Moreover, the runtime of these tools is an order of magnitude larger, even when the unpacking time is not included.

These satisfying results are mainly explained by the fact that packers still rely on weak encryption algorithms. These results may no longer hold if packers start using stronger encryption algorithms such as *AES* or *RSA*, or, at least, design more clever algorithms such as in the case of blending attacks [9]. Blending attacks manipulate content, starting from an initial attack payload, until the payload satisfies a given distribution. In our case, blending attacks could be used to craft similar malware code distributions, making the filter ineffective. In their paper, the authors suggest the possibility of crafting the distribution by substitution operations and padding. In our case, the padding is however limited by the boundaries of the binary sections. To conclude this discussion about the filter robustness, Table 9 provides a summary view that compares the robustness of the four different systems that we examined with respect to our attacker model: ✓ if the system is robust, ✗ otherwise. The stars (*) in the table correspond to modifications to which the system is not entirely robust.

5 Conclusion

In this paper, we introduced an accurate, robust, and efficient technique for detecting similarity between malware samples. We leverage the fact that current malware packers only employ compression and weak encryption, and, therefore, information about the original program can be extracted from a packed binary. Unlike previous work [7, 11, 13, 26], our technique is thus able to directly operate on packed binaries, avoiding the costly unpacking process. By doing this, our system is able to filter submissions to malware repositories or automated dynamic analysis tools. Large-scale experiments with almost 795,000 malware samples demonstrate that the filter achieves a significant reduction of the samples that need to be analyzed, with only a small amount of false positives.

References

1. ANUBIS. `http://anubis.iseclab.org`.
2. CWSandbox. `http://www.mwanalysis.org`.
3. Norman Sandbox. `http://www.norman.com/technology/norman_sandbox/`.
4. ThreatExpert. `http://www.threatexpert.com`.
5. VirusTotal. `http://www.virustotal.com`.
6. U. Bayer, P. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proc. Symp. Network and Distributed System Security (NDSS)*, 2009.
7. E. Carrera and G. Erdelyi. Digital genome mapping. In *Virus Bulletin*, 2004.
8. T. Ebringer, L. Sun, and S. Boztas. A fast randomness test that preserves local detail. In *Virus Bulletin*, 2008.

9. P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee. Polymorphic blending attacks. In *USENIX Security Symposium*, 2006.
10. M. Gheorghescu. An automated virus classification system. In *Virus Bulletin*, 2005.
11. X. Hu, T. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proc. ACM Conf. Computer and Communications Security (CCS)*, pages 611–620. ACM, 2009.
12. M. G. Kang, P. Poosankam, and H. Yin. Renovo: a hidden code extractor for packed executables. In *Proc. ACM Workshop Recurring Malcode (WORM)*, pages 46–53. ACM, 2007.
13. A. Karnik, S. Goswami, and R. Guha. Detecting obfuscated viruses using cosine similarity analysis. In *Proc. Asia Int. Conf. Modelling & Simulation (AMS)*, pages 165–170. IEEE Computer Society, 2007.
14. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proc. Symp. Recent Advances in Intrusion Detection (RAID)*, 2005.
15. C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *Proc. ACM Conf. Computer and Communications Security (CCS)*. ACM, 2003.
16. R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security and Privacy*, 5(2):40–45, 2007.
17. L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proc. Annual Computer Security Applications Conf. (ACSAC)*, pages 431–441, 2007.
18. R. Moskovitch, D. Stopel, C. Feher, N. Nissim, N. Japkowicz, and Y. Elovici. Unknown malcode detection and the imbalance problem. *J. Computer Virology*, 5(4):295–308, 2009.
19. M. Neugschwandtner, P. M. Comparetti, G. Jacob, and C. Kruegel. FORECAST – Skimming off the malware cream. In *Proc. Annual Computer Security Applications Conf. (ACSAC)*, 2011.
20. R. Perdisci, A. Lanzi, and W. Lee. Classification of packed executables for accurate computer virus detection. *Pattern Recognition Letters*, 29(14):1941–1946, 2008.
21. K. S. Reddy, S. K. Dash, and A. K. Pujari. New malicious code detection using variable length n-grams. In *Proc. Int. Conf. Information Systems Security*, LNCS vol. 4332, pages 276–288, 2006.
22. P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Annual Computer Security Applications Conference*, 2006.
23. A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical Report 800-22, NIST, 2001.
24. L. Sun, S. Versteeg, S. Boztaş, and T. Yann. Pattern recognition techniques for the classification of malware packers. In *Proc. Australian Conf. Information Security and Privacy*, LNCS, pages 370–390. Springer, 2010.
25. S. M. Tabish, M. Z. Shafiq, and M. Farooq. Malware detection using statistical analysis of byte-level file content. In *Proc. ACM SIGKDD Workshop CyberSecurity and Intelligence Informatics*, 2009.
26. A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia. Exploiting similarity between variants to defeat malware. In *Proc. BlackHat DC Conf.*, 2007.
27. G. Wicherski. peHash: A novel approach to fast malware clustering. In *USENIX Workshop Large-Scale Exploits and Emergent Threats (LEET)*, 2009.