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

Abstract

The steadily increasing number of malware variants is becoming a significant problem, clogging the input queues of automated analysis tools and polluting malware repositories. The generation of malware variants is made easy by automatic packers and polymorphic engines, which can produce many distinct versions of a single executable using compression and encryption. Malware analysis tools and repositories rely on executable digests (hashes) for indexing malware programs and discarding duplicates. Unfortunately, these executable digests are different for each malware variant. Thus, a great deal of time and resources are wasted by analyzing, running, and storing numerous instances of almost identical programs. To address this problem, we require a more robust similarity measure that can quickly identify and filter these variants, avoiding repeated (costly) analyses that provide no additional insights to a malware analyst.

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 is efficient because it 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 all connections between the original programs and their transformed version (that is, some indicators of similarity between two original programs can still be extracted from their packed version). 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 configure (optimize) the sensitivity parameter for the similarity computation. We performed experiments on a large malware repository containing 795 thousand samples. Our results show that the similarity measure is highly effective in filtering out malware variants obtained by simple re-packing or re-encryption, and can reduce the number of samples that need to be analyzed by a factor of three to five.

1 Introduction

Often motivated by financial gains, malware authors release an ever-increasing number of malware samples on the Internet. For example, in 2009, Panda Labs reported up to 55,000 new malware samples per day [19]. Overwhelmed by the quantity of new samples, malware analysts cannot rely on manual analysis to examine the characteristics and behavior of these malware samples. As a result, the 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 time required for the execution, especially considering that the instrumented environments used to confine malware and prevent them from harming real systems are usually much slower than "real" execution environments. In general, the length of the dynamic analysis process 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. The time necessary to revert and restart the instrumented environment in a clean state must also be taken into account. With the continuous increase of malware submissions, the execution time is quickly becoming a bottleneck for automated analysis tools.

Throwing more hardware at the problem offers only temporary relief, considering the growing number of malware variants produced by malware authors. Moreover, it is a wasteful approach, as a large majority of released malware samples are simple variations of existing ones. Thus, instead of re-running the same programs thousands of times, it would be preferable to spend the available resources on previously unseen malicious programs. To this end, we require an approach to determine whether a new sample is similar to one that was analyzed before. Of course, to be useful, this approach must deliver results faster than it takes to dynamically analyze a sample.

In the past years, different approaches have been explored to statically address the problem of malware similarity. In [10], the authors introduce a distance-based approach that uses the edit distance, 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 (see, for example [18, 21, 25]). In [8] and [11], the authors introduce a graph-based approach, which compares graph representations extracted from the disassembled code. Some of these approaches are computationally expensive. More importantly, they all require that the malicious code is *unpacked* and disassembled first. Unfortunately, existing generic unpackers rely on the dynamic instrumentation of executables [12, 17, 23]. As a result, they need to run samples, and hence, suffer from similar performance limitations as the dynamic analysis tools themselves.

In this paper, we present an efficient, static technique that can identify samples that are similar to those that were analyzed previously, 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 and remove 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[1]. While this approach makes the system efficient, its reliance on ephemeral features is not robust, and *peHash* can be trivially confused. Our similarity measure, on the other hand, is based on properties directly derived from the code of the malware program, and hence, more difficult to tamper with.

To summarize, the contributions of this paper are the following:

- We introduce an efficient and robust similarity comparison technique for malware samples. This technique operates directly on the packed code section(s) of the malicious program.
- We present a packer detection method that can identify the type of packing algorithm used: compression, encryption, and multi-layer encryption. The detection output is used to automatically configure the sensitivity of the similarity measure.
- We discuss a prefiltering method to select candidate samples within a malware repository. This prefilter relies on efficient heuristics to quickly discard irrelevant samples.
- We have developed a system based on our novel techniques, and we have evaluated this system over a large malware repository containing 795 thousand samples. Our experiments demonstrate that our similarity measure is effective in filtering out malware variants obtained by re-packing or re-encrypting the same, original malware. Moreover, the results show that the system can reduce the number of samples that need to be considered by dynamic analysis tools by a factor of three to five.

[1]PE is the executable format used by Microsoft Windows.

2 Similarity and the packing issue

Techniques to compute the static code similarity between malware samples face the same problems as static malware detection; the packers and mutation engines (e.g., polymorphism, metamorphism) 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 samples currently found in the wild. Initially, packers were only used to reduce the size of the executables by using compression. However, because compression is deterministic, it was easily reversed and the number of generated variants was limited. Thus, to hinder reverse engineering and detection, malware authors soon added an encryption step. Encryption makes unpacking more difficult and increases the number of variants that can be generated by simply changing the encryption key. Very recently, protections based on virtualization have been introduced, where the original program is translated to virtual instructions that are then executed by an embedded virtual machine [22]. In this work, we do not address virtualization-based packing since it is still restricted to a small number of commercial packers that malware writers rarely use.

In general, compression and encryption severely hinder any similarity computation on executables because the contents of the code sections are modified, and, with this, their 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 (see Table 3, page 12).

A first, important observation is the limited number of underlying key 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. With respect to encryption, reversible arithmetic operations (such as *add/sub*, *rol/ror*, *xor* with 8-bits or 32-bits keys) are the most commonly used form. 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 standard OS crypto APIs and/or libraries (using such APIs could raise suspicion and makes reverse engineering and detection easier).

Table 1 presents the key algorithms used by packers, as well as their impact on the byte alignment, sequence, and distribution of the packed program. The table entry "alignment" indicates whether a byte-aligned data block remains aligned after packing. The entry "byte sequence" discusses the effects of packing on the order of bytes (or regions) in the original program. The table entry "byte distribution" characterizes how the distribution of bytes (or n-grams) in the original binary is altered by the packing process.

Compression algorithms. Dictionary-based packers keep certain (incompressible) parts of the original program intact, while they compact other parts by replacing an entire sub-sequence with a reference (relative offsets) to a previously-seen, uncompressed occurrence of this sub-sequence. If two executables are similar before packing, their incompressible parts will be mostly similar as well. As for the compressed sub-sequences, one can expect that most of the relative offsets into the uncompressed regions are also similar. Dictionary-based algorithms align both the uncompressed regions and references to previously-seen regions on byte boundaries.

Entropy encoders operate by replacing frequent bytes (or blocks) with shorter bit strings (symbols). The lengths of these bit strings are typically not a multiple of a byte, and hence, the byte alignment is destroyed. This also significantly alters the byte distribution. However, observe that when considering the encoded symbols, their distribution is identical to the byte (block) distribution of the data before entropy encoding. Similar considerations hold for range encoders.

Dictionary compression	Principle	Frequent byte sequences are replaced by relative references to previous occurrences.
	Alignment	Byte alignment preserved by incompressible data and aligned references.
	Sequences	Order of incompressible sequences is preserved. Relative references are interleaved in between, but their number is bound by the size of the reference window.
	Distribution	Distribution is flattened because frequent byte sequences are replaced by relative references that tend to introduce infrequent (random) byte values.
	Examples	*LZ77*, which is at the core of *LZO* (used by *PolyEnE*) and *NRV* (used by *UPX*).
Entropy encoding	Principle	Most-frequent bytes (or blocks) are encoded by bit strings (symbols) of smaller size.
	Alignment	Byte alignment destroyed by the shortened representations.
	Sequences	Byte blocks are replaced by shorter representations but their sequence is preserved.
	Distribution	Byte distribution is destroyed due to lost alignment, but distribution can be reconstructed over the encoded representations (symbols).
	Examples	*Huffman*, which is combined with *LZ77* for *deflate* (used by *gzip*).
Range encoding	Principle	Entire byte sequences are replaced by a single (large) integer range representation.
	Alignment	Byte alignment destroyed by the shortened representations.
	Sequences	Sequences are shortened, but order remains.
	Distribution	Bytes distribution is destroyed due to lost alignment, but distribution can be recomputed over the encoded representation).
	Examples	Range encoding is combined with *LZ77* for *LZMA* (used by *NsPack*).
Arithmetic encryption	Principle	Blocks of bytes are independently encrypted using reversible arithmetic operations.
	Alignment	Byte alignment 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.
	Examples	*xor*, *add/sub* or *rot/rol* are used for 32-bit encryption in *PolyEnE*.
Key or operation variation	Principle	A different key or operation is used for the encryption of each new block.
	Alignment	Byte alignment 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 (whose lengths are equal to the cycle length multiplied by the size of the original blocks).
	Examples	*Yoda's Crypter* uses a cycle of *xor*, *add/sub* or *rot/rol* encryption operations.
Multi-layer encryption	Principle	The entire input is encrypted multiple times.
	Alignment	Byte alignment preserved by the key and the byte blocks.
	Sequences	If layers are aligned with identical block sizes, equivalent to a single encryption, otherwise equivalent to key variation due to the overlap.
	Distribution	If layers are aligned with identical key sizes, equivalent to a single encryption, otherwise equivalent to key variation due to the overlap.
	Examples	*tElock* uses multiple layers of 8-bit *xor* encryption.

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

Encryption algorithms. The simple arithmetic operations used by crypters only achieve a simple substitution of the byte blocks in the original code. Also, most existing crypters do not implement any chaining or other form of key and operation variation, and, when they do, the variation is always cyclic. Arithmetic encryption results in a permutation of the distribution of the original n-grams. The alignment of bytes is typically not affected.

An important conclusion that can be drawn from the aforementioned observations is that packers preserve certain properties of the original code. Compressors tend to alter the byte alignment. However, when considering the symbols compressed by encoding, some sequences are incompressible by dictionary-based compression and references to compressed sequences are deterministically determined. Compressors thus preserve similarity because distributions coming from originally similar programs will be computed over similar, compressed data. 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.

Structural modifications. In addition to modifications to the code itself, the packing process affects the structure of the executable. The sections hosting the packed code and data are often renamed and possibly merged. The read, write, and execute permissions of these sections are modified to allow the modification of their image in memory. Moreover, an additional code loader is appended to the executable to recover the original code during runtime. The entry point is accordingly redirected to point to the loader, which transfers control back when the original code is unpacked. Finally, the packing process also alters the linking information by destroying the `Import Address Table(IAT)`, which lists the different APIs imported by the executable. The original `IAT` is then rebuilt at runtime by the loader.

3 Description of the similarity measure

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 malware analysis systems (such as *Anubis*) and discard samples that are too similar to ones already analyzed. For this use case, we require a similarity measure that is fast, while offering sufficient robustness to tolerate small changes in the malicious code.

To compute the similarity between two (malware) programs, we compute the distance between their *code signals*. A code signal is essentially a bigram distribution over the raw bytes of the code section, but extracted in a novel way to compensate for the modifications (noise) introduced by packers (as discussed in the following section).

An overview of the system is presented in Figure 1. We keep a database of previously-analyzed malware programs that stores, for each sample, its code signal. Whenever a new sample arrives, we extract its code signal. Then, we compute the distance to those stored within the database (see Section 3.1). If this distance is below a certain threshold, a very similar malware 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 the precision of the distance computations, two additional steps are introduced. First, we use a packer detector that automatically configures the sensitivity of the distance computation based on the type of packing that was used (see Section 3.2). Second, the distance computations are not performed for all samples in the database. Instead, a prefilter selects likely candidates (see Section 3.3) to reduce the number of necessary calculations.

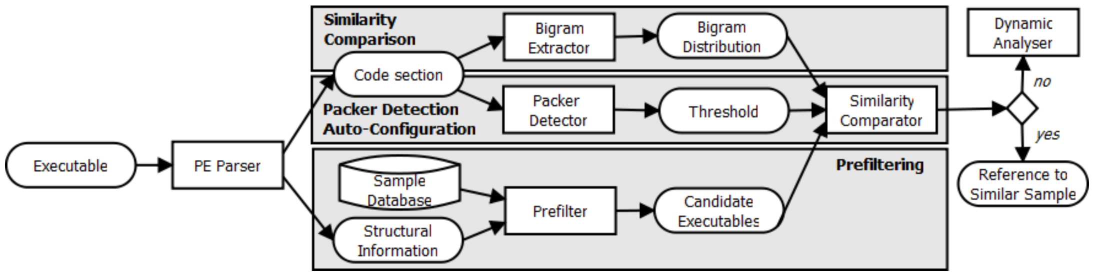


Figure 1: Architecture of our similarity filter.

3.1 Extracting and comparing code signals

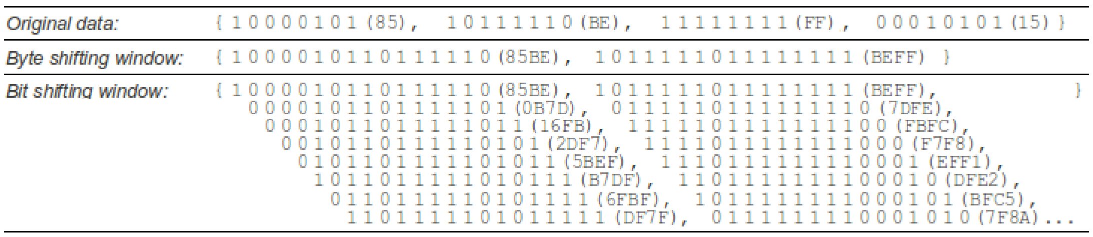
As mentioned previously, we employ code signals to characterize the executable section of a binary, and we use them as the basis for our distance computations. A code signal is a distribution of byte bigrams (pairs of subsequent bytes), extracted directly from a program's code segment. However, it is not a straightforward bigram distribution. The first difference is that, instead of advancing a two byte sliding window in increments of one byte, we use a bit-shifting approach in which the sliding window is advanced in increments of only a single bit. Second, the extracted distribution is not stored as-is, but sorted in descending order of frequencies. We discuss these steps and their rationale in more detail below.

One reason for operating directly on the raw bytes of the malware code is speed. No disassembly nor any other interpretation of the bytes are 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. The representation as a distribution of bigrams instead of a sequence representation offers two advantages. First, the distribution remains a vector of fixed size, independent of the size of the original code. Second, the distribution is less sensitive to code location (e.g., the distribution does not change when functions or instructions are reordered).

Of course, it would be possible to use larger n-grams instead of bigrams. However, this implies a large memory footprint without improving our results. On the other hand, smaller unigrams (single bytes) are not distinctive enough to clearly differentiate programs.

Extracting code signals. As mentioned previously, code signals are designed to compensate for "noise" (the transformations) caused by packing. As discussed in Section 2, when two similar programs are compressed or encrypted by current packers, the resulting binaries share certain similarities. We exploit these similarities that "shine through" the packing process to perform our comparison. In particular, we introduce two techniques to address the previously-identified problems of alignment destruction and distribution permutation, respectively.

1) Bit-shifting window: To recover from the destruction of the byte alignment, a bit-shifting window is used to extract the bigrams, instead of the traditional byte-shifting window. The bit-shifting process is shown in Figure 2. The importance of the bit-shift lies in its capacity to resynchronize with the correct alignment in a compressed stream of data. To illustrate the importance of this alignment, two similar sequences of bytes are shown in Figure 3, with only a few bits of difference. These sequences are compressed using *LZMA*. The figure shows that a byte-shifting window loses the alignment for the remaining of the extraction step whereas the bit-shifting window recovers as soon as the modified bits are out of its scope.

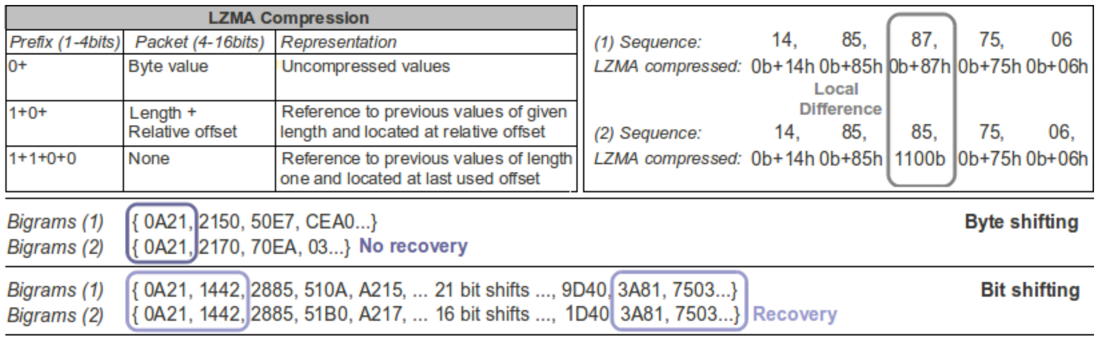


```
Original data:        { 1 0 0 0 0 1 0 1 (85),  1 0 1 1 1 1 1 0 (BE),  1 1 1 1 1 1 1 1 (FF),  0 0 0 1 0 1 0 1 (15) }
Byte shifting window: { 1 0 0 0 0 1 0 1 1 0 1 1 1 1 1 0 (85BE),  1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 (BEFF) }
Bit shifting window:  { 1 0 0 0 0 1 0 1 1 0 1 1 1 1 1 0 (85BE),  1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 (BEFF),              }
                          0 0 0 0 1 0 1 1 0 1 1 1 1 1 0 1 (0B7D),  0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 (7DFE),
                            0 0 0 1 0 1 1 0 1 1 1 1 1 0 1 1 (16FB),  1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 0 (FBFC),
                              0 0 1 0 1 1 0 1 1 1 1 1 0 1 0 1 (2DF7),  1 1 1 1 0 1 1 1 1 1 1 1 1 0 0 0 (F7F8),
                                0 1 0 1 1 0 1 1 1 1 1 0 1 0 1 1 (5BEF),  1 1 1 0 1 1 1 1 1 1 1 1 0 0 0 1 (EFF1),
                                  1 0 1 1 0 1 1 1 1 1 0 1 0 1 1 1 (B7DF),  1 1 0 1 1 1 1 1 1 1 1 0 0 0 1 0 (DFE2),
                                    0 1 1 0 1 1 1 1 1 0 1 0 1 1 1 1 (6FBF),  1 0 1 1 1 1 1 1 1 1 0 0 0 1 0 1 (BFC5),
                                      1 1 0 1 1 1 1 1 0 1 0 1 1 1 1 1 (DF7F),  0 1 1 1 1 1 1 1 1 0 0 0 1 0 1 0 (7F8A)...
```

Figure 2: Bigram extraction using bit shifting window.

2) Sorted distribution: Once all bigrams are extracted from a malware's code section (using the bit-shifting approach outlined above), we can compute the bigram frequencies. Their distribution is then normalized to obtain a probability distribution.

To address the possible, additional encryption of the code by simple arithmetic operations, the distribution is sorted in decreasing order. 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.



LZMA Compression		
Prefix (1-4bits)	Packet (4-16bits)	Representation
0+	Byte value	Uncompressed values
1+0+	Length + Relative offset	Reference to previous values of given length and located at relative offset
1+1+0+0	None	Reference to previous values of length one and located at last used offset

(1) Sequence: 14, 85, 87, 75, 06
LZMA compressed: 0b+14h 0b+85h 0b+87h 0b+75h 0b+06h
Local Difference
(2) Sequence: 14, 85, 85, 75, 06,
LZMA compressed: 0b+14h 0b+85h 1100b 0b+75h 0b+06h
Bigrams (1) { 0A21, 2150, 50E7, CEA0...}
Bigrams (2) { 0A21, 2170, 70EA, 03...} No recovery
Byte shifting
Bigrams (1) { 0A21, 1442, 2885, 510A, A215, ... 21 bit shifts ..., 9D40, 3A81, 7503...}
Bigrams (2) { 0A21, 1442, 2885, 51B0, A217, ... 16 bit shifts ..., 1D40 3A81, 7503...} Recovery
Bit shifting

Figure 3: Bit shifting window to recover from LZMA compression re-alignment.

In general, perfect recovery is possible in cases where the size of the encryption blocks is equal to or smaller than the size of the n-grams used to construct the distribution (two bytes, in our case). If blocks are of a larger size than n, the quality of the recovery process depends on the extent to which the single encryption operation on a large block can be approximated as separate (independent) encryption operations on sub-blocks of length n. This depends on the diffusion between the bits of different sub-blocks when the encryption operation is applied, and thus, ultimately, on the arithmetic operation and the key that are used.

Let us consider a bigram distribution of the code; this bigram distribution can be computed from the 4-gram distribution as follows: for any bigram x, $p(x) = \sum_y p(xy) + \sum_z p(zx)$ where y and z represent all the possible adjacent bigrams. Let us now take the example of a 32-bit block encryption function E, which is based on an arithmetic operation *op* (such as $\oplus$ or *add*). E is applied to the 4-byte values xy and vw, where x, y, v, w are all 2-byte values. This yields $x'y' = E(xy)$ and $v'w' = E(vw)$. The function E fails to achieve proper diffusion if $x = v$ implies that $x' = v'$ (and, similarly, $y = w$ implies that $y' = w'$), independently of the adjacent bytes. Figure 4 discusses different conditions under which certain arithmetic operations do not achieve diffusion. As one can see, *xor* never achieves diffusion, and with *add*, only a single carry bit influences the two high bytes. In such cases, the sum $\sum_y p(xy)$ from the original distribution is equal to $\sum_{y'} p(x'y')$ in the encrypted distribution. Consequently, we obtain the following approximation: $p(x) \approx p(x')$. Because of this bias, the sorting operation can successfully recover from the permutation introduced by the packer (crypter), even when encryption blocks are larger than bigrams.

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 $X' = X_1'X_2'$.

E **is xor operation:** Relation between encrypted values and inputs:

$$X_1' = X_1 \oplus K_1$$
$$X_2' = X_2 \oplus K_2$$

No diffusion is achieved between upper bytes of X_1 and lower bytes of X_2.

E **is addition:** Relation between encrypted values and inputs:

$$X_1' = X_1 + K_1 + carry$$
$$carry, X_2' = X_2 + K_2$$

No diffusion is achieved between upper bytes of X_1 and lower bytes of X_2 when $carry = 0$.
When $carry = 1$, only the rightmost bits of X_1 are impacted by the result of the encryption of X_2.

E **is rotation:** Rotation achieves diffusion because overflowing bytes are transferred from the side they overflow to the opposite side. Still, two particular key values 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 $X_2' = X_1$

Figure 4: Diffusion between upper and lower bytes for different, arithmetic encryption operations.

Example. The effectiveness of the two aforementioned transformations to produce meaningful code signals is shown on a concrete example in Figure 5. The figure shows the byte distributions of two similar programs A and B on the left. First, these two programs were packed with LZMA. The top right quadrant of the figure shows the byte distributions when extracted with a byte-shifting and a bit-shifting window, respectively. The similarities between the distributions for the bit-shifting approach can be clearly seen, while no similarities are present for the byte-shifting approach.

In the bottom right quadrant, one can see the effect of encrypting A and B with different encryption operations and keys. Looking at the unsorted distributions, no similarities are evident. Once the distributions are sorted, the situation changes. Note that small difference in the distributions for the bit-shifting approach and the sorted permutations are due to the fact that A and B are not identical. In fact, to produce B, 20% of all bytes in A were overwritten with random values. Nevertheless, the code signals show clear similarities.

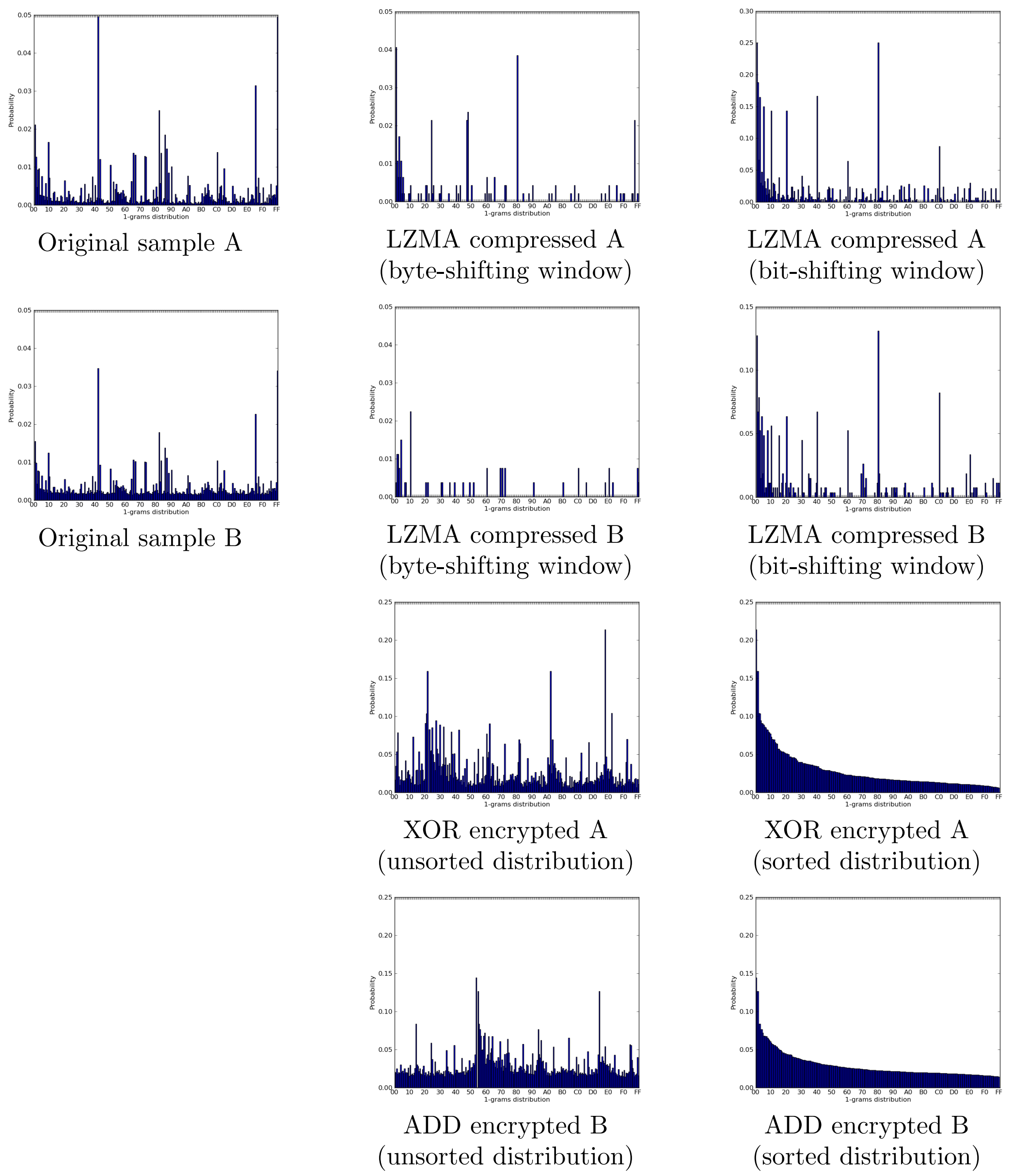


Figure 5: Reversing alignment destruction and distribution permutation.

Comparing code signals. The comparison between code signals (the bigram distributions) is performed using Pearson's χ^2 test:

$$\chi^2 = \sum_{i=0}^{2^{16}} \frac{(o_i - r_i)^2}{r_i^2},$$

where o_i is the distribution extracted from the submitted sample, and r_i are the reference distributions from the candidate samples taken from the database.

If the test value remains below a given threshold τ, the samples are considered similar. 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 DB). 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 for the type of program that the χ^2 test is applied to. This is because slightly different thresholds should be used depending on whether a sample is packed or not (and, if it is packed, whether it is compressed or encrypted).

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 bit strings. 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). To compensate for this, the similarity threshold should be reduced accordingly when checking packed executables.

To detect packed executables, we leverage the insight that a "flattened" distribution makes packed code more similar to random data. Thus, the statistical properties used to assess random generators can be used to detect packed executables, and even identify different types of protection: compression, encryption, and multi-layer encryption. The similarity threshold is then automatically reduced according to the type of protection detected.

Packer detection and classification. To detect packers and to identify the type of protection, we make use of four statistical tests T_1, T_2, T_3 and T_4, described below in Figure 6.

T_1**:** Uncertainty test: Compute the code entropy.
T_2**:** Uniformity test: Compute χ^2 between the code distribution and an equiprobable distribution.
T_3**:** Run test: Compute the longest sequence of identical bytes in the code.
T_4**:** 1^{st}-order dependency test: Compute the autocorrelation coefficient of the code.

Figure 6: Statistical tests for packer detection. These tests are performed over the raw bytes in the actual code section or the packed code section (depending on whether 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 above a 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 [24]. 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, encrypted code is more uniform that compressed code, 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, however, the non-overlapping boundaries of different layers introduce additional discontinuities. These discontinuities are measured by T_4. Table 2 explains how our four tests are combined to identify the generic type of packing. The advantage of our approach is that it relies only on statistical properties, without requiring any packer-specific knowledge.

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

Table 2: Packer detection and classification.

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 the packed code. Since packers modify the sections of executables, the risk is to perform the test over the section that contains the loader. Figure 7 shows our heuristic to find the section that contains the packed data. We tested this heuristic on 20 packers, and found that it always located the correct section. Once identified, this section is also used to extract the code signal.

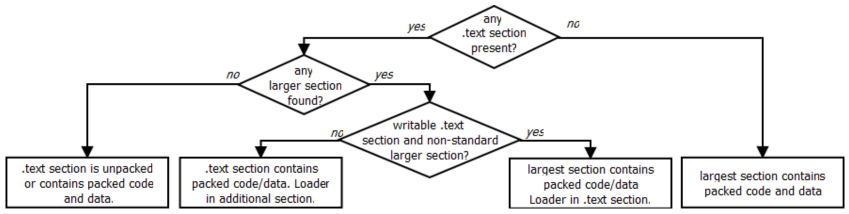

any .text section present?
yes
no
any larger section found?
no
yes
writable .text section and non-standard larger section?
no
yes
.text section is unpacked or contains packed code and data.
.text section contains packed code/data. Loader in additional section.
largest section contains packed code/data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

Figure 7: Heuristic used to find the packed section (if present) in a binary.

3.3 Fast prefilter

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

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. However, taking into account these factors, we can define minimum and maximum bounds to split executables into bins of different sizes. The prefilter first selects candidate samples from the bin that contains samples with sizes similar to the new sample.

PE-characteristic-based filtering. Further criteria of similarity between executables are provided by 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. PE features can be obtained by parsing the PE file. However, only a subset of these features are useful for prefiltering. In particular, we only consider PE features that provide sufficient differentiation between executables while being robust to packing (that is, PE header features that are not modified by packers). The 16 features currently considered by our heuristic are presented in Section 4, Table 7.

The prefilter computes the Hamming distance between the PE features of a new (incoming) sample and the features of all samples from the database that were selected by the first heuristic. When the distance is larger than a threshold, the corresponding database sample is no longer considered. All remaining samples become candidates for the subsequent similarity measure computation.

4 Evaluation

The filter presented in the previous section was implemented and is used to process samples submitted to an automated, dynamic malware analysis system. More precisely, the filter checks all incoming malware samples and sends to the analysis tool only those samples for which no similar instances can be found in the database of previously examined malware.

The evaluation was carried out in two steps. For the first step, presented in Section 4.1, we used our filter on known samples for which ground truth was available. The goal of this first step was to establish the similarity thresholds and verify the accuracy of our similarity measure. For the second step, presented in Section 4.2, we applied the filter to a large collection of malware samples that were provided to us by the authors of *Anubis* [1]. The goal of the 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 different packers. We also took advantage of this second step to study the scalability and the robustness of our approach.

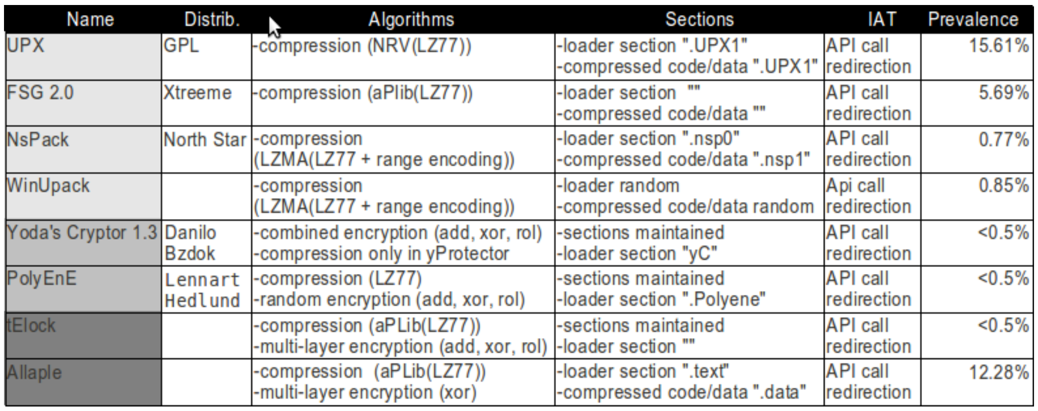
4.1 Experiments on known samples

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 *Open Office*, 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 and corresponding, compiled executables were made available to us. These bots belong to two different malware families; 23 *SdBot* and 42 *rBot* binaries. 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 effectiveness of our similarity measure.

4.1.1 Packer detection

To assess our packer detection technique, we selected seven packers, based on their popularity with malware writers. These packers were *FSG*, *UPX*, *NsPack*, *WinUPack*, *Yoda's Crypter*, *PolyEnE* and *tElock*. We also added instances of the *Allaple* worm as a representative example for polymorphic malware. Technically, *Allaple* is not a packer, but a stand-alone malware program. However, its polymorphic engine uses techniques similar to packing.



Name	Distrib.	Algorithms	Sections	IAT	Prevalence
UPX	GPL	-compression (NRV(LZ77))	-loader section ".UPX1" -compressed code/data ".UPX1"	API call redirection	15.61%
FSG 2.0	Xtreeme	-compression (aPlib(LZ77))	-loader section "" -compressed code/data ""	API call redirection	5.69%
NsPack	North Star	-compression (LZMA(LZ77 + range encoding))	-loader section ".nsp0" -compressed code/data ".nsp1"	API call redirection	0.77%
WinUpack		-compression (LZMA(LZ77 + range encoding))	-loader random -compressed code/data random	Api call redirection	0.85%
Yoda's Cryptor 1.3	Danilo Bzdok	-combined encryption (add, xor, rol) -compression only in yProtector	-sections maintained -loader section "yC"	API call redirection	<0.5%
PolyEnE	Lennart Hedlund	-compression (LZ77) -random encryption (add, xor, rol)	-sections maintained -loader section ".Polyene"	API call redirection	<0.5%
tElock		-compression (aPLib(LZ77)) -multi-layer encryption (add, xor, rol)	-sections maintained -loader section ""	API call redirection	<0.5%
Allaple		-compression (aPLib(LZ77)) -multi-layer encryption (xor)	-loader section ".text" -compressed code/data ".data"	API call redirection	12.28%

Table 3: Specifications of the tested packers.

An overview of the packers (and *Allaple's* polymorphic engine) is provided in Table 3. The tools implement a wide variety of compression and encryption algorithms, and the table shows that there are four compressors, two crypters, and two multi-layer crypters. According to [7], the eight packers cover 35% of the samples from the *Anubis* data set (see column *Prevalence*). This corresponds to 72% of all packed samples. Moreover, because our detection technique is not signature-based but based on information theoretic metrics, the obtained results also extend to many other packers that use similar algorithms.

For the experiment, we used our packer detection techniques introduced in Section 3.2. To produce a test set, we packed each of the 384 executables from S_1 with all seven packers, respectively, and then added 120 samples of *Allaple*. The unpacked versions of the 384 programs were also included. The necessary thresholds ($t_1 = 4.73$, $t_2 = 0.0012$, $t_3 = 2$, $t_{4a} = 0.005$, $t_{4b} = 0.002$) for the detector were selected empirically based on a small subset of samples taken from the test set.

The detection results are presented as a confusion matrix in Table 4. One can see that the detector is able to distinguish very well between unpacked and packed executables. More precisely, the detection rate for unpacked samples is 99.74%, while it is over 90% 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. In particular, compressed samples were very well identified. On the other hand, a certain fraction of crypters was misclassified as compressors. The lowest detection 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

Name	Unpacked	Packed	Compressed	Encrypted	Multi-layer Encrypted
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%
Yoda's Crypter	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-layer crypters	02.42%	97.58%	00.28%	71.58%	25.72%

Table 4: Detection and classification of packers by statistical testing.

layers of *xor* encryption are basically equivalent to a single layer (especially when identical block sizes and alignments are used for every layer). Thus, in many cases, our detector actually produces a correct result by classifying a multi-layer-encrypted sample as the output of a regular crypter.

When putting our detection results into the context of related work, we note that no previous technique provides fine-grained distinctions between different types of packing without making use of packer-specific signatures. For example, in [16], the authors present a system that computes the entropy of fixed-size code blocks and then uses the average and maximal entropy values to detect packed executables. Applying their technique to our test set yields a detection rate of around 90%, similar to our results. In [20], a packing detection technique is introduced that does not rely on statistical properties of the code but on structural properties of the executable sections. Compared to both previous systems, our technique is more precise because we do not only distinguish between packed and regular code, but, thanks to our additional tests, we also provide a more precise classification and recognize compressors, crypters, and multi-layer crypters.

In [9], the authors use entropy to establish profiles that allow the identification of specific packer tools. This is more precise than our detector. However, the system needs to be trained for each individual packer that should be recognized, something that our system does not require (and that is not desirable, given the large number of different packers in the wild).

Discussion. Our filter leverages the results of the packer detector for the automated tuning of the similarity threshold. For this, the most important distinction is between between packed and unpacked samples, which works well; the additional classification results are used for further refinement. It is important to observe that a misclassification only leads to the use of a suboptimal threshold, but it does not prevent the system from computing correct similarity results.

4.1.2 Filter robustness

The goal of the next experiment was to understand to which extent our filter can recognize similar executables, even when they are packed. In other words, what amount of changes between two malware programs can be tolerated before they are deemed different, especially when the variants are packed. To assess the robustness of our similarity measure to modifications of a program, we selected ten random binaries from S_1. We then generated 10,000 variants for each program. To this end, we overwrote an increasing number of randomly-selected bytes in (the code sections of) the binaries, increasing the fraction of modified bytes from 0% to 30%. The original executables and their modified versions were then packed with the seven packers presented previously. *Allaple* was left aside because its polymorphic engine is directly embedded in the binary.

Packer	Threshold	Percentage of modifications									
		0.00%	0.05%	0.1%	0.5%	1.0%	2.0%	5.0%	10%	20%	30%
None	0.0020	100%	100%	100%	100%	100%	0%	0%	0%	0%	0%
FSG	0.0018	100%	100%	100%	100%	100%	100%	100%	0%	0%	0%
UPX	0.0018	100%	100%	98%	58%	24%	17%	0%	0%	0%	0%
NsPack	0.0018	100%	100%	100%	100%	100%	100%	0%	0%	0%	0%
WinUPack	0.0018	100%	100%	100%	100%	100%	100%	0%	0%	0%	0%
YodaCrypter	0.0015	26%	21%	33%	27%	25%	23%	9%	0%	0%	0%
PolyEne	0.0015	90%	50%	7%	41%	1%	39%	22%	30%	1%	1%
tElock	0.0013	100%	100%	100%	99%	100%	100%	6%	0%	0%	0%

Table 5: Similarity resilience to packers.

Table 5 presents the results for the experiment. More precisely, the table shows, for different levels of changes to the code, the fraction of variants that our filter considered similar to the original program. We can see that our measure accurately identifies most variants as similar to the original program when the amount of overwrites was below 2%. For modifications that exceed this level, the similarity quickly drops. This is expected (and desirable), and it indicates that the filter can correctly identify a program as a variant when it shares most bytes with the original. For programs that are too different, the similarity reflects this and drops rapidly.

As expected, the filter performs worst in the case of crypters. Even for variants that start out as identical to the original (before packing), our system does not detect all of them as similar. The reason is that the size of the encryption blocks used by *Yoda's Crypter* and *PolyEne* are 32 bits, which is larger than the 16-bit bigram size that we use. Moreover, *Yoda's Crypter* uses a cycle of different encryption operations and keys per block, further worsening detection. Nevertheless, a reasonable number of variants are still identified correctly. The excellent results for the multi-layer crypter *tElock* are interesting (and possibly surprising). These results can be explained by the fact that *tElock* combines several layers of 8-bit block encryptions using *xor*, which, for our analysis, is equivalent to a single encryption layer with a one byte block size.

4.1.3 Tuning the filter granularity

The goal of the next experiment is to demonstrate that our filter offers a satisfactory precision for its intended application, which is the recognition of similar malware samples, even when they are packed. For this, we turned our attention to S_2, the set of 65 classified bot samples. More precisely, to build our test set, these 65 bots, together with all benign 384 programs from S_1, were packed and submitted to the filter.

The results of the experiment are shown in Table 6. To measure the precision of our filter, we use the following metrics: (i) the rate of true hits, TH, which correspond to those 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 cases are the most critical, because they may result in a loss of interesting information); and (iii) the rate of misses (or false negatives), M, which correspond to cases where samples are submitted for further analysis even though they should have been discarded (these errors are much less severe, because a miss only results in an unnecessary analysis run).

We present the results for two different sets of thresholds. The first set of thresholds, which was also used for the experiments in previous sections, 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. In this case, we observe 96% of true hits on average, with only 3.7% misses and, more importantly, only 0.3% of false hits.

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

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

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

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*

	Family granularity thresholds							Version granularity thresholds			
Packer	Thrsh.	TH(f)	FH(f)	M(f)	TH(v)	FH(v)	M(v)	Thresh.	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%
YodaCrypter	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	-							-			

Table 6: Precision of the similarity measure for various packers.

However, depending on the level of granularity that is desired, it might be preferable to analyze different versions of the same malware family. In this case, the family granularity thresholds are likely too loose. This can be seen by looking at the false hit rates for malware versions, denoted as $FH(v)$, which reach almost 6% when using family granularity thresholds. To be able 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, the false hits drop close to 0%. However, we also have to accept that the rate of misses increases, which leads to more unnecessary analyses. This shows that the filter offers a trade-off between accuracy and efficiency that can be adjusted via the similarity thresholds.

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 of the worm, namely, *Allaple.b* and *Allaple.e*. The results are also given in Table 6. The worm variants are accurately detected in more than 92% of the cases, with a good distinction between the two versions.

4.1.4 Configuring the prefilter

To increase the speed of our system, we apply a fast prefilter to select candidate samples for further analysis. In this section, we discuss details about the configuration of this prefilter.

The first heuristic divides samples into bins of different (program code) sizes. These size ranges must be configured properly so that the variants of a given program fall within the same bin, while the bins are tight enough to be effective. To measure the impact of current packers on the sizes of binaries, we used the set of 100K patched (modified) and packed samples from Section 4.1.2. Our analysis results indicated that similarity is preserved up to 2% of modifications. For this amount of modifications, the maximum size variation that we observed, over all packers, was 4.4%. Based on this observation, the lower and upper bounds for the first heuristic were set to 95.6% and 104.4% of the original size. Outside these boundaries, samples are considered too different.

The second heuristic is used to determine those candidate samples that share sufficient structural similarities with the new input sample. The structural features are extracted from the PE headers of the binaries. Of course, only a subset of the possible PE features can be used to distinguish programs. In particular, we require features that distinguish well between different programs, are preserved by the packing process, and are identical (or similar) for similar samples.

To find useful, structural features, we again examined the patched and packed samples discussed in Section 4.1.2. This time, we examined their PE headers, computing the entropy, H, and the absolute number of different values, *card*, for each feature. While large values for H and *card* are a good indicator that a feature might be interesting for our purpose, we had to discard those that were different for each packed sample (e.g., timestamps or checksums). Moreover, we had to discard features that are destroyed by the packing process, such as information from the section table or the import table. Table 7 provides the list of the remaining 16 features that satisfy our requirements. To select a sample from the database as a candidate for comparison with a new malware binary, we require that all features of the two programs are identical (the threshold for the Hamming distance is 0). As our experiments demonstrate, even such a restrictive check yields good results. We discuss the robustness of the prefiltering step in more detail in Section 4.2.2.

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

Table 7: PE Header characteristics selected for comparison.

4.2 Large scale experiments

The experiments with known malware (bot) samples allowed us to analyze the accuracy of our filter, to tune detection thresholds, and to 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. We obtained these samples from the authors of [6]. Moreover, we were given behavioral information (execution traces) for each sample and a reference clustering. This clustering partitioned the malware programs into 91,522 different groups, where all samples in a group exhibit similar runtime activity.

4.2.1 Precision and scalability of the filter

We applied our filter to the entire data set of almost 795 thousand malware samples. To evaluate the precision of the filter, we use the metrics previously introduced in Section 4.1.3, namely, 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 that were provided to us [6]. In a first step, we considered two samples as similar when they produced similar behaviors (execution traces), and hence, ended up in the same behavioral cluster. 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 command and control servers). As a result, similar samples might end up in different behavioral clusters. Thus, to improve the reference clustering, we also considered some simple, 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 used this information in a second step, where 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 8 shows the precision of our filter for three sets of thresholds. The first two correspond to the thresholds for family and version granularity, respectively, while the third is an extra set with more conservative thresholds. These three sets represent different points in the trade-off between reducing unnecessary analysis runs (true hits) and the risk of discarding potentially interesting samples (false hits).

For the first thresholds, it can be seen that 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 accordingly, the reduction factor that can be achieved.

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

Table 8: Accuracy of the comparison according to the selected thresholds. Thresholds legend: U – Unpacked, C – Compressed, E – Encrypted, MLE – Multi-Layer Encrypted

We then analyzed the false hits produced by our filter in more details. We found that incorrect similarities can be explained either by the failure of the heuristic to find the section containing the packed code, or, in most cases, by the misclassification of samples that, although they belong to different families, are part of the same class of malware. This was mainly true 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]).

With respect to misses, we found that many cases were caused by similar samples that exhibit similar dynamic behavior but that were protected by different packers. Not surprisingly, for a given executable, our 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 when the number of submitted samples increases. According to Figure 8, the database size increases sub-linearly with the number of submissions. Figure 9 shows a linear increase of the computation time with the number of entries. The average computation time for similar samples is lower because, as soon as a similar sample is found in the database, the computation stops. Even in the worst case, for a unique sample, the filter takes no more than 300 ms. This would be 1,200 times faster than the 6 minutes required to execute a sample within *Anubis*. Moreover, the slowdown that we observe with the increase of the database size indicates that the system will scale at least to tens of millions of samples.

The prefilter plays an important role in helping to achieve excellent performance. In Figure 10, it can be seen that the two heuristics reduce the candidate set to less than 1% of the database

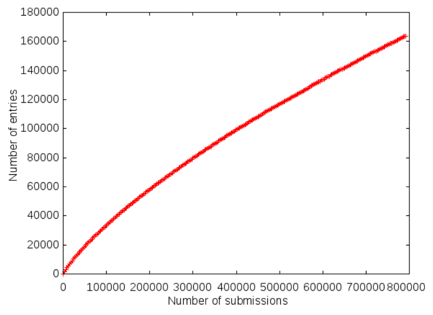


Figure 8: Database growth.

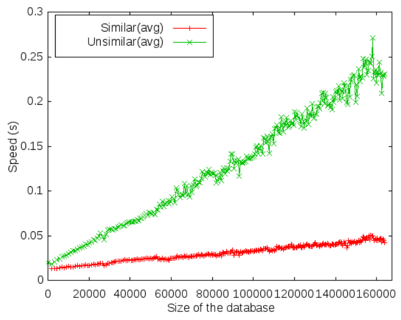


Figure 9: Time/Submission.

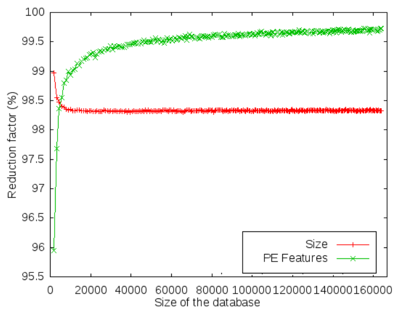


Figure 10: prefilter reduction.

samples. Moreover, the figure shows that the prefilter maintains its effectiveness independent of the size of the database.

4.2.2 Comparison with existing techniques and robustness

To compare our filter with previous work on identifying malware similarity, we selected a subset of 18,645 samples from our real-world data set. These samples were chosen because we were given the corresponding, unpacked binaries, produced by a dynamic, generic unpacker that is part of *Anubis*.

For the comparison, we first reimplemented *peHash* [27]. This tool operates mostly on structural characteristics of malware samples, and hence, does not require to unpack or disassemble the code beforehand. We also wanted to understand how much the precision of our filter suffers because it has to operate on packed code (bytes) instead of disassembled instructions. For this, we implemented a second version of our filter, where the bigram distribution (code signal) of a binary is not computed over the raw (and possibly packed) bytes, but over bigrams of disassembled instructions. This technique is similar to the mechanism used by *Vilo* [26]. Finally, to compare with an alternative approach to detect malware similarity, we used an existing tool that operates on the programs' control flow graphs [15]. Unpacked samples are needed as input for the two latter systems.

Method	TH	FH	M	Time	Prerequisites
Distance-based (*Filter*)	80.8%	00.7%	18.5%	6 min	None
Hash-based (*peHash*)	81.1%	00.6%	18.3%	9 min	None
Distance-based	84.3%	00.5%	15.2%	239 min*	Code unpacked and disassembled
Graph-based	83.4%	00.4%	16.2%	847 min*	Code unpacked and disassembled

Table 9: Comparison to other similarity measures. (* Unpacking time not included)

Table 9 compares the results obtained for the four tested tools, both in terms of precision and runtime. A first observation is that *peHash* performs quite similar to our approach. However, the approach followed by *peHash* comes at the significant expense of robustness, as we will discuss below. A second observation is that the two systems that 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 to unpack and disassemble the malicious code. Moreover, the runtime of these tools is slower by an order of magnitude. In addition, and more importantly, previously unpacking the samples with a generic, dynamic unpacker has a computational cost that is similar to that of the dynamic analysis that our filter is designed to avoid.

Robustness. In terms of robustness, it is useful to compare our filter to *peHash*. *peHash* does not require to unpack or disassemble the code; in fact, the system basically ignores the code altogether (with the exception of computing the Kolmogorov complexity for sections to estimate their degree of compression/encryption). Instead, the system mostly looks at structural information extracted from the PE header (which is somewhat similar to our prefilter).

The problem is that it is easy for a packer to change the structural features that *peHash* is relying upon. To demonstrate this, we developed a small obfuscation tool that can apply the following, very simple transformations to a binary: append new data sections of random size and position at the end of the memory image; increase the size of data sections on disk (without modifying the virtual memory layout); add access bits (shared, read, write, or execute) to sections; and inject random data in additional, small data sections. Using this tool, we generated 20K variants from ten original programs. These variants, submitted to *peHash*, generated 8,380 different hashes. Our technique, on the other hand, identified only 163 different samples. This shows that *peHash* can be easily fooled.

Of course, we also need to examine the reasons why our filter considered some samples as different that should have been recognized as similar. The first reason was that the sizes of the binaries were changed by the obfuscator. This caused the prefilter to put samples into different bins. The second reason was that our simple heuristic to identify the packed code section (see Figure 7) made mistakes. Both problems can be addressed: To handle the first one, we can increase the size of bins. This will result in a small performance loss, which, given the overall speed of our filter, is acceptable. The second problem would require a more sophisticated heuristic to distinguish packed code from random data. For this, we could leverage the code signals, which we already extract.

More generally, we also need to consider attacks that malware authors could launch to subvert our filter. One venue for the attacker is to target the prefilter. This can be achieved by modifying artificially the size of the executable and/or the selected PE features. However, the configuration of the prefilter can be easily loosened, either by increasing the range of the size-bins, by increasing the Hamming distance threshold, or by simply removing some features from the computation. The reason is that our similarity measure ultimately depends on the actual bytes in the code section. While loosening the prefilter criteria would lower its effectiveness, our system is currently at least three orders of magnitude faster than a dynamic analysis run. Thus, even a significantly slower analysis would still provide substantial benefits.

To tamper with the core of the technique, an attacker needs to modify the code section. One simple approach would be to append random data to it. To counter this, we could modify our filter to compute the similarity not for the entire code section, but for fixed-sized blocks. Samples are similar when a certain fraction or number of code blocks "match." More complex code modifications, such as interweaving random data with code, requires correct disassembly of the program. This is hard for packers that aim to use arbitrary binaries as input. The reason is that robust disassembly of x86 binaries is theoretically undecidable and difficult in practice [14].

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 [8, 11, 13, 26], our technique is thus able to directly operate on packed binaries, avoiding the costly unpacking process. This allows our system to filter submissions to malware repositories or automated dynamic analysis tools. Large scale experiments with almost 795 thousand 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: Analyzing unknown binaries. `http://anubis.iseclab.org`, 2010.

[2] CWSandbox :: Behavior-based malware analysis. `http://www.mwanalysis.org`, 2010.

[3] Norman Sandbox. `http://www.norman.com/technology/norman_sandbox/`, 2010.

[4] ThreatExpert. `http://www.threatexpert.com`, 2010.

[5] VirusTotal: free online virus and malware scan. `http://www.virustotal.com`, 2010.

[6] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proc. Symp. Network and Distributed System Security (NDSS)*, 2009.

[7] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. A view on current malware behaviors. In *USENIX Workshop Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[8] E. Carrera and G. Erdelyi. Digital genome mapping. In *Virus Bulletin*, 2004.

[9] T. Ebringer, L. Sun, and S. Boztas. A fast randomness test that preserves local detail. In *Virus Bulletin*, 2008.

[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] N. Krishnamoorthy, S. Debray, and K. Fligg. Static detection of disassembly errors. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 259–268, 2009.

[15] 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.

[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. IEEE Computer Society, 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] PandaLabs. Annual report. Technical report, 2009.

[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] R. Rolles. Unpacking virtualization obfuscators. In *USENIX Workshop Offensive Technologies (WOOT)*, 2009.

[23] 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.

[24] 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.

[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.