Deobfuscating Embedded Malware using Probable-Plaintext Attacks

Christian Wressnegger[1,2], Frank Boldewin[3], and Konrad Rieck[2]

[1] idalab GmbH, Germany
[2] University of Göttingen, Germany
[3] www.reconstructer.org

Abstract. Malware embedded in documents is regularly used as part of targeted attacks. To hinder a detection by anti-virus scanners, the embedded code is usually obfuscated, often with simple Vigenère ciphers based on XOR, ADD and additional ROL instructions. While for short keys these ciphers can be easily cracked, breaking obfuscations with longer keys requires manually reverse engineering the code or dynamically analyzing the documents in a sandbox. In this paper, we present Kandi, a method capable of efficiently decrypting embedded malware obfuscated using Vigenère ciphers. To this end, our method performs a probable-plaintext attack from classic cryptography using strings likely contained in malware binaries, such as header signatures, library names and code fragments. We demonstrate the efficacy of this approach in different experiments. In a controlled setting, Kandi breaks obfuscations using XOR, ADD and ROL instructions with keys up to 13 bytes in less than a second per file. On a collection of real-world malware in Word, Powerpoint and RTF files, Kandi is able to expose obfuscated malware from every fourth document without involved parsing.

Keywords: embedded malware, obfuscation, cryptanalysis

1 Introduction

Documents containing malware have become a popular instrument for targeted attacks. To infiltrate a target system, malicious code is embedded in a benign document and transfered to the victim, where it can—once opened—unnoticeably infiltrate the system. Two factors render this strategy attractive for attackers: First, it is relatively easy to lure even security-aware users into opening an untrusted document. Second, the complexity of popular document formats, such as Word and PDF, constantly gives rise to zero-day vulnerabilities in the respective applications, which provide the basis for unnoticed execution of malicious code. Consequently, embedded malware has been used as part of several targeted attack campaigns, such as Taidoor [28], Duqu [1] and MiniDuke [6].

To hinder a detection by common anti-virus scanners, malicious code embedded in document files is usually obfuscated, often in multiple layers with increasing complexity. Although there exist a wide range of possible obfuscation

strategies, many attackers resort to simple cryptographic ciphers when implementing the first obfuscation layer in native code. Often these ciphers are variants of the so-called *Vigenère cipher* using XOR and ADD/SUB instructions for substitution and ROL/ROR for transposition. The resulting code can fit into less than 100 bytes and, in contrast to strong ciphers, exposes almost no detectable patterns in the documents [see 4]. As an example, Figure 1 shows a simple deobfuscation loop using XOR that fits into 28 bytes.

Due to the simplicity and small size, such native code seems sufficient for a first obfuscation layer, yet the resulting encryption is far from being cryptographically strong. For short keys up to 2 bytes the obfuscation can be trivially broken using brute-force attacks. However, uncovering malware obfuscated with longer keys, as for example the 4-byte key in Figure 1, still necessitates manually reverse engineering the code or dynamically analyzing the malicious document in a sandbox with vulnerable versions of the target application [e.g., 7, 17, 20]. While both approaches are effective in removing the obfuscation layer, they require a considerable amount of time in practice and are thus not suitable for analyzing and detecting embedded malware at end hosts.

In this paper, we present KANDI, a method capable of efficiently breaking Vigenère-based obfuscations and automatically uncovering embedded malware in documents without the need to parse the document's file format. The method leverages concepts from classic cryptography in order to conduct a probable-plaintext attack against common variants of the Vigenère cipher. To this end, the method first approximates the length of possible keys and then computes so-called *difference streams* of the document and plaintexts likely contained in malware binaries. These plaintexts are automatically retrieved beforehand and may include fragments of the PE header, library names and common code stubs. Using these streams it is possible to look for the plaintexts directly in the obfuscated data. If sufficient matches are identified, KANDI automatically derives the obfuscation key and reveals the full embedded code for further analysis, for example, by an anti-virus scanner or a human expert.

We demonstrate the efficacy of this approach in an empirical evaluation with documents of different formats and real malware. In a controlled experiment KANDI is able to break obfuscations using XOR and ADD/SUB with keys up to 13 bytes. On a collection of real-world malware in Word, Powerpoint and RTF documents with unknown obfuscation, KANDI is able to deobfuscate every fourth document and exposes the contained malware binary, including several

```
00:  be XX XX XX XX                 mov   edx, ADDRESS
05:  31 db                          xor   ebx, ebx
07:  81 34 1e XX XX XX XX     start: xor   dword [edx + ebx], KEY
0e:  81 c3 04 00 00 00              add   ebx, 0x04
14:  81 fb XX XX XX XX              cmp   ebx, LENGTH
1a:  7c eb                          jl    start
```

Fig. 1. Example of native code for a Vigenère-based obfuscation. The code snippet deobfuscates data at `ADDRESS` of length `LENGTH` using the 4-byte key `KEY`. For simplicity we omit common tricks to avoid null bytes in the code.

samples of the recent attack campaign MiniDuke [6]. Moreover, KANDI is significantly faster than dynamic approaches and enables scanning documents and deobfuscating malware at a throughput rate of 16.46 Mbit/s, corresponding to 5 documents of ∼400 kB per second.

It is necessary to note that KANDI targets only one of many possible obfuscation strategies. If a different form of obfuscation is used or no plaintexts are known in advance, the method obviously cannot uncover obfuscated data. We discuss these limitations in Section 5 specifically. Nonetheless, KANDI defeats a prevalent form of obfuscation in practice and thereby provides a valuable extension to current methods for the analysis of targeted attacks and embedded malware in the wild.

The rest of this paper is organized as follows: Obfuscation using Vigenère ciphers and classic cryptanalysis are reviewed in Section 2. Our method KANDI is introduced in Section 3 and an empirical evaluation of its capabilities is presented in Section 4. We discuss limitations and related work in Section 5 and 6, respectively. Section 7 concludes the paper.

2 Obfuscation and Cryptanalysis

The obfuscation of code can be achieved using various techniques, ranging from simple encodings to strong ciphers and emulator-based packing. Implementations of complex techniques, however, often contain characteristic patterns and thus increase the risk of detection by anti-virus scanners [4]. As a consequence, simple encodings and weak ciphers are still widely used for obfuscation despite their shortcomings. In the following section we investigate a specific type of such basic obfuscation, which is frequently used to hide malware in documents.

2.1 Vigenère-based Obfuscation

The substitution of bytes using XOR and ADD/SUB—a variant of so-called *Vigenère ciphers* [19]—is one of the simplest yet widely used obfuscation techniques. These ciphers are regularly applied for cloaking shellcodes and embedded malware. Figure 1 and 2 show examples of these ciphers in x86 code.

```
start:  mov  al, byte [edx]
        add  al, ADD_KEY
        rol  al, ROL_KEY
        mov  byte [edx], al
        inc  edx
        cmp  edx, LENGTH
        jl   start
```

(a) Obfuscation using ADD and ROL

```
start:  mov   al, byte [PTR + ebx]
        sub   byte [edx], al
        inc   ebx
        and   ebx, 0x0f
        inc   edx
        cmp   edx, LENGTH
        jl    start
```

(b) Obfuscation with 16-byte key

Fig. 2. Code snippets for Vigenère-based obfuscation: (a) Data stored at `[edx]` is obfuscated using ADD and ROL, (b) Data stored at `[edx]` is obfuscated using SUB with the 16-byte key at `PTR`.

Due to the implementation with only a few instructions, Vigenère-based obfuscation keeps a small footprint in the code, thereby complicating the task of extracting reliable signatures for anti-virus scanners. Additionally, this obfuscation is fast, easily understandable and good enough to seemingly protect malicious code in the first layer of obfuscation. Despite these advantages Vigenère ciphers suffer from several well-known weaknesses.

Definition of Vigenère Ciphers. Before presenting attacks against Vigenère-based obfuscation, we first need to introduce some notation and define the family of Vigenère ciphers studied in this work. We consider the original code of a malware binary as a sequence of n bytes $M_1 \dots M_n$ and similarly represent the resulting obfuscated data by $C_1 \dots C_n$. When referring to cryptographic concepts, we sometimes denote the original code as plaintext and refer to the obfuscated data as ciphertext. The Vigenère-based obfuscation is controlled using a key $K_1 \dots K_l$ of l bytes, where l usually is much smaller than n. Moreover, we use $\hat{K}_i = K_{(i \bmod l)}$ to access the individual bytes of the key.

Using this notation, we can define a family of Vigenère ciphers, where each byte M_i is encrypted with the key byte $\hat{K}_i$ using the binary operation $\circ$ and decrypted using its inverse operation $\circ^{-1}$, as follows:

$$C_i = M_i \circ \hat{K}_i \quad \text{and} \quad M_i = C_i \circ^{-1} \hat{K}_i.$$

This simple definition covers several variants of the Vigenère cipher, as implementations only differ in the choice of the two operations $\circ$ and $\circ^{-1}$. For example, if we define $\circ$ as addition and $\circ^{-1}$ as subtraction, we obtain the classic form of the Vigenère cipher. Table 1 lists binary operations that are frequently used for obfuscating malicious code. Note that a subtraction can be expressed as an addition with a negative element and thus is handled likewise.

Table 1. Operators of Vigenère ciphers used for obfuscation.

Operation	**Encryption** $\circ$	**Decryption** $\circ^{-1}$
Addition (ADD)	$(X + Y) \mod 256$	$(X - Y) \mod 256$
Subtraction (SUB)	$(X - Y) \mod 256$	$(X + Y) \mod 256$
Exclusive-Or (XOR)	$X \oplus Y$	$X \oplus Y$

Theoretically, any pair of operations that is inverse to each other can be used to construct a Vigenère cipher. In practice, most implementations build on logic and arithmetic functions that induce a *commutative group* over bytes. That is, the operation $\circ$ is commutative and associative as well as there exists an identity element and inverse elements providing the operation $\circ^{-1}$. These group properties are crucial for different types of efficient attacks as we will see in Sections 2.2 and 2.4. Note that ROL and ROR instructions are not commutative and thus are treated differently in the implementation of our method KANDI presented in Section 3.

Another important observation is that some bytes are encrypted with the same part of the key. In particular, this holds true for every pair of bytes M_i and M_j whose distance is a multiple of the key length, that is, $i \equiv j \pmod{l}$. This repetition of the key is a critical weakness of Vigenère ciphers and can be exploited to launch further attacks that we discuss in Sections 2.3 and 2.4.

With these few basic definitions in mind, we can pursue three fundamentally different approaches for attacking Vigenère ciphers: (1) *brute-force attacks and heuristics*, (2) *ciphertext-only attacks* and (3) *probable-plaintext attacks*. In the following, we discuss each of these attack types in detail and check whether they are applicable for deobfuscating embedded malware.

2.2 Brute-force Attacks and Heuristics

A straightforward way of approaching malware obfuscations is to brute-force the key used by the malware author. There are two basic implementations for such an attack: First, one encrypts all plaintext patterns that are assumed to be present in the original binary with each and every key and tries to match those. Second, one decrypts the binary or parts of it and looks for the presence of the plaintext as a usual signature engine would do. In both cases a valid key is derived if a certain amount of plaintexts match. For short keys, this approach is both fast and effective. In practice, brute-force attacks prove to be a valuable tool for analyzing malware obfuscated using keys up to 2 bytes [3, 26].

Theoretically, an exhaustive search over the complete key space can be used to also derive keys with more than 2 bytes. However, this obviously comes at the price of runtime performance. For a key length of only 4 bytes there are more than 4.2 billion combinations that need to be checked in the worst case. This clearly exceeds the limits of what is possible in the scope of the deobfuscation of embedded malware. Even worse, 4-byte and 8-byte keys fit the registers of common CPU architectures and therefore, do not require much different deobfuscation routines. In fact, the underlying logic is identical to the use of single-byte keys and the code size is only marginally larger as illustrated in Figure 1.

A more clever way of approaching the problem is by relying on the structure of embedded malware binaries, which are often PE files. In this format `\x00` bytes are used as padding for sections and headers which gives rise to a heuristic. We recall from Section 2.1 that the binary operation $\circ$ has an identity element, which simply is 0 for XOR as well as ADD instructions. Therefore, whenever a large block of `\x00` bytes is encrypted, the key is revealed multiple times and can be read off without extra effort. Hence, once a highly repetitive string is spotted in obfuscated data, deobfuscation is a simple task for a malware analyst. According to our tests the very same technique is leveraged in a proprietary system for the analysis of malware called *Cryptam* [16]. While effective in many cases when a full binary including padding is obfuscated, this heuristic fails when a malware does not encrypt `\x00` bytes. Furthermore, such an approach cannot differ between variants of Vigenère ciphers. Since XOR and ADD have the same identity element, there is no way to decide which one was used for obfuscation in this setting.

2.3 Ciphertext-Only Attacks

A more advanced type of classic attacks against Vigenère ciphers only makes use of the ciphertext. Some of these attacks can be useful for determining the length of the obfuscation key, whereas others even enable recovering the key if certain conditions hold true in practice.

Index of Coincidence. A classic approach for determining the key length from ciphertext only is the *index of coincidence*, commonly denoted as κ [9, 10]. Roughly speaking it represents the ratio of how many bytes happen to appear at the same positions if you shift data against itself. Formally, the index of coincidence is defined as

$$\kappa = \frac{\sum_{i=1}^{256} f_i(f_i - 1)}{n(n-1)},$$

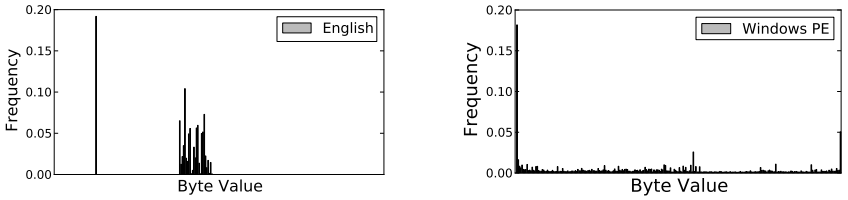
where f_i are the byte frequencies in data of n bytes. Under the condition that we know the index of some plaintext κ_p we are able to infer the key length l of the Vigenère cipher. It is estimated as the ratio of the differences of κ_p to the index of random data κ_r and the ciphertext κ_c:

$$l \approx \frac{\kappa_p - \kappa_r}{\kappa_c - \kappa_r}.$$

The Kasiski Examination. Another ciphertext-only attack for determining the key length is the so-called *Kasiski examination* [12]. The underlying assumption of this method is that the original plaintext contains some identical substrings. Usually these patterns would be destroyed by the key; however, if two instances of such substrings are encrypted with the same portion of the key, the encrypted data contains a pair of identical substrings as well. This implies that the distance between the characters of these substrings is a multiple of the key length. Thus, by gathering identical substrings in the ciphertext, it is possible to support an assumption about the key length.

Key Recovery using Frequency Analysis. Natural languages tend to have a very characteristic frequency distribution of letters. For instance, in the English language the letter e is with more than 12% the significantly most frequent letter in the alphabet [14]. Only topped by the space character, which is used in written texts in order to separate words.

This frequency distribution can be exploited to derive the key used for the encryption. As one can easily imagine, the actual frequency distribution does not change by simply replacing one character with another as in the case of a key of length $l = 1$. The larger the key length gets, the more the distribution is flattened out because identical letters may be translated differently depending on their position in the text. However, since it is possible to determine the length of the key beforehand, one can perform the very same frequency analysis on all characters that were encrypted with the same single-byte key $\hat{K}_i$.



(a) Distribution of English text (b) Distribution of PE files

Fig. 3. The byte frequency distributions of English text and Windows PE files.

Although effective in decrypting natural language text, key recovery using frequency analysis is not suitable for deobfuscating embedded malware. If the obfuscated code corresponds to regular PE files, the byte frequencies are almost equally distributed and can hardly be discriminated, because executable code, header information and other types of data are mixed in this format. As an example, Figure 3 shows the byte frequency distributions of English text and PE files, where except for a peak at `\x00` the distribution of PE files is basically flat. The presented ciphertext-only attacks thus only provide means for determining the key length of Vigenère-based obfuscation, but without further refinements are not appropriate for actually recovering the key.

2.4 Probable-Plaintext Attacks

To effectively determine the key used in a Vigenère-based obfuscation, we consider classic attacks based on known and probable plaintexts. We refer to these attacks as *probable-plaintext attacks*, as we cannot guarantee that a certain plaintext is indeed contained in an obfuscated malware binary.

Key Elimination. In particular, we consider the well-known technique of *key elimination*. The idea of this technique is to determine a relation between the plaintext and ciphertext that does not involve the key: Namely, the *difference* of bytes that are encrypted with the same part of the key. Formally, for a key byte $\hat{K}_i$ this difference can be expressed using the inverse operation $\circ^{-1}$ as:

$$C_i \circ^{-1} C_{i+l} = (M_i \circ \hat{K}_i) \circ^{-1} (M_{i+l} \circ \hat{K}_i) = M_i \circ^{-1} M_{i+l}.$$

Note that this relation of differences only applies if the operator used for the Vigenère cipher induces a commutative group. For example, if we plug in the popular instructions XOR and ADD from Table 1, the difference of the obfuscated bytes C_i and C_{i+l} allows to reason about the difference of the corresponding plaintext bytes:

$$C_i \oplus C_{i+l} = (M_i \oplus \hat{K}_i) \oplus (M_{i+l} \oplus \hat{K}_i) = M_i \oplus M_{i+l}$$
$$C_i - C_{i+l} = (M_i + \hat{K}_i) - (M_{i+l} + \hat{K}_i) = M_i - M_{i+l}.$$

Based on this observation, we can implement an efficient probable-plaintext attack against Vigenère ciphers. Given a plaintext $P = P_1 \dots P_m$, we introduce

the *difference streams* ΔP and ΔC. If the difference streams match at a specific position and the plaintext P is sufficiently large, we have successfully determined the occurrence of a plaintext in the obfuscated data. In particular, we compute the difference stream

$$\Delta P = (P_1 \circ^{-1} P_{1+l}) \dots (P_{m-l} \circ^{-1} P_m)$$

for the plaintext P and compare it against each position i of the ciphertext C using the corresponding stream

$$\Delta C = (C_i \circ^{-1} C_{i+l}) \dots (C_{i+m-l} \circ^{-1} C_{i+m}).$$

Using this technique, we can efficiently search for probable plaintexts in data obfuscated using a Vigenère cipher without knowing the key. This enables us to check for common strings in the obfuscated code, such as header information, API functions and code stubs. Once the position of a probable plaintext is found it is possible to derive the used key by applying the appropriate inverse operation: $K_j = C_{i+j} \circ^{-1} P_{i+j}$ with i being the position where the difference stream of a probable plaintext matches. The more plaintexts match in the obfuscated code, the more reliably the key can finally be determined.

3 Deobfuscating Embedded Malware

After describing attacks against Vigenère ciphers, we now present our method KANDI that combines and extends these attacks for deobfuscating embedded malware. The three basic analysis steps of KANDI are described in the following sections and outlined in Figure 4. First, our method extracts probable plaintexts from a representative set of code (Section 3.1). Applied to an unknown document, it then attempts to estimate the key length (Section 3.2) and finally break any Vigenère-based obfuscation if present in the file (Section 3.3).

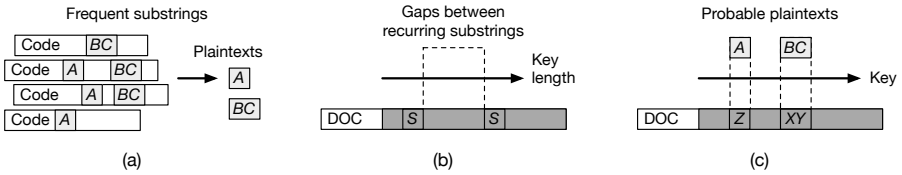


Fig. 4. Schematic depiction of KANDI and its analysis steps: (a) Extraction of plaintexts, (b) derivation of the key length and (c) probable-plaintext attack.

In particular, we are using the *Kasiski examination* for determining the key length in step (b) and the technique of *key elimination* against XOR and ADD/SUB substitutions in step (c). Additionally, we are testing each possible transposition for ROL/ROR instructions. We consider this a legit compromise since there exists only a few combinations to check.

3.1 Extraction of Plaintexts

The deobfuscation performance of KANDI critically depends on a representative set of probable plaintexts. In the scope of this work, we focus on Windows PE files, as these are frequently used as initial step of an attack based on infected documents. However, our method is not restricted to this particular type of data and can also be applied to other representations of code from which probable plaintexts can be easily extracted, such as DEX files and ELF objects.

In the first step, we thus extract the most common binary strings found in PE files distributed with off-the-shelf Windows XP and Windows 7 installations. Profitable plaintexts are, for instance, the DOS stub and its text, API strings, library names or code patterns such as push-call sequences. To determine these strings efficiently, we process the collected PE files using a suffix array and extract all binary strings that appear in more than 50% of the files. Additionally, we filter the plaintexts according to the following constraints:

- **Plaintext length.** In order to ensure an expressive set of probable plaintext, we require that each plaintext is at least 4 bytes long.
- **Zero bytes.** As described in Section 2.2, a disadvantage of common heuristics is that they are not able to deal with malware that does not obfuscate \x00 byte regions. In order not to suffer from the very same drawback, we completely exclude \x00 bytes and reject plaintexts containing them.
- **Byte repetitions.** We also exclude plaintexts that contain more than four repetitions of a single byte. These might negatively influence the key elimination as described in Section 2.4.

We are well aware and acknowledge that there exist more sophisticated ways to extract probable plaintexts. This for instance is day-to-day business of the anti-virus industry when generating signatures for their detection engines. Also, well-known entrypoint stubs as well as patterns from specific compilers, packers and protectors might represent valuable probable plaintexts.

3.2 Deriving the Key Length

In the second step, KANDI uses the Kasiski examination (Section 2.3) to inspect the raw bytes of a document—without any further parsing or processing of the file. The big advantage of this method over the index of coincidence proposed by Friedman [9] is that we neither need to rely on the byte distribution of the original binary nor do we have to precisely locate the embedded malware. Furthermore, the Kasiski examination allows us to take multiple candidates of the key length into consideration. Depending on the amount of identical substrings that suggest a particular key length, we construct a ranking of candidates for later analysis. That way, it is possible to compensate for and recover from misinterpretations.

However, finding pairs of identical substrings in large amounts of data needs careful algorithm engineering in order to work efficiently. We again make use of suffix arrays for determining identical substrings in linear time in the length

of the analyzed document. Since the Kasiski examination only states that the distances between identical substrings in the ciphertext refer to *multiples* of the key length, it is necessary to also examine the integer factorization thereof. Fortunately, there exists a shortcut to this factorization step that works very well in practice: If KANDI returns a key that repeats itself, e.g. `13 37 13 37`, this indicates that we correctly derived the key but under an imprecise assumption of the key length ($l = 4$ rather than 2). In such cases we simply collapse the repeating key and correct the key length accordingly.

3.3 Breaking the Obfuscation

Equipped with an expressive set of probable plaintexts and an estimation of the key length, it is now possible to mount a probable-plaintext attack against Vigenère-based obfuscation. The central element of this step is the key elimination introduced in Section 2.4. It enables us to look for probable plaintexts within the obfuscated data and derive the used key automatically. Again, KANDI directly operates on the raw bytes of a document and thereby avoids parsing the file.

Robust Key Recovery. If a probable plaintext is longer than the estimated key length, the overlapping bytes can be used to reinforce our assumption about the key. To this end, we define the *overlap ratio* r that is used to specify how certain we want to be about a key candidate. The larger r is, the stricter KANDI operates and the more reliable is the key. If we set $r = 0.0$, a usual match of plaintexts is enough to support the evidence of a key candidate. This means that we will end up with a larger amount of possibly less reliable hints. Our experiments show that for the grand total incorrect guesses will average out and in many cases it is possible to reliably deobfuscate embedded malware.

If a more certain decision is desired the overlap ratio r can be increased. However, for larger values of r we require longer probable plaintexts: $r = 0.0$ only requires a minimal overlap, $r = 0.5$ already half of the probable plaintext's length and $r = 1.0$ twice the size. As an example, if the estimated key length is 4 and $r = 0.5$, only plaintexts of at least 6 bytes are used for the attack. Depending on the approach chosen to gather probable plaintexts, it might happen that the length of the available plaintexts ends up being the limiting factor for the deobfuscation. We will evaluate this in the next section.

Incorporating ROL and ROR. Finally, in order to increase the effectiveness of KANDI, we additionally consider transpositions using ROL and ROR instructions. ROL and ROR are each others inverse function, that is, when iterating over all possible shift offsets they generate exactly the same output but in different order. Furthermore, in most implementations these instructions operate on 8 bits only such that the combined overall number of transpositions to be tested is very small. Consequently, we simply add a ROL shift as a preprocessing step to KANDI. Although we attempt to improve over a plain brute-force approach for breaking obfuscation, we consider the 7 additional tests as a perfectly legit tradeoff from a pragmatic point of view.

We are also well aware that it is possible to render our method less effective by making use of chaining or adding other computational elements that are not defined in the scope of Vigenère ciphers and therefore out of reach for KANDI. We discuss this limitation in Section 5. Nevertheless, our evaluation shows that we are able to deobfuscate a good deal of embedded malware in the wild, including recent samples of targeted attack campaigns, such as MiniDuke [6]. Thereby, KANDI proves to be of great value for day-to-day business in malware analysis.

4 Evaluation

We proceed to evaluate the deobfuscation capabilities and runtime performance of KANDI empirically. Since it is hard to determine whether embedded malware in the wild is actually using Vigenère-based obfuscation or not, we start off with a series of controlled experiments (Section 4.1). We then continue to evaluate KANDI on real-world malware in Word, Powerpoint and RTF documents as well as different image formats (Section 4.2). We need to stress that this collection contains malware with unknown obfuscation. Nonetheless, KANDI is able to expose obfuscated malware in every fourth file, thereby empirically proving that (a) Vigenère ciphers are indeed used in the wild and (b) that our method is able to reliably reveal the malicious payload in these cases.

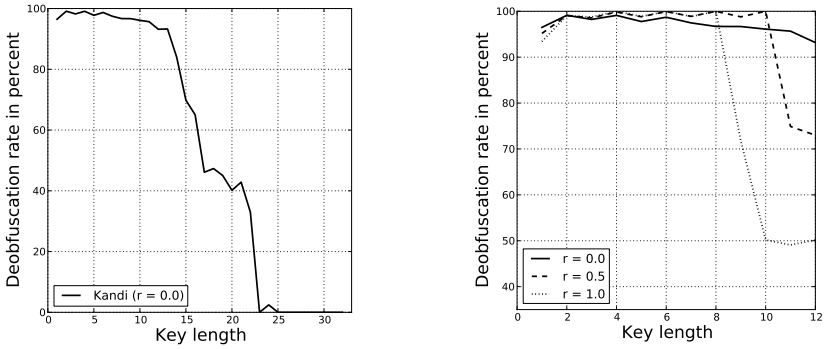
4.1 Controlled Experiments

To begin with, we evaluate KANDI in a controlled setting with known ground truth, where we are able to exactly tell if a deobfuscation attempt was successful or not. In particular, we conduct two experiments: First, we obfuscate plain Windows PE files and apply KANDI to them. In the course of that, we measure the runtime performance and throughput of our approach. Second, the obfuscated PE files are embedded in benign Word documents in order to show that KANDI not only works on completely encrypted data, but is also capable of deobfuscating files embedded inside of documents.

Evaluation Datasets. In order to create a representative set of PE files for the controlled experiments, we simply gather all PE files in the system directories of Windows XP SP3 (`system` and `system32`) and Windows 7 (`System32` and `SysWOW64`). This includes stand-alone executables as well as libraries and drivers and yields a total of 4,780 files. We randomly obfuscate each of the PE files with a Vigenère cipher using either XOR, ADD or SUB. We draw random keys for this obfuscation and vary the key length from 1 to 32 bytes, such that we finally obtain 152,960 ($32 \times 4{,}780$) unique obfuscated PE files.

To study the deobfuscation of embedded code, we additionally retrieve one unique and malware-free Word document for each PE file from VirusTotal and use it as host for the embedding. Malware appearing in the wild would be embedded at positions compliant with the host's file format. This theoretically provides valuable information where to look for embedded malware. As KANDI

does not rely on parsing the host file, we simply inject the obfuscated PE files at random positions. We end up with a total of 152,960 unique Word documents each containing an obfuscated PE file.



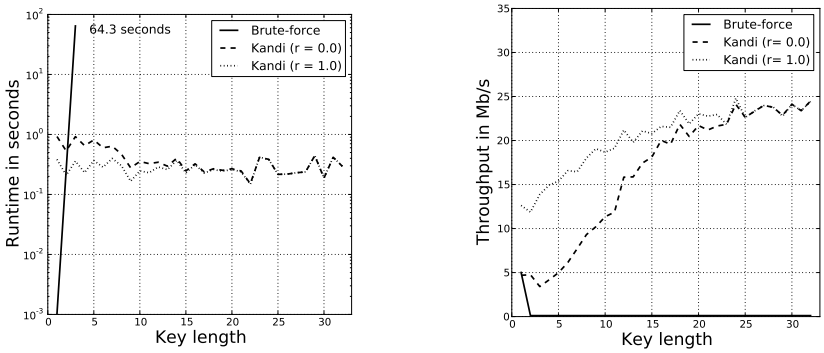
(a) Obfuscated PE files (b) Influence of the overlap ratio

Fig. 5. Deobfuscation performance of KANDI on obfuscated PE files. Figure (b) shows the performance for different overlap ratios.

Deobfuscation of Obfuscated PE Files. To demonstrate the capability of our method to break Vigenère-based obfuscations, we first apply KANDI to the 152,960 obfuscated PE files. The probable plaintexts for this experiment are retrieved as described in Section 3.1 without further refinements. Figure 5(a) shows results for this experiment, where the key length is plotted against the rate of deobfuscated PE files. For key lengths up to 13 bytes, the obfuscation can be reliably broken with a success rate of 93% and more. This nicely illustrates the potential of KANDI to automatically deobfuscate malware. We also observe that the performance for keys longer than 13 bytes drops. While our approach is not capped to a specific key length, the limiting factor at this point is the collection of plaintexts and in particular the length of those.

To study the impact of the plaintext length, we additionally apply KANDI with different values for the overlap ratio r as introduced in Section 3.3. The corresponding deobfuscation rates are visualized in Figure 5(b). Although a high value of r potentially increases the performance, it also reduces the number of plaintexts that can be used. If there are too few usable plaintexts, it gets difficult to estimate the correct key. As a result, KANDI attains a deobfuscation performance of almost 100% for $r = 1.0$ if the keys are short, but is not able to reliably break obfuscations with longer keys.

Runtime Performance. We additionally examine the runtime performance of KANDI. For this purpose, we randomly draw 1,000 samples from the obfuscated PE files for each key length and repeat the previous experiment single-threaded on an Intel Core i7-2600K CPU at 3.40GHz running Ubuntu 12.04. As baseline for this experiment, we implement a generic brute-force attack that is applied to



(a) Average runtime per file (b) Average throughput

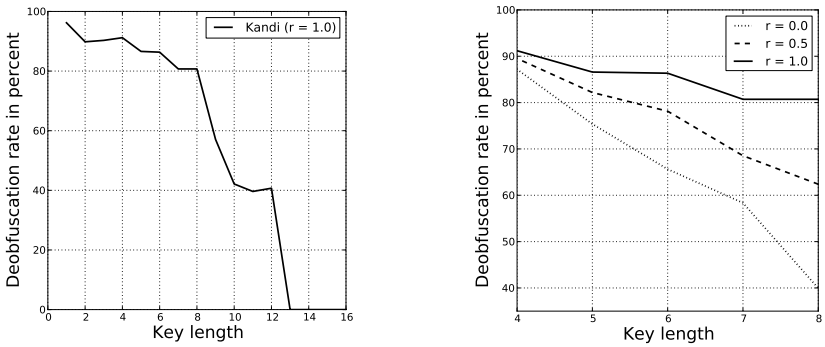
Fig. 6. Runtime performance of KANDI in comparison to a brute-force attack on a batch of 1,000 randomly drawn obfuscated PE files.

the first 256 bytes of each file. Due to the defined starting point and the typical header structure of PE files 256 bytes are already sufficient to reliably break the obfuscation in this setting. Note that this would not be necessarily the case for *embedded* malware.

The results of this experiment are shown in Figure 6 where the runtime and throughput of each approach are shown on the y-axis and the key length on the x-axis. Obviously, the brute-force attack is only tractable for keys of at most 3 bytes. By contrast, the runtime of KANDI does not depend on the key length and the method attains a throughput of 16.46 Mbit/s on average, corresponding to an analysis speed of 5 files of ∼400 kB per second. Consequently, KANDI's runtime is not only superior to brute-force attacks but also significantly below dynamic approaches like OmniUnpack [17] or PolyUnpack [18] and thus beneficial for analyzing embedded malware at large scales.

Deobfuscation of Injected PE Files. As last controlled experiment, we study the deobfuscation performance of KANDI when being operated on obfuscated PE files that have been injected into Word documents. Figure 7(a) shows the results of this experiment. For keys with up to 8 bytes, our method deobfuscates most of the injected PE files—without requiring the document to be parsed. Moreover, we again inspect the influence of the overlap ratio r in this setting. Similar to the previous experiment, a larger value of r proves beneficial for short keys, such that keys up to 8 bytes are broken with a success rate of 81% and more. This influence of the overlap ratio gets evident for keys between 4 and 8 bytes as illustrated Figure7(b). For keys of length $l = 8$ a high value of r even doubles the deobfuscation performance in comparsion to the default setting.

Due to this, we use an overlap ratio of $r = 1.0$ for the following experiments on real-world malware. We expect embedded malware found in the wild to mainly use keys of 1 to 8 bytes. The reasons for this assumption is that such keys fit into CPU registers and therefore implementations are more compact. Furthermore, 4-byte keys are already intractable for brute-force attacks.



(a) Obfuscated PE files (b) Influence of the overlap ratio

Fig. 7. Deobfuscation performance of KANDI on Word documents containing obfuscated PE files. Figure (b) shows the performance for different overlap ratios.

4.2 Real-World Experiments

To top off our evaluation we proceed to demonstrate how KANDI is able to deobfuscate and extract malware from samples seen in the wild. To this end, we have acquired four datasets of real-world malware embedded in documents and images with different characteristics.

Table 2. Overview of the four datasets of malicious documents and images.

Dataset name	Type	Formats	Samples
Exploits 1	Documents	`DOC, PPT, RTF`	992
Exploits 2	Documents	`DOC, PPT, RTF`	237
Dropper 1	Documents	`DOC, PPT, RTF`	336
Dropper 2	Images	`PNG, GIF, JPG, BMP`	52
Total			1,617

Malware Datasets. Embedded malware is typically executed by exploiting vulnerabilities in document viewers. For the first dataset (*Exploits 1*) we thus retrieve all available Word, Powerpoint and RTF documents from VirusTotal that are detected by an anti-virus scanner and whose label indicates the presence of an exploit, such as `exploit.msword` or `exploit.ole2`. Similarly, we construct the second dataset (*Exploits 2*) by downloading all documents that are tagged with one of the following CVE numbers: `2003-0820`, `2006-2492`, `2010-3333`, `2011-0611`, `2012-0158` and `2013-0634`.

As our method specifically targets PE files embedded in documents, we additionally compose two datasets of malware droppers. The first set (*Dropper 1*) contains all available Word, Powerpoint and RTF documents that are detected by an anti-virus scanner and whose label contains the term `dropper`. The second dataset (*Dropper 2*) is constructed similarly by retrieving all malicious images

labeled as dropper. An overview of all four datasets is given in Table 3. We deliberately exclude malicious PDF files from our analysis, as this file format allows to incorporate JavaScript code. Consequently, the first layer of obfuscation is often realized using JavaScript encoding functions, such as Base64 and URI encoding. Such encodings are not available natively for other formats and hence we do not consider PDF files in this work.

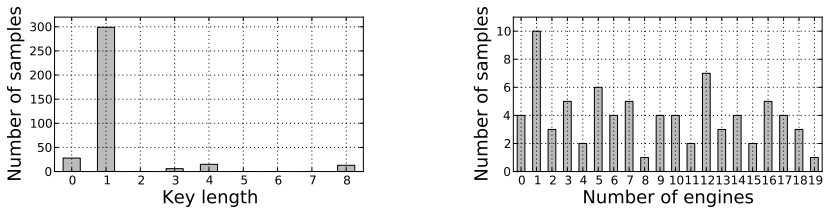
Table 3. Deobfuscation performance of KANDI on real-world malware. The last columns detail the number of samples that were successfully deobfuscated.

Dataset	Not Obfuscated	Obfuscated	Deobfuscated by Kandi	
Exploits 1	211	781	180	23.1%
Exploits 2	35	203	64	31.7%
Dropper 1	86	250	81	32.4%
Dropper 2	27	25	9	36.0%
Total	359	1,258	334	26.6%

Deobfuscation of Embedded Malware. We proceed to apply KANDI to the collected embedded malware. Due to minor modifications by the malware author, it is not always possible to extract a valid PE file. To verify if a deobfuscation attempt was successful we thus utilize a PE checker based on strings such as Windows API function (e.g. `LoadLibrary`, `GetProcAddress`, `GetModuleHandle`) and library names as found in the import table (e.g. `kernel32.dll`, `user32.dll`) Additionally, we look for the MZ and PE header signatures and the DOS stub. We consider a deobfuscation successful if either a valid PE file is extracted or at least five function or library names are revealed in the document.

We observe that for 359 of the samples no deobfuscation is necessary, as the embedded malware is present in clear. KANDI identifies such malware by simply returning an obfuscation key of `0x00`. We support this finding by applying the PE checker described earlier. The remaining 1,258 samples are assumed to be obfuscated. Every fourth of those samples contains malware obfuscated with the Vigenère cipher and is deobfuscated by KANDI. That is, our method automatically cracks the obfuscation of 334 samples and extracts the embedded malware—possibly multiple files per sample. Table 3 details the results for the individual datasets. A manual analysis of the remaining files on a sample basis does not reveal obvious indicators for the Vigenère cipher and we conclude that KANDI deobfuscates most variants used in real-world embedded malware.

Figure 8(a) shows the distribution of the key lengths discovered by KANDI. The majority of samples is obfuscated with a single-byte key and seems to be in reach for brute-forcing. However, to do so one would need to precisely locate the encrypted file, which is not trivial. Moreover, our method also identifies samples with longer keys ranging from 3 to 8 bytes that would have been missed without the help of KANDI. Rather surprising are those samples that use 3 bytes as a key. One would suspect these to be false positives, but we have manually verified that these are correctly deobfuscated by our method.



(a) Detected key lengths (b) VirusTotal detections

Fig. 8. (a) Distribution of key lengths detected by KANDI; (b) Number of anti-virus scanners detecting the extracted malware binaries.

As the final step of this experiment, we analyze the extracted malware binaries with 46 different anti-virus scanners provided by VirusTotal. Since some of these scanners are prone to errors when it comes to manipulated PE headers, we consider only those 242 deobfuscated malware binaries that are valid PE files (conform to the format specification). The number of detections for each of these files is shown in Figure 8(b). Several binaries are poorly detected by the anti-virus scanners at VirusTotal. For instance, 19% (46) of the binaries are identified by less than 10 of the available scanners. This result suggests that the extracted binaries are unkown to a large portion of the anti-virus companies—likely due to the lack of tools for automatic deobfuscation.

Finally, the analyzed binaries also contain several samples of the MiniDuke malware discovered in early February 2013 [6]. A few months back, this threat has been completely unknown, such that we are hopeful that binaries deobfuscated by KANDI help the discovery of new and previously unknown malware.

5 Limitations

The previous evaluation demonstrates the capabilities of KANDI in automatically deobfuscating embedded malware. Our approach targets a specific form of obfuscation and thus cannot uncover arbitrarily obfuscated code in documents. We discuss limitations resulting from this setting in the following and present potential extensions of KANDI.

Obfuscation with Other Ciphers. Our approach builds on classic attacks against Vigenère ciphers. If a different cryptographic cipher is used for the obfuscation, our method obviously cannot recover the original binary. For example, the RC4-based obfuscation used in the trojan Taidoor [28] is resistant against probable-plaintext attacks as used for KANDI. However, the usage of standard cryptographic primitives, such as RC4 and AES, can introduce detectable patterns in native code and thereby expose the presence of embedded malware in documents [see 4]. To stay under the radar of detection tools, attackers need to carefully balance the strength of obfuscation and its detectability, which provides room for further cryptographic attacks.

Availability of Plaintexts. The efficacy of probable-plaintext attacks critically depends on a sufficiently large set of plaintexts. If no or very few plaintexts are available, the obfuscation cannot be reliably broken. As a result, attackers might try to eliminate predicable plaintexts from their code, for example, by removing header information or avoiding common libraries. Designing malware that does not contain predictable plaintexts is feasible but requires to expend considerable effort. In practice, many targeted attacks therefore use multiple layers of obfuscation, where only few indicative patterns are visible at each layer. Our evaluation demonstrates that this strategy is often insufficient, as KANDI succeeds in breaking the obfuscation of every fourth sample we analyzed.

Other Forms of Vigenère-based Obfuscation. Our implementation of KANDI is designed to deobfuscate streams of bytes as generated by native obfuscation code. Consequently, the method cannot be directly applied to other encodings, as for example employed in malicious PDF documents using JavaScript code. However, with only few modifications, KANDI can be extended to also support other streams of data, such as unicode characters (16 bit) and integers (32 bit). In combinations with techniques for detection and normalization of common encodings, such as Base64 and URI encoding, KANDI might thus also help in breaking Vigenère-based obfuscations in PDF documents and drive-by-download attacks. However, extending the Vigenère cipher by, for instance, introducing chaining defines a different (although related) obfuscation and cannot be handled with the current implemention of KANDI. We leave this to future work.

6 Related Work

The analysis of embedded malware has been a vivid area of research in the last years, in particular due to the increasing usage of malicious documents in targeted attacks [e.g., 1, 6, 28]. Several concepts and techniques have been proposed to locate and examine malicious code in documents. Our approach is related to several of these, as we discuss in the following.

Analysis of Embedded Malware. First methods for the identification of malware in documents have been proposed by Stolfo et al. [27] and Li et al. [15]. Both make use of content-based anomaly detection for learning profiles of regular documents and detecting malicious content as deviation thereof. This work has been further extended by Shafiq et al. [21], which refine the static analysis of documents to also locate the regions likely containing malware. Although effective in spotting suspicious content, these methods are not designed to deobfuscate code and thus are unsuitable for in-depth analysis of embedded malware.

Another branch of research has thus studied methods for analyzing malicious documents at runtime, thereby avoiding the direct deobfuscation of embedded code [e.g., 8, 15, 20]. For this dynamic analysis, the documents under investigation are opened in a sandbox environment, such that the behavior of the application processing the documents can be monitored and malicious activities detected. These approaches are not obstructed by obfuscation and can

reliably detect malicious code in documents. The monitoring at run-time, however, induces a significant overhead which is prohibitive for large-scale analysis or detection of malware at end hosts.

Recently, a large body of work has focused on malicious PDF documents. Due to the flexibility of this format and its support for JavaScript code, these documents are frequently used as vehicles to transport malware [25]. Several contrasting methods have been proposed to spot attacks and malware in JavaScript code [e.g., 5, 13] and the structure of PDF files [e.g., 23, 29]. While some malicious PDF documents make use of Vigenère-based obfuscation, other hiding strategies are more prominent in the wild, most notably the dynamic construction of code. As a consequence, we have not considered PDF documents in this work, yet the proposed deobfuscation techniques also apply to Vigenère ciphers used in this document format.

Deobfuscating and Unpacking Malware. Aside from specific work on embedded malware, the deobfuscation of malicious code has been a long-standing topic of security research. In particular, several methods have been developed to dynamically unpack malware binaries, such as PolyUnpack [18], OmniUnpack [17] and Ether [7]. These methods proceed by monitoring the usage of memory and identifying unpacked code created at runtime. A similar approach is devised by Sharif et al. [22], which defeats emulation-based packers using dynamic taint analysis. These unpackers enable a generic deobfuscation of malicious code, yet they operate at runtime and, similar to the analysis of documents in a sandbox, suffer from a runtime overhead.

Due to the inherent limitations of static analysis, only few approaches have been proposed that are able to statically inspect obfuscated malware. An example is the method by Jacob et al. [11] that, similar to Kandi, exploits statistical artifacts preserved through packing in order to analyze malware. The method does not focus on deobfuscation but rather efficiently comparing malware binaries and determining variants of the same family without dynamic analysis.

Probable-Plaintext Attacks. Attacks using probable and known plaintexts are among the oldest methods of cryptography. The Kasiski examination used in Kandi dates back to 1863 [12] and similarly the key elimination of Vigenère ciphers is an ancient approach of cryptanalysis [see 19]. Given this long history of research and the presence of several strong cryptographic methods, it would seem that attacks against weak ciphers are largely irrelevant today. Unfortunately, these weak ciphers regularly slip into implementations of software and thus probable-plaintext attacks based on classic techniques are still successful, as for instance in the cases of WordPerfect [2] and PKZIP [24].

To the best of our knowledge, Kandi is the first method that applies these classic attacks against obfuscation used in embedded malware. While some high-profile attack campaigns have already moved to stronger ciphers, such as RC4 or TEA, the convenience of simple cryptography and the risk of introducing detectable patterns with involved approaches continues to motivate attackers to use weak ciphers for obfuscation.

7 Conclusion

Malicious documents are a popular infection vector for targeted attacks. For this purpose, malware binaries are embedded in benign documents and executed by exploiting vulnerabilities in the program opening them. To limit the chances of being detected by anti-virus scanners, these embedded binaries are usually obfuscated. In practice this obfuscation is surprisingly often realized as simple Vigenère cipher. In this paper, we propose KANDI, a method that exploits well-known weaknesses of these ciphers and is capable of efficiently decrypting Vigenère-based obfuscation. Empirically, we can demonstrate the efficacy of this approach on real malware, where our method is able to uncover the code of every fourth malware in popular document and image formats.

While our approach targets only one of many possible obfuscation strategies, it helps to strengthen current defenses against embedded malware. Our method is fast enough to be applied on end hosts and thereby enables regular anti-virus scanners to directly inspect deobfuscated code and to better identify some types of embedded malware. Moreover, by statically exposing details of the obfuscation, such as the key and the operations used, our method can also be applied for the large-scale analysis of malicious documents and is complementary to time-consuming dynamic approaches.

Acknowledgments The authors would like to thank Emiliano Martinez and Stefano Zanero for support with the acquisition of malicious documents. The authors gratefully acknowledge funding from the German Federal Ministry of Education and Research (BMBF) under the project PROSEC (FKZ 01BY1145).

References

1. Bencsáth, B., G. Pék, L.B., Felegyhazi, M.: Duqu: Analysis, detection, and lessons learned. In: European Workshop on System Security (EUROSEC) (2012)
2. Bergen, H.A., Caelli, W.J.: File security in WordPerfect 5.0. Cryptologia 15(1), 57–66 (1991)
3. Boldewin, F.: OfficeMalScanner. `http://www.reconstructer.org/code.html`
4. Calvet, J., Fernandez, J.M., Marion, J.Y.: Aligot: Cryptographic function identification in obfuscated binary programs. In: ACM Conference on Computer and Communications Security (CCS). pp. 169–182 (2012)
5. Cova, M., Kruegel, C., Vigna, G.: Detection and analysis of drive-by-download attacks and malicious JavaScript code. In: International World Wide Web Conference (WWW). pp. 281–290 (2010)
6. CrySyS Malware Intelligence Team: Miniduke: Indicators. Budapest University of Technology and Economics (February 2013)
7. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: Malware analysis via hardware virtualization extensions. In: ACM Conference on Computer and Communications Security (CCS). pp. 51–62 (2008)
8. Engelberth, M., Willems, C., Holz, T.: MalOffice: Detecting malicious documents with combined static and dynamic analysis. In: Virus Bulletin Conference (2009)
9. Friedman, W.: The index of coincidence and its applications in cryptology. Tech. rep., Riverbank Laboratories, Department of Ciphers (1922)

10. Friedman, W., Callimahos, L.: Military Cryptanalytics. Aegean Park Press (1985)
11. Jacob, G., Comparetti, P.M., Neugschwandtner, M., Kruegel, C., Vigna, G.: A static, packer-agnostic filter to detect similar malware samples. In: Flegel, U., Markatos, E., Robertson, W. (eds.) Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA). LNCS, vol. 7591, pp. 102–122. Springer Berlin Heidelberg (2012)
12. Kasiski, F.W.: Die Geheimschriften und die Dechiffrir-Kunst. E. S. Mittler und Sohn (1863)
13. Laskov, P., Šrndić, N.: Static detection of malicious JavaScript-bearing PDF documents. In: Annual Computer Security Applications Conference (ACSAC). pp. 373–382 (2011)
14. Lewand, R.: Cryptological mathematics. Classroom Resource Materials, The Mathematical Association of America (2000)
15. Li, W.J., Stolfo, S., Stavrou, A., Androulaki, E., Keromytis, A.D.: A study of malcode-bearing documents. In: Hämmerli, B., Sommer, R. (eds.) Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA). LNCS, vol. 4579, pp. 231–250. Springer Berlin Heidelberg (2007)
16. Malware Tracker Ltd.: Cryptam. `http://www.cryptam.com` (visited June, 2013)
17. Martignoni, L., Christodeorescu, M., Jha, S.: OmniUnpack: Fast, generic, and safe unpacking of malware. In: Annual Computer Security Applications Conference (ACSAC). pp. 431–441 (2007)
18. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In: Annual Computer Security Applications Conference (ACSAC). pp. 289–300 (2006)
19. Schneier, B.: Applied Cryptography. John Wiley and Sons (1996)
20. Schreck, T., Berger, S., Göbel, J.: BISSAM: Automatic vulnerability identification of office documents. In: Flegel, U., Markatos, E., Robertson, W. (eds.) Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA). LNCS, vol. 7591, pp. 204–213. Springer Berlin Heidelberg (2012)
21. Shafiq, M.Z., Khayam, S., Farooq, M.: Embedded malware detection using markov n-grams. In: Zamboni, D. (ed.) Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA). LNCS, vol. 5137, pp. 88–107. Springer Berlin Heidelberg (2008)
22. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Automatic reverse engineering of malware emulators. In: IEEE Symposium on Security and Privacy. pp. 94–109 (2009)
23. Smutz, C., Stavrou, A.: Malicious PDF detection using metadata and structural features. In: Annual Computer Security Applications Conference (ACSAC). pp. 239–248 (2012)
24. Stay, M.: ZIP attacks with reduced known plaintext. In: Matsui, M. (ed.) Fast Software Encryption (FSE). pp. 125–134. Springer Berlin Heidelberg (2002)
25. Stevens, D.: Malicious PDF documents explained. IEEE Security & Privacy 9(1), 80–82 (2011)
26. Stevens, D.: XORSearch. `http://blog.didierstevens.com/programs/xorsearch/` (visited June, 2013)
27. Stolfo, S., Wang, K., Li, W.J.: Towards stealthy malware detection. In: Christodorescu, M., Jha, S., Maughan, D., Song, D., Wang, C. (eds.) Malware Detection, Advances in Information Security, vol. 27, pp. 231–249. Springer US (2007)
28. The Taidoor campaign: An in-depth analysis. Trend Micro Incorporated (2012)
29. Šrndić, N., Laskov, P.: Detection of malicious PDF files based on hierarchical document structure. In: Network and Distributed System Security Symposium (NDSS) (2013)