SAD THUG: Structural Anomaly Detection for Transmissions of High-value Information Using Graphics

Jonathan P. Chapman, ***Fraunhofer FKIE***

https://www.usenix.org/conference/usenixsecurity18/presentation/chapman

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-939133-04-5

Open access to the Proceedings of the 27th USENIX Security Symposium is sponsored by USENIX.

SAD THUG: Structural Anomaly Detection for Transmissions of High-value Information Using Graphics

Jonathan P. Chapman
Fraunhofer FKIE

Abstract

The use of hidden communication methods by malware families skyrocketed in the last two years. Ransomware like Locky, Cerber or CryLocker, but also banking trojans like Zberp or ZeusVM, use image files to hide their tracks. Additionally, malware employed for targeted attacks has been using similar techniques for many years. The DuQu and Hammertoss families, for instance, use the popular JPEG file format to clandestinely exchange messages. Using these techniques, they easily bypass systems designed to protect sensitive networks against them. In this paper, we show that these methods result in structural changes to the respective files. Thus, infections with these malware families can be detected by identifying image files with an unusual structure. We developed a structural anomaly detection approach that is based on this insight. In our evaluation, SAD THUG achieves a mean true positive ratio of 99.24% for JPEG files using 10 different embedding methods while maintaining a mean true negative ratio of 99.323%. For PNG files, the latter number drops slightly to 98.88% but the mean true positive ratio improves to 99.318%. We only rely on the fact that these methods change the structure of their cover file. Thus, as we show in this paper, our approach is not limited to detecting a particular set of malware information hiding methods but can detect virtually any method that changes the structure of a container file.

1 Introduction

Malware infections are, and remain, a constant threat to computer users worldwide. For the second quarter of 2016, Microsoft reports that 21.2% of the systems that are running their Windows operating and are configured to share encounters with the company encountered malware at least once, up from 14.8% in the year before.[1] Victims of malware may be private individuals, or small businesses that e.g. lose money or files due to infections with a banking trojan or ransomware. Or they may be large corporations, public institutions like the National Health Service in the United Kingdom, which was severely affected by the WannaCry ransomware, or even political entities such as the Democratic National Committee (DNC) in the United States, which was attacked by the group associated with the Hammertoss malware [23, 9].

Practically all malware uses the Internet to establish a command and control (C&C) channel with its authors. For instance, banking trojans upload credentials harvested from the infected machine. Similarly, malware used in targeted attacks exfiltrates passwords, documents, or other sensitive information or retrieves new commands from its operator. Network operators on the other hand seek to detect or prevent malware communications to protect their systems. Application level gateways are important tools to these ends. However, in a recent study Gugelmann et al. [27] were able to bypass all three tested systems simply by `base64` encoding data. With respect to attempts to establish a covert channel, which includes the methods discussed in this paper, they point out that no product even claims to be able to detect them.

Consequently, the use of steganography, the science of hiding even the fact that communicating is taking place, by malware has surged in the last two years [52, 38, 39, 20, 53]. More particularly, malware used in targeted attacks like DuQu [14], Hammertoss [23] or Tropic Trooper [8] has been hiding data in image files for many years. General purpose malware like the ZeusVM [59] and Zberp [2] banking trojans followed suit. However, most of the recent surge in the use of steganography may be attributed to exploit kits. These kits bundle attacks against common web browsers and are leased to other malware authors to help them distribute their software [25, 41].

Significant resources were invested in research for detecting steganography exploiting compressed image data

[48, 11, 12, 21]. However, most malware families, including those used in targeted attacks, sidestep these efforts by hiding their data not in the image data itself but in the container file that is used to deliver it. Until now, only Stegdetect [48] implements methods that can detect specific attacks of this kind. However, it is limited to JPEG files and can effectively only detect variations of one particular method. Also, when we employed Stegdetect to analyze a realistic data set, it caused a significant number of false positives, rendering it unfit for practical use.

In this paper we introduce SAD THUG, or Structual Anomaly Detection for Transmissions of High-value information Using Graphics, a machine-learning based anomaly detection approach to uncover malware that modifies the structure of image files. While technically our approach can be used with any structured file format, for this work we focused on the two image file formats which are most widely used on the Internet and also most frequently exploited by malware, JPEG and PNG. For both formats, SAD THUG achieves exceptional accuracy. We also show that it can detect both known and unknown methods, so long as they cause significant anomalies in the structure of the image files they use as a cover medium.

Our contributions to the state of the art are as follows:

- In contrast to previous work for detecting structural anomalies in JPEG files, our approach uses a learned model and achieves near perfect results for a wide range of information hiding methods.
- Our approach is not limited to a particular file format and is the only approach with the demonstrated capability of detecting structural anomalies in PNG files.
- SAD THUG achieves a very low false positive ratio for JPEG files and a low ratio for PNG files.
- Our findings are backed by an comprehensive evaluation using $270,000$ JPEG files and $33,000$ PNG files along with additional files used by live malware.

The remainder of this paper is organized as follows. First, we briefly define the usage scenario for our approach. Then in section 3, we describe the JPEG and PNG file formats, methods for structural information hiding, and how they are abused by a wide range of malware families. We then introduce a small set of previously unpublished structural embedding methods that complement the methods currently used by malware. With this background, we introduce our detection approach in section 5, and describe our evaluation and results in section 6. Before contrasting our approach with previous work in the field (section 8), we briefly describe its inherent limitations. Finally, we draw our conclusions and show avenues for future work in section 9.

2 Threat Model

Companies and organizations, in particular those that handle sensitive data, use network separation to contain the effect of malware infections and other attacks. On the other hand, fully disconnected, or air-gapped, networks are often not an option. In these cases, most organizations only allow communications to take place using email or HTTP through a proxy server. Here, the proxy server doubles as an application level gateway (ALG) that only allows communication to take place that adheres to the HTTP standard.

However, malware authors adapted to these precautions. Instead of attacking systems directly, they use email and HTTP to attack their victims. Spear fishing email is often and effectively used in targeted attacks [10, 55, 28, 17], and additionally, exploit kits [25, 41] or collections of attacks against web browsers and their plugins gained significant popularity as a tool for infecting end user systems. Finally, practically all malware families use the HTTP(S) protocol for their C&C communications, allowing them to simply use their victim's HTTP proxy servers.

Hence, organizations started adopting more advanced ALGs, often referred to as web application firewalls (WAFs). WAFs implement ancillary security features like payload signatures to prevent malicious communications through them. Additionally, many ALGs execute a man-in-the-middle attack against TLS/SSL connections to prevent unwanted communication from taking place under a simple layer of off-the-shelf cryptography. However, malware authors once again adjusted to the new situation by more elaborately hiding their communications. Since they still almost exclusively use the HTTP protocol, WAFs remain in the right place to detect or prevent their communications. Yet they are increasingly unable to do so. A study covering three commercial WAFs [27] showed that none of them was able to detect the exfiltration of sensitive data once that data was `base64` encoded. The authors also pointed out that they were not aware of any product that claims to be able to detect advanced techniques like establishing a covert channel using messages hidden in image files. Our work provides an important cornerstone for closing this gap.

Figure 1 depicts the simplified structure of a partially segmented network. On the left side of the figure, client systems reside in a protected network – including a compromised system, as indicated by a warning sign. The systems in this network have no direct access to untrusted networks but they may communicate with an email and

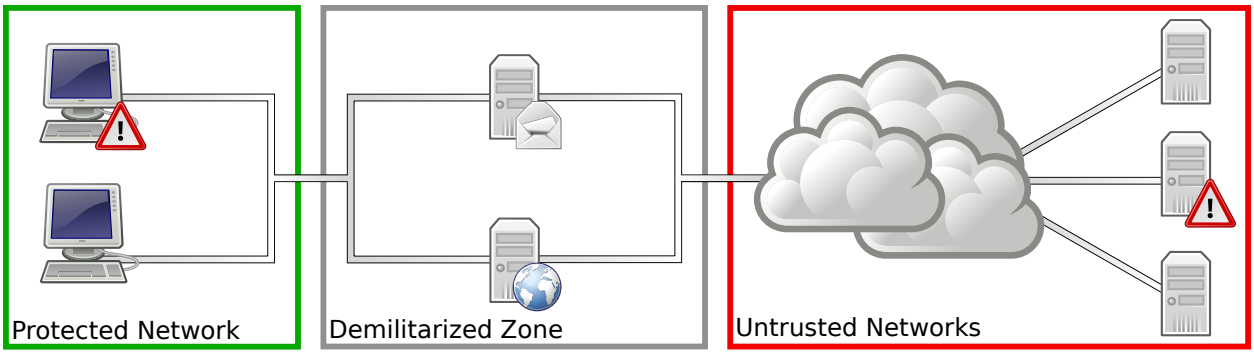


Figure 1: Schematics of a partially segmented network. Icons: Tango Project.

HTTP proxy server residing in the organization's demilitarized zone. These in turn have access to other, untrusted networks, e.g. the Internet. To communicate with another system under the attacker's control in those networks, again indicated by a warning sign, the attacker has to exploit the servers in the demilitarized zone.

For the purposes of this paper, we define an attacker as an entity that has control over two systems. One system resides in a segmented network. The attacker wants to establish a communication channel between this system and another system outside that network that allows it to transfer significant amounts of data between them. However, all communications have to traverse an uncompromised ALG. The ALG, on the other hand, has to distinguish between benign and malicious data exchanged between systems inside and outside a given network with no advance knowledge on which particular systems or data may be malicious or not.

We are aware of 40 malware families, including four proof-of-concepts, that use various techniques to hide their C&C communications. 34 families exploit images for this purpose. There are two facts supporting this choice, in particular with regard to WWW traffic. First, compressed images primarily consist of high entropy data that is difficult to distinguish from encrypted data. Second, viewing a single web page usually requires downloading dozens, sometimes well above one hundred, image files. Hence, attackers can hide their communications among a large volume of benign data transfers.

The malware families exploiting images can further be subdivided into two evenly sized groups. The first half hides their messages in the image data – the detection of which has been covered by an extensive number of research papers. The second half however exploits the structure of the corresponding file – an approach that has received little to no attention so far despite being used by high profile malware like DuQu [14] or Hammertoss [23]. Therefore, our work focuses on the detection of methods falling into the second category.

3 Background

In this section, we first briefly introduce the file formats most widely exploited by malware for hiding their communications, JPEG and PNG. We then summarize the fundamental structural embedding methods before pointing out how different malware families implement these approaches in practice.

3.1 JPEG File Structure

The JPEG File Interchange Format (JFIF) [31] and Exchangeable Image File Format (Exif) [19] are both containers for JPEG compressed image data. Unless we specifically need to explain a detail with respect to one of these formats, we will simply refer to a "JPEG file", assuming that the data is stored in either one of them. For simplicity, and like most decoders for JPEG files, for the remainder of this paper we do not distinguish between the segments of the container format and those that syntactically belong to the JPEG compressed data except that we introduce them in separate sections below.

3.1.1 JPEG Container Formats

Both JFIF and Exif files borrow from the JPEG data format they are designed to contain. They consist of a series of segments, each starting with a two byte "marker" code. The code indicates the type of a given segment and is sometimes followed by a two byte length field. Both files begin with a "start of image" (SOI) marker, and an "end of image" marker indicates the end of the image data.

There are 16 codes that indicate an application-specific or "APPn" segment follows where n is a number between 0 and 15. These segments start with a zero-terminated ASCII string to identify the nature of their

content. Somewhat contradictory to the marker's designation, the JFIF standard requires that the SOI marker is followed by an "APP0" marker with identifier "JFIF" that contains mandatory meta data. Similarly, Exif files start with an "APP1: Exif" segment that also contains meta data on the image. In contrast to the JFIF standard, Exif does discuss the possibility of encountering additional data behind the end of image marker, and recommends that such data should be ignored.

3.1.2 JPEG Data Format

The JPEG compression algorithms's [30] core depends on the block-wise transformation of an input image's color channels into frequency components. It achieves its lossy data reduction by dividing the respective coefficients using a quantization table, allowing users or their applications to choose a trade off between the quality and file size achieved. The resulting data is stored in segments, each of which starts with a two byte marker indicating the segment's type. Most but not all of these segments also include a two byte length field, limiting their size to $65,535$ bytes. Furthermore, most segments contain or consist of a header indicating how the following data should be interpreted. While some obvious restrictions exist, e.g. quantization tables must occur before the encoded image data that refers to them, the JPEG standard is generally permissive with respect to the order of segments.

3.2 PNG File Structure

The Portable Network Graphics (PNG) standard was written partly due to the realization that the earlier Graphics Interchange Format (GIF) standard relied on a patented compression algorithm. It provides lossless compression for bitmap images with a 24 bit color space and optional alpha channel. PNG files start with a fixed header followed by a variable number of segments and end with an "IEND" segment. Each segment starts with a four byte payload length field followed by four ASCII letters indicating its type, the optional payload and finally a checksum. The case of each letter in the type identifier indicates some properties of the segment, e.g. an upper case first letter indicates that the segment is "critical" and the decoder must be able to interpret it. Technically, the standard only mandates that the file header is followed by an "IHDR", which has a fixed structure and indicates the dimensions and other basic properties of the image, and the closing "IEND" segment.

3.3 Structural Embedding Methods

In this section, we briefly describe the basic methods for hiding data exploiting the structure of container file formats. As we will see below, the methods actively used by current malware are variations of these approaches. Figure 2 shows a generic structured container file format without hidden data as well as with data embedded using the three methods described below.

Append This approach simply appends the steganographic payload at the end of the cover file. Thus, the structure of the cover file remains intact but it is followed by additional data.

Byte Stuffing File containers often allow the length of a segment to be specified even if it is already implied by the segment's type or header. While the resulting files are not strictly standard-compliant, most parsers only read the expected data from the segment and ignore the additional bytes that follow. Therefore, attackers may expect that their file is accepted as legitimate by most decoders.

Segment Injection Finally, container file formats like JPEG and PNG permit the addition of segments that are not used in the decoding process. For instance, comment segments allow storing data for informational purposes, e.g. to indicate which program was used to modify the file, but have no influence on the decoded data. Hence, attackers can add such segments without risk of losing compatibility and with little risk of discovery.

3.4 Structural Embedding Methods Used by Malware

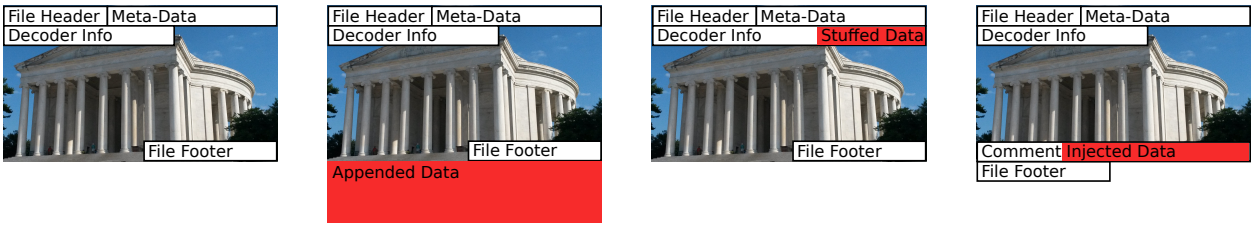
In this section, we briefly introduce the structural embedding methods used by eleven live malware families, grouped by the file format they exploit. For reference, we included their basic properties on the left half of table 1 in section 6.

3.4.1 JPEG-based Methods

Cerber The Cerber malware [5] transfers a malware binary by *append*ing it to a JPEG file. Before appending the file, it is encrypted by simply `XOR`ing the binary with a single constant byte.

DuQu, DuQu 2.0 The DuQu malware [15, 14] executable contains a simple JPEG file. To exfiltrate screenshots and process lists gathered from the infected system, it bzips and encrypts the data using the AES cipher. The encrypted data is then *append*ed to the JPEG file and sent to the C&C server.

Hammertoss [23] uses the *append* method to deliver configurations and commands to infected systems. Here, the attackers use a JPEG file of their liking and then append the RC4-encrypted message to the end of that file.



(a) Without embedding (b) Append (c) Byte Stuffing (d) Segment Injection

Figure 2: Examples for a structured container file without and with data embedded using different methods.

Microcin The Microcin malware [16] retrieves JPEG files that contain additional executable modules. While it uses the *append* paradigm, it first adds the sequence `ABCD` followed by a small header and finally the encrypted payload.

SyncCrypt Once the SyncCrypt ransomware's [6] initial infection is successful, it downloads a JPEG file. From the JPEG file, it extracts a ZIP file that, along with an HTML and PNG image file, contains the malware's executable. The file is hidden simply by *append*ing it to a given cover file.

Tropic Trooper The Tropic Tropper malware [8] uses the *append* approach in conjunction with a JPEG file to deliver a malware binary to an infected system.

Zberp The Zberp malware is a hybrid built from the Carberp and ZeusVM banking trojans [2]. It uses ZeusVM's method described to transfer configurations, which we describe below.

ZeusVM The ZeusVM banking trojan [51] uses a variation of the *segment injection* approach to hide the configuration and web-inject data provided to the infected systems. It injects a comment segment into a JPEG file but sets the length header field for that segment to 16,144 regardless of the length of the actual payload.

3.4.2 PNG-based Methods

Brazilian EK An unnamed Brazilian exploit kit [37] uses a simple yet effective method to deliver its payload to the infected users. It *append*s an `XOR`-encrypted malware executable to an otherwise inconspicuous PNG file.

CryLocker The CryLocker ransomware [34] uses a variation of the *byte stuffing* method. It creates a file that consists of a PNG file header and the mandatory `IHDR` segment only. However, it injects information on the compromised system into the `IHDR` segment. While the resulting file is not compliant with the PNG standard, CryLocker successfully used the `imgur.com` picture sharing platform for sending information to its creators.

DNSChanger The DNSChanger exploit kit [3] hides additional modules used to attack home routers in a comment segment *injected* into a PNG cover file.

3.4.3 Discussion

While most malware uses variations of the *append* paradigm, we have seen a diverse set of approaches for structurally hiding data in image files. In comparison to image data-based approaches, these methods can be implemented straightforwardly. However, there is a more important while less obvious property of these methods that makes them even more attractive. Image data-based methods can only embed a limited number of bits before their manipulation becomes obvious and even when that is acceptable, the total size of the image poses an insurmountable limit for them. Structural embeddings on the other hand generally not only do not affect the rendered image but also allow the transfer of messages of arbitrary sizes. Even where some limits apply, e.g. the maximum segment size when injecting a segment like DNSChanger, this can easily be overcome by distributing the message over several segments. Thus, in principle, structural embedding methods could be used to exfiltrate terabytes of data in a single file transfer.

4 Proposed Embedding Methods

In this section, we propose a small set of new embedding methods that exploit the file structure of JPEG or PNG files. We used the `identify` command from the ImageMagick [29] suite to establish the fact that only the *pHYs Byte Stuffing* method triggers a warning during the decoding. Using a regular image viewer, we also verified that none of these methods caused any visual changes to the encoded images.

4.1 JPEG-based Methods

APP0 Byte Stuffing For this method, we exploit the fact that the structure of the mandatory `APP0: JFIF` segment in `JFIF` files is well-defined. Since the segment's length is nevertheless indicated by a length field, we can

simply append data after the original payload of the segment and then adjust the length field accordingly.

APP1: Comment Injection APP markers are designed to be used for application specific data. Hence, they start with a null-terminated ASCII string that indicates the nature of the data in the segment and parsers are supposed to ignore data they do not understand. Here, we simply chose the APP1 marker with identifier `Comment` because it should cause the least suspicion.

4.2 PNG-based Methods

pHYs Byte Stuffing The PNG standard contains a number of optional segments that usually have no effect on the decoded image. From these segments, we arbitrarily selected the `pHYs` segment, which indicates the physical scale of the image. Since it has a fixed structure, we can apply the *byte stuffing* paradigm and simply add additional data to an existing `pHYs` segment or *inject* a stuffed segment when the cover file does not contain a `pHYs` segment yet.

aaAa Injection The PNG standard uses a four ASCII letter code to determine the type of a segment and several other of its properties. A code starting with two lower case letters is designated as ancillary, non-publicly registered. The third letter is supposed to always be upper case and by using a lower case forth letter, we indicate that the segment may be copied by a decoder that does not recognize it. Besides these restrictions, we should only make sure that our new segment type is not used by any widely used application. For simplicity, we simply chose `aaAa`, which satisfies all of these criteria.

5 The SAD THUG Approach

Our approach consists of two main phases, a training phase for building a formal model and a classification phase to check whether files correspond to that model. Since this model is based on empirical data, it represents how a given standard is implemented rather than how it is specified.

To build our model or to classify files against it, we first decompose each given file into a sequence of symbols describing the file's segments. This process is sketched in section 5.1. We then describe how we model the knowledge obtained during the training phase, which is described in section 5.3. Finally, we describe how we use the trained model to determine whether a given file is anomalous with respect to our training data set or not.

5.1 File Decomposition

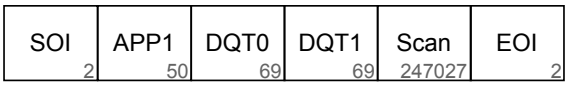
For both training and detection, we first decompose each given file into a sequence $s = s_0, ..., s_{n-1}$ of segments. Generally, such a sequence can be obtained trivially and at negligible cost by sequentially parsing the file. Given a file type T, S_T refers to the set of all segments for that type. Correspondingly, the alphabet Σ_T includes all segment types that occur in files of that type. We use $\ell(s_i)$ to refer to the length of segment s_i.

While the length of a segment is clearly defined, i.e. the count of bytes in the file until either the next segment or the end of the file is encountered, there is some ambiguity with respect to the type of a segment. Most segments start with a header or byte sequence that indicates their type. Often, their payload starts with another header that is needed to correctly interpret the segment's payload. Although the segment type is defined by the outer header, the inner header may have significant impact on how the segment is interpreted. Thus, we suggest identifying subtypes based on these inner headers where appropriate. These subtypes will be treated as fully separate types in all respects.

For instance, in section 3.1, we introduced the JPEG file format's *APP* segments. They use the same segment type indicator but are supposed to start with a string indicating the software using the given segment, i.e. the purpose of a segment or even whether it should be ignored completely by most decoders can only be determined by interpreting this inner header. Hence, segments with different inner headers are written and read for different purposes and should thus be assigned different subtypes.

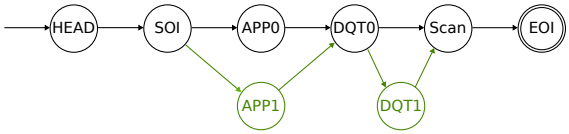
Our prototype parses PNG or JPEG files. For both file types, the length of a segment corresponds to the length of the encoded segment in a given file, as explained above. When data is encountered following a valid segment that cannot be decoded, it is stored as a *residual data* segment encompassing all bytes up to the end of the file. To determine the type of a segment in a PNG file, our parser simply uses the segment names described in section 3.2. However, when parsing JPEG files, it introduces subtypes for various segment types as illustrated for *APP* segments above.

Figure 3 a) shows a simplified decomposed Exif file. In the figure, each segment corresponds to a box where a smaller grey number on the bottom right of each box indicates the length of the respective segment in the parsed file. It starts with a start-of-image segment on the left, followed by an APP1 marker and two quantization tables. They are followed by a large scan segment, which contains the encoded image data. The file ends with an end-of-image marker, as indicated on the right hand side of fig. 3 a).



(a) A simplified, decomposed Exif file

(b) Automaton after learning a simplified JFIF file



(c) The automaton from b) after learning a simplified Exif file

Figure 3: Simplified data structures used by SAD THUG.

5.2 Model

For each given file, we want to determine whether its structure is reasonably close to the structure of benign files observed during a training phase. We characterize the entirety of benign file structures as a formal language. Thus, each decomposed image corresponds to a sequence of symbols and our core problem is to determine whether a given sequence is a word in that language. To achieve this, during the training phase we build a discrete finite automaton that approximates this language based on the training samples. In the classification phase, we check whether a given decomposed file corresponds to a word in the language described by that automaton.

More formally, for each file type T we build a directed graph $G_T = (V, v_0, \Sigma_T, E, F, \gamma)$ with a set of vertices V, a designated vertex v_0, corresponding to the head of a file, the alphabet of segment types Σ_T for the given file type, directed edges E between elements of V and a set of vertices F corresponding to the last segments in the training files. Additionally, γ maps an edge to its annotations.

In our automaton, an edge $v \to v' \in E$ indicates that in the training data set, two segments corresponding to v and v', were observed at least once in that order. When the segment corresponding to v' has a fixed length, we use the annotations to store how often this transition was observed during training. For variable length segments however, we store all observed lengths. This allows us to derive a profile for the lengths expected in the context defined by the given edge. In the classification phase, we use these annotations to enforce additional constraints on the inspected files.

5.3 Training Phase

To train our classifier, we build the model described in section 5.2 that reflects the segments observed in the training set, including their observed order and length. Figure 4 shows the algorithm for building the respective automaton. It starts with a set of decomposed training files A and initializes an empty automaton $G_T = (V, v_0, \Sigma_T, E, F, \gamma)$. Processing each image individually, as described by method `train_with_file`, the automaton is constructed and, once all files have been processed, returned. The automaton can then be used in the classification phase to classify previously unobserved files.

Require: A {Set of decomposed training files}
  $\sigma : S_T \to V'$ {Returns vertex corresponding to given segment's type}

  $V \leftarrow \{v_0\}$ {Vertices}
  $E \leftarrow \{\}$ {Edges}
  $F \leftarrow \emptyset$ {Final states}
  $\gamma : E \to \mathbb{N}^*$ {Annotations}
  for s **in** A **do**
    call train_with_file (s)
  return $(V, v_0, \Sigma, E, F, \gamma)$ {Trained automaton with annotations}

  method train_with_file (s)
    $v \leftarrow v_0$ {Start with the HEAD state}
    for s_i **in** $s = s_0, ..., s_{n-1}$ **do**
      $v' \leftarrow \sigma(s_i)$
      $V \leftarrow V \cup \{v'\}$ {Add vertex, if missing}
      $E \leftarrow E \cup \{(v, s_i) \to v'\}$ {Add transition}
      if $\ell(s_i)$ is fixed **then**
        $\gamma(v, v') \leftarrow \gamma(v, v') + 1$
      else
        $\gamma(v, v') \leftarrow \gamma(v, v') \frown \ell(s_i)$
      $v \leftarrow v'$
    $F = F \cup \{v\}$ {Add current state to final states}

Figure 4: Training algorithm

In each iteration of `train_with_file`, we start with the predecessor variable v pointing to a virtual `HEAD` state that represents the beginning of the file. For each observed segment, we determine the corresponding vertex v' and add it to the set of vertices V in the automaton if necessary. Also, we ensure that the automaton contains an edge $e \in E$ from v's predecessor v to that vertex. Finally, we update the annotations $\gamma(e)$ for that edge. If the segment's length is fixed, we increment the annotation for that edge by one, assuming the annotations were initialized to 0. For variable length segments, the annotations were initialized to an empty tuple and we append the observed length to the edge's annotation. Finally, we set v' to be the next predecessor v and process the next

```
Require: (V, v_0, Σ_T, E, F, γ) {Trained automaton with annotations}
Require: α {Length sensitivity parameter}
Require: τ {Confirmation threshold}
Require: s {Decomposed image}
  v ← v_0
  for s_i in s = s_0, ..., s_{n-1} do
      v' ← σ(s_i)
      if not is_acceptable_transition(v, s_i, v') then
          return anomaly
      v ← v'
  if not v ∈ F then
      return anomaly
  else
      return normal

  method is_acceptable_transition (v, s_i, v')
      if not (v → v') ∈ E then
          return false
      if ℓ(s_i) is fixed then
          if not γ(v, v') ≥ τ then
              return false
      else
          C = {x || x ∈ γ(v, v') ∧ (|x − ℓ(s_i)| ≤ ⌈ℓ(s_i) · α⌉}
          if not |C| ≥ τ then
              return false
      return true
```

Figure 5: Classification algorithm

segment.

After all segments are processed, v contains the vertex corresponding to the last processed segment. Hence, we add this vertex to the set of legitimate final states F. When this procedure has been completed for all individual files, we return the resulting automaton.

Figure 3 b) shows an automaton after training on a simplified JFIF file while fig. 3 c) shows the same automaton after learning the simplified Exif file depicted in fig. 3 a). Here, the added vertices and edges are highlighted in green and we omit lengths in b) and c). Both files start with a fixed length SOI marker, so in the first step, the annotation for the edge from the HEAD state to the respective vertex is incremented. However, in the Exif file, it is followed by an APP1 rather than an APP0 marker and the corresponding vertex and an edge to it are added. The automaton already contains a vertex corresponding to the DQT0 segment following in the file and hence we only need to add another edge to process it. That segment however is followed by a previously unobserved DQT1 segment. Thus, again a new vertex and an edge from the DQT0 to the new DQT1 vertex are added. From there, an edge to the existing Scan vertex is added, reflecting the sequence of segments in the Exif file. Since – like in the JFIF file the automaton was trained with – the last segment is an EOI segment behind the Scan segment, the respective final transition only updates the automaton's annotations.

5.4 Classification Phase

Once the finite-state automaton has been built using the procedure described above, we enter the classification phase. Here, we treat each file as a sequence of symbols that are either accepted or rejected as words in a language of legitimate files of that type. This process can be tuned by adjusting the two parameters τ and α. τ is the number of times a transition has to have been observed during training before we accept that transition in the classification phase. Obviously, with τ set to 1, we accept any transition ever observed during the training phase. As we increase τ, our classifier becomes more restrictive but also more robust against coincidental anomalies in the training data or deliberate attempts to manipulate it during the training phase.

For transitions to a variable length segment s_i, we only consider those observations that are within a reasonable range from that segment's length, determined by our parameter α. More specifically, we calculate the range by taking the ceiling of multiplying α with the given segment's length: $\lceil \alpha \cdot \ell(s_i) \rceil$. Figure 6 illustrates this concept. It shows the absolute frequency of sizes for the JPEG DC0 huffman table that lie between 0 and 100. A green line indicates an observation of length 33. With α set to 0.1, this corresponds to a range of 4, i.e. the area highlighted in green in fig. 6. Our training data contains many observations within this range, so we accept the observation as legitimate. As another example, take the red line at 70 in the figure. Its larger absolute value results in a significantly larger range as well. However, as the area highlighted in red shows, there are few observations in this range, so – depending on the configuration – our approach will classify this observation as an anomaly.

Figure 5 shows the full classification algorithm. It requires a trained automaton, the two parameters τ and α and finally a decomposed image as inputs. The result returned is the classification for the given file which may be "anomaly", if the file is considered to be malicious, or "normal" otherwise.

Like in the training algorithm, decomposed files are processed segment by segment. As sketched above, the main task is to identify whether individual transitions occurred in the training phase – taking into account our parameters τ and α. This is handled by method `is_acceptable_transition` in fig. 5. Here, we first check whether a transition exists from the previous vertex to a vertex that represents the current segment. If that segment has not been observed during the training phase or not been observed to follow the previous segment, the check fails and we consider the file to be anomalous. Otherwise, we verify whether the transition's annotations satisfy the constraints imposed by our parameters τ and

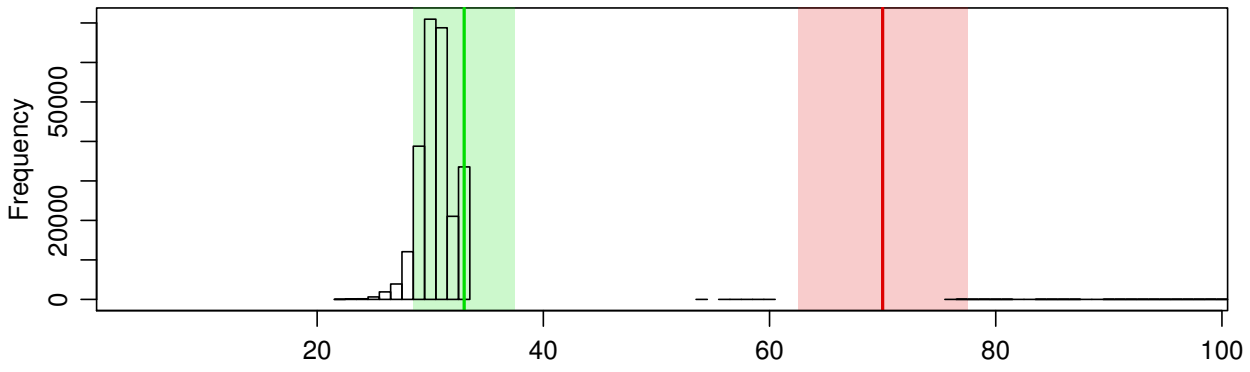


Figure 6: The length of JPEG huffman table segments in bytes (excerpt).

α. If the segment's size is fixed, we check whether the stored observation count reaches or exceeds the desired threshold τ. For variable length segments, we first determine the observed lengths that were within $\lceil \alpha \cdot \ell(s_i) \rceil$ bytes from s_i's length $\ell(s_i)$. We then check whether they exceed our threshold τ and reject the image, if that is not the case. Given that all observed transitions were successfully validated, we only need to check whether the vertex corresponding to the last segment is also a final state in our trained automaton. If and only if that is the case, we accept the image as normal with respect to our training set.

6 Evaluation

We evaluated our approach using a large body of JPEG and PNG files with embeddings from ten different malware families. The respective data sets are derived from a total of 270,000 JPEG and 33,000 PNG files downloaded from popular websites. The same data set is also used for training and to determine our approach's false positive ratio. Given the design of our experiment and the size of our data sets, we believe that the results presented here closely resemble those achieved in a real environment.

In this section, we thus first describe the general design of our experiment, before discussing the details of our data sets. Section 6.3 describes how we obtained a meaningful configuration for our approach. We then provide an overview to Stegdetect, which we use as a benchmark. Finally, in section 6.5, we describe the results of our evaluation for both approaches.

6.1 Experiment Design

We conduct a ten-fold cross-validation on a large data set of 270,000 JPEG and 33,000 PNG files, downloaded from the Internet as described in section 6.2.1, to verify the accuracy and effectiveness of our approach. In each iteration, we use nine tenths of each data set as training data. The remaining data is further subdivided to construct realistic data sets using a diverse set of embedding methods. We use them – along with additional data sets – as test sets for our evaluation.

As a consistent measure for the quality of the detection, we use the *true classification ratio*. This metric can be applied on both files without and with a steganographic payload. For the former files, it corresponds to the fraction of the files that were classified as benign. Files that contain an embedding, on the other hand, must be classified as malicious to contribute to the respective true classification ratio. Thus, a value of 1 indicates a perfect result for the given data set while a value of 0 shows that the approach is not at all able to correctly classify items in the respective subgroup.

6.2 Data Sets

6.2.1 Base Data Set

We obtained a large data set closely resembling a set of images retrieved by average users browsing the Internet. To do so, we determined the top 25 websites according to Alexa [7] but after replacing semantic duplicates with a single domain name. For instance, `google.co.in`, `google.co.jp` and `google.com` all redirect to the same website, based on your assumed locality and were thus were replaced by a single instance of `google.com` in our list. We then recursively crawled this pruned list but stopped the recursion once a non-image resource was retrieved from a third-party domain. Many professionally operated websites serve static resources under a different domain name and thus without this exemption many images that were part of a website would not be loaded by our simulated Internet user. Through this process, we obtained a total of 271,968 JPEG and 33,651 PNG files. We removed randomly selected files from these sets

to trim them to 270,000 JPEG and 33,000 PNG files. This facilitates creating evenly-sized groups from them, as discussed below. Note that our unbiased crawling returned more than 8 times as many JPEG than PNG files, reflecting the popularity of the two file formats.

Since we obtained these files from third parties, we cannot completely rule out the possibility that they do in fact contain hidden messages. However, the sites we crawled are professionally run by respectable operators, so we assume that they do not deliberately provide malicious image files. On the other hand, the sites we crawled may allow users to upload content or reference user-uploaded content on third party websites and some users may decide to abuse their functionality to upload files with steganographic content. Since we are crawling popular websites with a large user base only, it is safe to assume that only a diminishing fraction of users – if any – engage in such activities. In turn, if our base data set does contain images with steganographic content, their quantity will be negligible. Weighing this against the inevitable lack of diversity in a self-assembled data set and consequentially the remoteness of such a data set from a live deployment, we opted for the approach described above.

Further analysis of the data set nevertheless revealed some interesting details. For instance, 15,005 files or 5.56%, of the JPEG files and 777 or 2.35% of the PNG files contain data behind their EOI or IEND segment. 4,484 JPEG files have 3 or less residual bytes behind their EOI marker, i.e. they are unlikely to carry any hidden message. In the PNG partition, only 56 files fall into that category. For both formats, the lion's share of the remaining files with four or more appended bytes is made up by `twitter.com`. It accounts for 9,527 of the 10,521 JPEG and 475 of the 721 respective PNG files. In a manually inspected sample, these files contained the space character ($0x20$) appended up to 455,942 times. The only reasonable explanation for this phenomenon is a programming error. `qq.com` accounts for most of the remaining files, i.e. 887 JPEG and 126 PNG files. Here, the files contain 46 additional bytes each, primarily a 32 letter hexadecimal ASCII string. Since this corresponds to the length of an MD5 hash, we assume that the data serves as a kind of watermark.

We acknowledge that these observations may be considered anomalies and that the respective files could be removed from the data set on that grounds. However, we left them in the data set for two reasons. First, with respect to SAD THUG, the presence of these files may decrease but not increase its detection performance, i.e. we avoid a potential unfair advantage for SAD THUG in our evaluation. Second, the files are part of an unbiased snapshot of files provided on the Internet. Removing them would conceal a challenge that a detection method would face in practice.

In our evaluation, we use the base data set for two purposes. First, it serves as a training set for our approach. Second, we use it to create sets of files that contain messages embedded with one of a total of 12 methods (for reasons explained below, this figure does not include CryLocker's and DuQu's methods).

6.2.2 Payload Data Sets

Malpedia is a curated collection of live malware samples and analysis [46]. After removing signatures, notes and script-based samples from the collection, we obtained a data set that contains a total of 4,558 malicious files.

ZeusVM Configuration The ZeusVM malware uses JPEG files to transfer two pieces of configuration to infected machines. The first part consists primarily of a list of URLs that are used for command and control. We discuss the other part, web-injects, below. To create this data set, we extracted and parsed the content from 24 live configurations for ZeusVM. From these configurations, we determined the smallest and largest number of values as well as all unique values for each option. To generate new configurations that closely resemble the original ones, we chose a random count between the minimum and maximum number of values observed for each given option and then added random values from the pool of observed values for that option.

ZeusVM Web-Inject ZeusVM's configuration contains templates that determine which and how websites visited by an infected machine should be modified. To generate the respective data set, we relied on the configurations parsed as described in the previous paragraph. Likewise, we determined how many web-injects were provided in the live configurations and chose a random sample from the given web-injects with a size ranging between those numbers.

Web Exploits target web browsers using malicious JavaScript, HTML or other code. To simulate an attack that hides this kind of data, we randomly selected files from a collection of 2,543 malicious JavaScript files [43].

CryLocker Payload The CryLocker ransomware [34] exploits the `imgur.com` website to upload information about each infected system. From the scarce information available on that payload, we inferred the format and chose reasonable values for its variables.

DuQu Payload The DuQu malware uses steganography to exfiltrate data from infected machines. According to Symantec [1], its logger creates a screenshot and process list every 30 seconds which will eventually be uploaded to a C&C server. To create a realistic data set, we set up a Windows virtual machine to automatically create

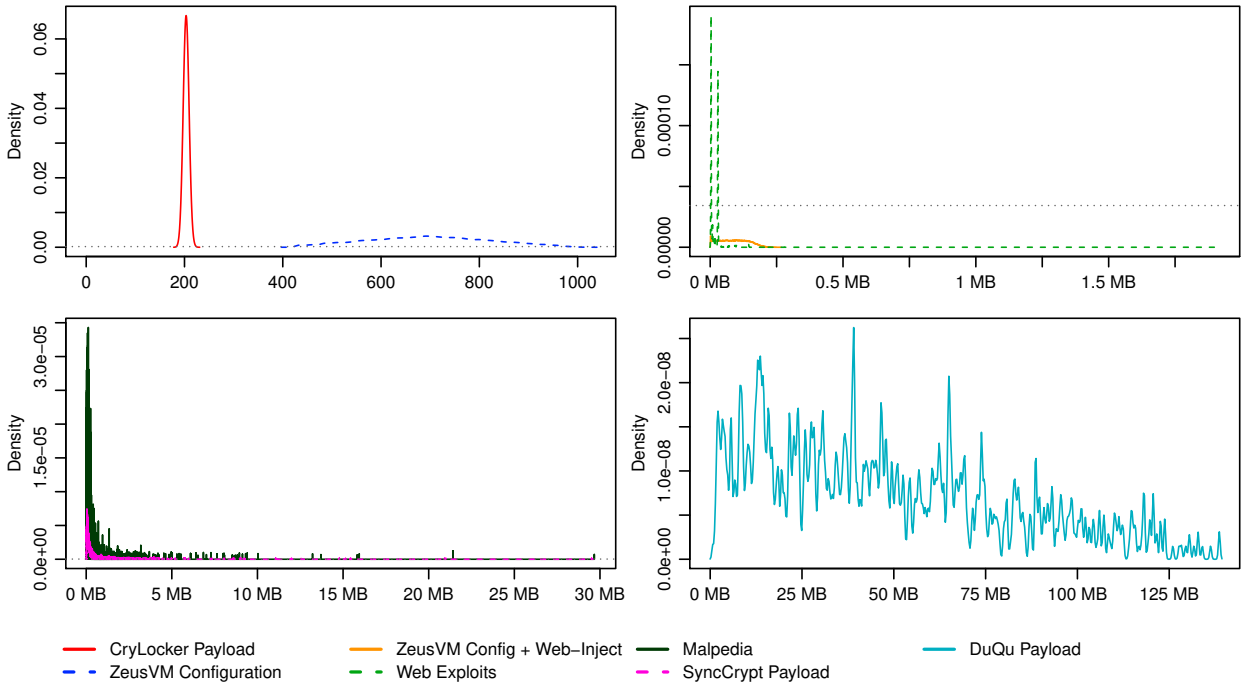


Figure 7: The size distributions of the payloads used in our evaluation (grouped by maximum size in bytes).

a screenshot and store a process list every 30 seconds. By running a series of office tutorials from YouTube in fullscreen mode on that machine, we ensured that the screenshots are similar to those of a system being used for regular office tasks. We then selected random intervals with a duration of at least 30 minutes, and concatenated the screenshots and process lists generated in a randomly selected time frame with that length. To create the final DuQu payload, we compressed the data using the `bzip2` algorithm and encrypted it using the AES cipher.

SyncCrypt Payload The SyncCrypt ransomware uses JPEG files to transfer a hidden ZIP file. In that ZIP file however, it hides the malware's main executable along with an HTML and a PNG file. Hence, to simulate that payload, we randomly chose a PNG file from the respective base data set, an HTML file from the *Web Exploits* set and a random malware binary from *Malpedia* and stored them in a ZIP file.

Discussion In this section, we briefly introduced the payload data sets used. Figure 7 shows the distribution of the size in bytes of the messages drawn from these data sets for our evaluation. Since the data sets cover a large variety, we grouped them by the size of the largest message in them, starting with the smallest data sets in the top left and ending with the largest on the bottom right. To provide a point of reference, a dotted horizontal line indicates the maximum density in the next plane. The plane on the top left shows the CryLocker and ZeusVM Configuration data sets, which only contain messages up to about one kilobyte and are clearly concentrated on about 200 or 700 bytes. The complete ZeusVM payload, including the malware's configuration and web-injects, evenly spreads from close to zero to 250 KB. Most of the files from the Web Exploits data set are very small, however the largest message drawn from this set almost reaches 2 MB, as we can see in the same plane. The SyncCrypt Payload consists, among others of a random malware sample drawn from the Malpedia data set and thus closely resembles the latter data set's size distribution, as seen in the bottom left plane. Finally, the DuQu Payload data set's size distribution ranges from just above 700 KB to 141.92 MB.

6.2.3 Additional Considerations for Data Sets

ZeusVM/Zberp The Zberp banking trojan is based on the ZeusVM malware and inherited its embedding method. Hence, for our evaluation we do not distinguish between the two. However, we use two different data sets to establish our approach's efficacy with respect to their method. First, we obtained a set of 24 JPEG files containing live configurations which were extracted from dumps of ZeusVM control panels. We denote this data set as *ZeusVM*. Second, we used the leaked KINS builder[2] for the ZeusVM malware to embed configurations from our

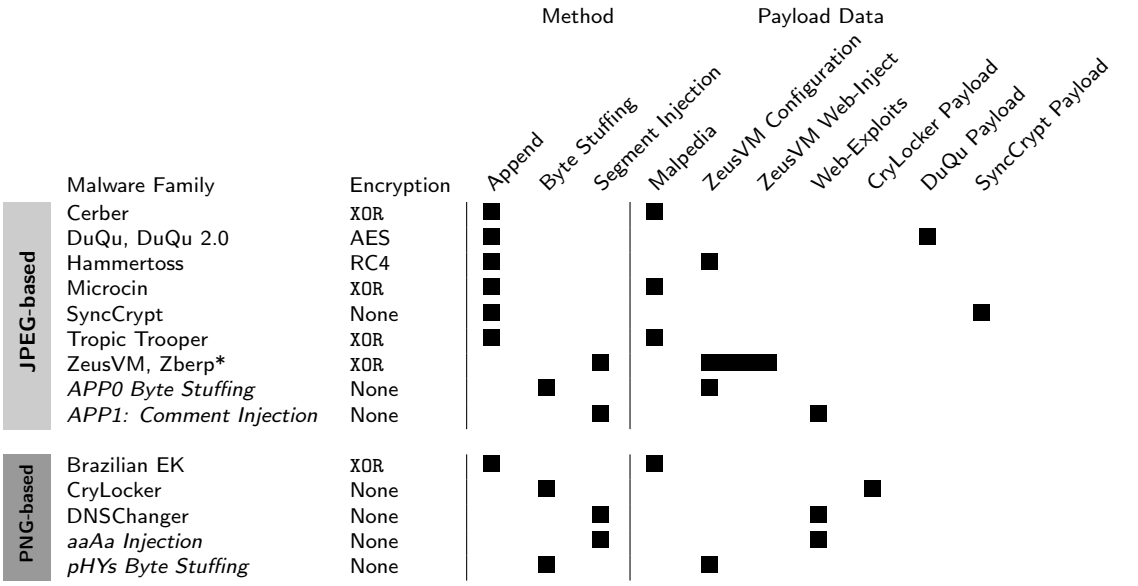


Table 1: Evaluation data sets; names in italics correspond to the methods proposed in section 4. The payload for the ZeusVM, Zberp* data set is constructed by combining two payload data sets.

payload data set into randomly selected JPEG files from the base data set. Since the builder would fail for JPEG files that did not end with an EOI marker, we excluded those files from the selection process. The respective data set is called *ZeusVM, Zberp** below.

DuQu The DuQu malware uses a static JPEG file stored inside its executable to exfiltrate data. Since this file does not depend on the input, we created a data set independent from our base data set. Using our DuQu payload data set and the JPEG file used by DuQu, the 1000 files in that set provide a very realistic approximation of DuQu's C&C traffic.

CryLocker The method used by the CryLocker ransomware effectively creates a PNG file header without any image data. Thus, it does not depend on any input and – like DuQu – we created and use an independent data set of 1000 files for our evaluation.

6.2.4 Grouping

To perform our evaluation, we partitioned the files in our base data sets into ten evenly-sized groups. We then further subdivided each JPEG group into nine subgroups while we divided the PNG groups into five subgroups each. As explained in section 6.1, for each step in our cross-validation, we used nine of the ten groups as training data. The subgroups in the remaining group serve as a test set for our classifier. Here, the files in one subgroup would remain unchanged, i.e. without any malicious embedding, to allow us to establish the false positive ratio. In the remaining subgroups, we embedded messages in accordance with table 1 and section 6.2.3. Note that the CryLocker, DuQu, and ZeusVM data sets do not depend on our base data set and are thus not included in these numbers.

6.3 Parameterization

In section 5.4 we introduced two parameters, α and τ that allow tuning the precision and recall of our approach. To determine a reasonable configuration, we executed a systematic grid-based parameter evaluation using ten values for each parameter and chose the parameter set that maximized our approach's weighted mean true classification ratio. We doubled the weight for the data set without any embedding to introduce a slight preference for a lower false positive ratio.

For τ, we can choose any positive integer, so we opted for the first ten possible values, i.e. 1 through 10, to determine whether there exists a local optimum in this range. α can take any positive real value. However, we argue that very large values for α would make the approach overly permissive. E.g. with a value of 1, all lengths from 0 up to twice the given length would support the legitimacy of the observed file. Hence, an attacker could simply create a very large segment and be sure that it would be supported by the model, if it appeared in the correct order. Thus, we select 0.5 as a reasonable upper bound for α. From this starting point, we chose 10 evenly distributed values, i.e. set α to 0.05, 0.1 etc. up to and including 0.5. Following this methodology, for JPEG files the most restrictive configuration $\tau = 10$ and $\alpha = 0.05$ scored best. For PNG files we chose the configuration $\tau = 2$ and $\alpha = 0.1$.

6.4 Stegdetect: Append and Invisible Secrets

Provos and Honeyman published several papers on the topic of hiding messages in JPEG files and detecting such embeddings, which we briefly discuss in section 8. While their work focussed on detecting hidden messages in image data, the reference implementation of their Stegdetect [48] tool also contains two methods called *append* and *invisible secrets*. The first method checks whether a file contains at least 4 additional bytes behind the end of the image data. The *invisible secrets* method on the other hand checks whether a comment segment starts with an integer reflecting the length of the following payload. We disabled all other detection methods to avoid triggering unnecessary false positives. However, their implementation was unable to parse a significant fraction of the files in the test sets. We include the fraction that could not be handled as *error* in our comparison to allow our readers to account for these files.

6.5 Results

The left plot in fig. 8 indicates the detection performance of both SAD THUG, indicated by green boxes, and Stegdetect for JPEG files. Only SAD THUG is able to process PNG files and thus the right hand side of fig. 8 shows results solely for our approach. For Stegdetect, we show the true classification ratio using blue boxes and the error ratio, as explained in section 6.4, in red. Given that all values are close to either 0 or 1, we split the graph into an upper and a lower part. The upper part contains the upper 6% range while the lower part contains the lower 6% range, respectively. There were no observations in between these intervals.

As indicated by fig. 8 a), the worst true negative ratio SAD THUG achieved for JPEG files was 99.33% with a maximum of 99.59% and mean 99.48%. Stegdetect on the other hand achieved a mean true negative ratio of 95.45%. This is due to the fact that a surprisingly large number of the JPEG files in our base data set contain data appended behind their EOI marker, as discussed in section 6.2.1. SAD THUG implicitly compensates for this, resulting in a far better true negative ratio than Stegdetect. However, as a side effect, SAD THUG also accepts some files that contain a message added using the *append* paradigm. In section 9.2, we discuss how this can be fixed easily. While we expected Stegdetect to classify files with *append*-based embeddings perfectly, ranging from Cerber to Tropic Trooper in fig. 8, it does not. However, the difference is explained by its failure to parse a significant fraction of the files and is thus, on its own, not indicative of a shortcoming of the method.

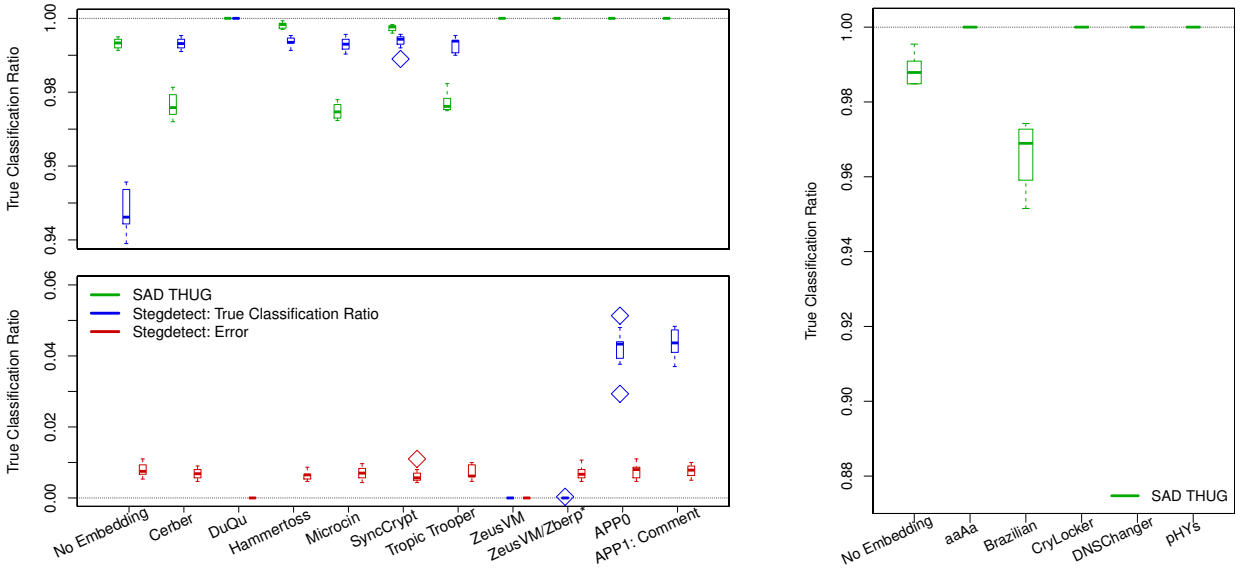
The picture changes once we consider the remaining methods. Here, SAD THUG achieves a 100% true positive ratio while Stegdetect does not detect any ZeusVM file and a parsing error triggers its only true positive for the ZeusVM/Zberp* data set. As discussed above, the files in the ZeusVM/Zberp(*) data sets always end with an end-of-image marker and thus do not trigger Stegdetect's heuristic. The *APP*0 and *APP*1*: Comment* data sets on the other hand include any residual data that was present in the files used to construct them. Hence, here Stegdetect does not detect the actual embedding but the residual data in the base data set. Thus, one could argue that the 2.93% to 5.13% true positives it achieves are in fact false positives.

On the right hand side of fig. 8, we see SAD THUG's detection results for the PNG data sets. We are not aware of any other approach for classifying these files and hence cannot provide a basis for comparison. Here, SAD THUG correctly classifies all files across all cross-validation steps for all except two data sets. For the Brazilian EK's method, which uses the *append* paradigm, results are again distorted by residual data present in the base data set. Here, up to 4.85% of the files are incorrectly classified as benign with a mean true classification ratio of 96.59%. At the same time, SAD THUG achieves a mean true positive ratio of 98.88%. There was no obvious pattern with respect to what files caused the usually single digit count of false positives in each group.

To summarize, SAD THUG achieves very high true classification ratios for both JPEG and PNG files. It classifies several data sets perfectly but is somewhat impeded with respect to *append*-based methods by the presence of a large number of files with residual data in our training data. Here, the worst true classification ratios is 95.15% while the overall average ratios are 99.25% for both JPEG and PNG files. Stegdetect on the other hand scores well for *append*-based methods but fails to detect methods relying on other paradigms. Additionally and in contrast to SAD THUG, Stegdetect causes a large number of false positives, 5.26% on average.

7 Limitations

While our evaluation in section 6 shows that our approach is very effective with regard to detecting embedded messages that change the structure of JPEG or PNG files, it is not designed to detect embeddings in the encoded image data. Thus if an attacker chooses to embed messages in the image data stored in a file, this fact cannot be detected using our approach. A large number of approaches exist that do attempt to detect such embeddings (cf. section 8). With respect to detecting structural embeddings, SAD THUG significantly outperforms the only previous method attempting to solve this problem.



(a) JPEG

(b) PNG

Figure 8: The classification performance of SAD THUG and two Stegdetect methods for JPEG files and SAD THUG's performance for the PNG format.

Similarly, our prototype could be evaded by using a file type that it currently does not support. However, there are several points that mitigate this limitation. First, our approach is agnostic with respect to file types and the prototype parser could simply be extended to parse the structure of another file type. Second, an ALG may expect to observe files of one type much less often than others. As we pointed out in section 2, a web application firewall (WAF) typically observes far more images than HTML documents, since each HTML document usually references dozens of image files. While PDF, DOC or XLS files are often provided as downloads, they generally make up a much smaller fraction of a website's content than HTML documents. Therefore, WAFs may refer to more computationally expensive methods, like on-the-fly conversion into image files, or even require user interaction before letting such files pass through them.

Like all supervised machine learning-based approaches, our approach's effectiveness depends on the training data set. A training data set that is not representative for the benign data observed in the classification phase may increase our approach's false positive ratio. For instance, some programs, e.g. image optimizers, write files with an unusual structure. If for a given program of that kind no files were present in the training data, SAD THUG is likely – and rightfully so – to classify their files as anomalies. However, due to SAD THUG's generalization properties, this can usually be remediated by adding a small number of curated files from that software to the training data.

Like all supervised machine learning-based approaches, SAD THUG is to some degree vulnerable to poisoning attacks. If an attacker manages to inject a large number of files into its target's training data set, this will have a predictable effect on the resulting automaton. Thus, it could try to create transitions in the automaton that would accept the structural anomalies created by its approach. In section 9.2, we discuss several avenues for future work that may mitigate this threat.

Finally, short of manipulating the target's automaton, an attacker could make informed guesses about it as well as about the target's parameterization to devise a strategy to bypass SAD THUG. Generally, such a strategy would allow an attacker to add a few bytes to each variable length segment in a file, possibly at the cost of the file's compatibility with common decoders. i.e. even when an attacker successfully implements a method that bypasses SAD THUG, it will only be able to transfer a small number of bytes per file – compared against an arbitrary number of bytes with structural embedding in general.

8 Related Work

In this section, we provide a brief overview of related work. We focus on three areas. First, we take a quick look at legitimate use of steganography for censorship circumvention. Second, we provide an overview of other approaches for detecting malware or its communications in settings similar to that sketched in section 2. Finally,

we discuss other methods for detecting steganographic message exchanges and their utility with respect to structural embedding methods. Other approaches that apply similar machine learning methods for solving information security challenges – Sivakorn's HVLearn [54] or Görnitz et al.'s work [24] just to name two – provide a valuable background for this work. However, space constraints do not allow us to discuss them in due detail here.

Several systems have been proposed for bypassing censorship systems that may act like an application level gateway in our threat model. While SAD THUG was designed to prevent unwanted communications from malware, the problems are obviously related. Approaches designed to circumvent censorship could be employed to bypass legitimate restrictions according to our threat model while approaches like SAD THUG could be used to detect attempts to circumvent censorship. Systems like Burnett et al.'s Collage [18], Invernizzi et al.'s MIAB [32] or Feamster et al.'s Infranet [22] use stegosystems like Outguess [47] or HUGO [45] to hide messages in JPEG image data. Thus, by their choice of cover media, they are not affected by SAD THUG. Mohajeri et al.'s SkypeMorph [40] and Weinberg et al.'s StegoTorus [57] replicate or hide data in voice-over-IP traffic – which could not traverse a reasonably configured ALG in our threat model. However, StegoTorus can also hide data in HTTP headers and JavaScript, PDF or SWF files. Since our prototype currently only supports JPEG and PNG files, these methods are unaffected by SAD THUG. However by adding appropriate parsers, it may be able to detect StegoTorus's data hiding methods. Finally, Wustrow et al.'s TapDance [58] requires that the attacker's system in the protected network is able to engage in a TLS connection with a system outside that network. This method is not applicable if the ALG conducts man-in-the-middle attacks against TLS connections. If it does not, SAD THUG would not be able to inspect the data transfered and obfuscation on the payload level would not be necessary anyway.

Switching to the position of the ALG in our threat model, we first take a look at Bartos et al.' approach [13] which analyzes an HTTP proxy's log files. While the approach is very lightweight, in this domain, data in- or exfiltration attacks using image files are practically indistinguishable from legitimate transfers and thus their method cannot provide the utility of SAD THUG.

Similarly, Rahbarinia et al.'s Mastino approach [50], Stringhini et al.'s Shady Paths method [56] and Kwon et al.'s approach [35] use the observation that exploit kits often send browsers through a chain of redirects before delivering the actual exploit. However, this limits these approach's utility to the infection phase and even there the redirects are not a technical necessity. More so, when exploit code is extracted from an image file by an otherwise inconspicuous JavaScript, a technique used by several exploit kits, e.g. Angler [44], Astrum [4] or Sundown [36], the approaches are unlikely to detect the attack. Finally, they cannot detect C&C interactions using hidden messages in image files. The same holds for Invernizzi et al.'s Nazca approach [33] but simply for the reason that they explicitly ignore media files like images.

SpyProxy, proposed by Moshchuk et al. [42], is limited to detecting successful exploitation attempts but not impeded by the use of steganography in the process. To the users in the network, it serves as a proxy but before delivering unknown content to a client, it redirects the respective URL to a farm of sandboxes and only if its rendering does not trigger a sandbox violation, it is relayed to the user. Taylor et al. use a similar approach but use honeyclients to impersonate the client requesting a conspicuous resource. Like all sandbox-based approaches, they are resource-intensive and also subject to evasion techniques like busy-waits or fingerprinting. Gu et al.'s BotMiner [26] is one of the few approaches that may detect C&C communications after infection. However, not only does it heavily rely on other sensors but also on observing communications with external hosts that do not occur in our threat model. Similarly, Yu et al.'s PSI approach [60] does not implement a detection method of its own but provides a framework integrating existing network-based detection methods, like the Bro and Snort IDS or the Squid HTTP proxy. Thus, while it cannot detect the attacks SAD THUG is designed to detect, it could integrate our approach to provide comprehensive protection against them.

Finally, we want to take a brief look at approaches for detecting network-based steganography using JPEG files. Provos, partly in conjunction with Honeyman, published a small series of papers on hiding messages in JPEG files and detecting such embeddings [48, 47, 49]. Like the other methods discussed below, their methods are concerned with the embedding in or detection of embeddings in the image data of these files. The Stegdetect tool described in [48] uses a small set of specialized χ^2 tests on the DCT coefficient distribution of the file in question to detect one of three embedding algorithms. Additionally, as we pointed out in section 6, the Stegdetect tool contains methods for detecting structural embeddings like the ones we detect with SAD THUG. While these methods were not covered by the respective paper, we included them in our evaluation to determine their effectiveness and provide a comparison for our own method. Our evaluation in section 6 shows that Stegdetect performs well for embedding methods based on the append paradigm but effectively fails to detect embeddings using other methods. Also, for JPEG files SAD THUG scores a mean false positive ratio that is one order of magnitude below that of Stegdetect.

In another statistical approach to detecting information hiding in DCT coefficients, Andriotis et al. [11] use Benford's law on the distribution of the DCT coefficients to determine whether they carry a hidden message. Barbier, Filiol and Mayoura's method [12] on the other hand uses a training set to derive the probability density for individual bits of the encoded coefficients. If a suspicious file does not match these ratios, is is considered malicious. The work by Cogranne, Denemark and Fridrich [21] uses a roughly similar approach but employs advanced techniques to derive their empirical model and test suspicious images against it. Despite their indisputable merit, these approaches do not solve the problem at hand. Their methods are designed to detect anomalies in the image data – which is disregarded by our approach – and do not consider information hidden in the structure of image files. SAD THUG on the other hand has demonstrated its ability to very reliably detect this kind of embedding in the evaluation presented in section 6.

9 Conclusions and Future Work

9.1 Conclusions

In this paper, we presented SAD THUG, an approach for detecting structural anomalies in image files caused by hiding messages in them. It derives an abstract model for the legitimate structure of container files from a training set and verifies whether newly observed files correspond to that model to classify them as either benign or malicious. SAD THUG achieved perfect classification across all cross-validation data sets for eight methods and scored well or very well for the remaining sets. Its mean false positive ratio was just 0.68% for JPEG files and 1.12% for PNG files. Hence, in this paper we presented a very effective solution to a problem faced by computer users and administrators around the world today.

9.2 Future Work

Currently, our approach is limited to the most common embedding methods that change the structure but not the image data in JPEG and PNG files. Nevertheless, future malware could rely on DCT coefficient-based steganography in JPEG files and some malware has been observed abusing PNG image data to hide its communication. Also, malware could use a combination of structural and coefficient-based embedding to minimize the observable effect in each domain. Thus, our approach should be integrated with an approach or approaches that can detect embeddings in image data to provide comprehensive detection.

In section 6.5, we pointed out that a surprisingly large fraction of image files referenced by popular websites contain additional bytes behind their image data. This had some effect on SAD THUG's ability to detect embedding methods with a similar effect on the cover file's structure. As highlighted by this observation – like for all machine learning-based approaches – attackers could try to influence our method's ability to detect their attacks by poisoning its training set.

There are several avenues that should be explored to mitigate this threat. First and foremost, we could simply remove residual data in the training data set as well as in files delivered to systems. This would effectively prevent the establishment of a covert channel using a large fraction of the methods discussed in this paper. For the remaining methods, SAD THUG scored perfectly. We abstained from simulating this approach for our evaluation because that would have completely voided Stegdetect's detection.

Additionally, the training data could be hardened by not including files from sites that allow users to upload images. Thus, attackers would have to compromise each website they want upload data to. The effect of this approach could be even increased by using a cross-validation approach. Here, a given website's images would be verified against an automaton trained only on other page's files, i.e. an attacker would have to compromise even more websites based on the construction of the training data set. Finally, instead of using absolute counts to determine whether a transition has been observed sufficiently often to include it in our model, we could use weights that depend on the input data. These weights could for instance be scaled to limit the influence that either individual files or sources have on SAD THUG's automaton. While SAD THUG is already surprisingly robust against a skewed training set, we believe that these methods would not only improve its reliability with respect to classification in general but also render it close to impossible to attack by poisoning its training set.

10 Acknowledgments

The authors would like to express their gratitude towards the many people that supported the efforts leading up to this work. In particular, Daniel Plohmann of Fraunhofer FKIE, who does not only preserve the most profound knowledge on reverse engineering and malware of all kinds and creeds but is also always willing to share his knowledge and insights. Matthew Smith of the University of Bonn identified key factors that allowed us to significantly improve our evaluation. Among others, Elmar Padilla of Fraunhofer FKIE provided some comments and feedback on an earlier version of this paper. Finally, we would like to thank the anonymous reviewers for their helpful comments and remarks!

References

[1] W32.duqu: The precursor to the next stuxnet version 1.4. Tech. rep., Symantec, 2011.

[2] Zberp banking trojan: A hybrid of carberp and zeus. https://blog.emsisoft.com/2014/05/27/zberp-banking-trojan-a-hybrid-of-carberp-and-zeus/, 2014. EmsiSoft.

[3] Home routers under attack via malvertising on windows, android devices. https://www.proofpoint.com/us/threat-insight/post/home-routers-under-attack-malvertising-windows-android-devices, 2016. Proofpoint.

[4] Readers of popular websites targeted by stealthy stegano exploit kit hiding in pixels of malicious ads. https://www.welivesecurity.com/2016/12/06/readers-popular-websites-targeted-stealthy-stegano-exploit-kit-hiding-pixels-malicious-ads/, 2016. ESET Research.

[5] Mcafee labs threats report, june 2017. Tech. rep., McAfee, 2017.

[6] ABRAMS, L. Synccrypt ransomware hides inside jpg files, appends .kk extension. https://www.bleepingcomputer.com/news/security/synccrypt-ransomware-hides-inside-jpg-files-appends-kk-extension/, 2017. Bleeping Computer.

[7] ALEXA. The top 500 sites on the web. http://www.alexa.com/topsites, 2017.

[8] ALINTANAHIN, K. Operation tropic trooper: Relying on tried-and-tested flaws to infiltrate secret keepers. Tech. rep., Trend Micro, 2015.

[9] ALPEROVITCH, D. Bears in the midst: Intrusion into the democratic national committee. https://www.crowdstrike.com/blog/bears-midst-intrusion-democratic-national-committee/, 2016. CrowdStrike.

[10] AMIN, R. M., RYAN, J. J. C. H., AND VAN DORP, J. Detecting targeted malicious email. *IEEE Security & Privacy 10*, 3 (2012).

[11] ANDRIOTIS, P., OIKONOMOU, G., AND TRYFONAS, T. JPEG steganography detection with benford's law. *Digital Investigation 9*, 3–4 (2013).

[12] BARBIER, J., FILIOL, E., AND MAYOURA, K. Universal detection of JPEG steganography. *Journal of Multimedia 2*, 2 (2007).

[13] BARTOS, K., SOFKA, M., AND FRANC, V. Optimized invariant representation of network traffic for detecting unseen malware variants. In *Proceedings of the USENIX Security Symposium* (2016).

[14] BENCSÁTH, B., ÁCS-KURUCZ, G., MOLNÁR, G., VASPÖRI, G., BUTTYÁN, L., AND KAMARÁS, R. Duqu 2.0: A comparison to duqu. Tech. rep., CrySyS Lab, 2015.

[15] BENCSÁTH, B., PÉK, G., BUTTYÁN, L., AND FELEGYHAZI, M. Duqu: A stuxnet-like malware found in the wild. Tech. rep., CrySyS Lab, 2011.

[16] BERDNIKOV, V., KARASOVSKY, D., AND SHULMIN, A. Microcin malware: Technical details and indicators of compromise version 1.2 (september 25, 2017). Tech. rep., Kaspersky Lab, 2017.

[17] BLOND, S. L., URITESC, A., GILBERT, C., CHUA, Z. L., SAXENA, P., AND KIRDA, E. A look at targeted attacks through the lense of an NGO. In *Proceedings of the USENIX Security Symposium* (2014).

[18] BURNETT, S., FEAMSTER, N., AND VEMPALA, S. Chipping away at censorship firewalls with user-generated content. In *Proceedings of the 19th USENIX Conference on Security* (Berkeley, CA, USA, 2010), USENIX Association, pp. 29–29.

[19] CAMERA & IMAGING PRODUCTS ASSOCIATION. Exchangeable image file format for digital still cameras - Exif version 2.31, 2016.

[20] CIMPANU, C. Steganography is very popular with exploit kits all of a sudden. https://www.bleepingcomputer.com/news/security/steganography-is-very-popular-with-exploit-kits-all-of-a-sudden/, 2016. Bleeping Computer.

[21] COGRANNE, R., DENEMARK, T., AND FRIDRICH, J. Theoretical model of the fld ensemble classifier based on hypothesis testing theory. In *Proceedings of the International Workshop on Information Forensics and Security* (2014).

[22] FEAMSTER, N., BALAZINSKA, M., HARFST, G., BALAKRISHNAN, H., AND KARGER, D. R. Infranet: Circumventing web censorship and surveillance. In *USENIX Security Symposium* (2002), pp. 247–262.

[23] FIREEYE. Hammertoss: Stealthy tactics define a russian cyber threat group. Tech. rep., FireEye, 2015.

[24] GÖRNITZ, N., KLOFT, M., RIECK, K., AND BREFELD, U. Active learning for network intrusion detection. In *Proceedings of the 2Nd ACM Workshop on Security and Artificial Intelligence* (New York, NY, USA, 2009), AISec '09, ACM, pp. 47–54.

[25] GRIER, C., BALLARD, L., CABALLERO, J., CHACHRA, N., DIETRICH, C. J., LEVCHENKO, K., MAVROMMATIS, P., MCCOY, D., NAPPA, A., PITSILLIDIS, A., PROVOS, N., RAFIQUE, M. Z., RAJAB, M. A., ROSSOW, C., THOMAS, K., PAXSON, V., SAVAGE, S., AND VOELKER, G. M. Manufacturing compromise: the emergence of exploit-as-a-service. In *Proceedings of the Conference on Computer and Communications Security* (2012).

[26] GU, G., PERDISCI, R., ZHANG, J., AND LEE, W. Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *Proceedings of the USENIX Security Symposium* (2008).

[27] GUGELMANN, D., STUDERUS, P., LENDERS, V., AND AGER, B. Can content-based data loss prevention solutions prevent data leakage in web traffic? *IEEE Security & Privacy 13*, 4 (2015).

[28] HARDY, S., CRETE-NISHIHATA, M., KLEEMOLA, K., SENFT, A., SONNE, B., WISEMAN, G., GILL, P., AND DEIBERT, R. J. Targeted threat index: Characterizing and quantifying politically-motivated targeted malware. In *Proceedings of the USENIX Security Symposium* (2014).

[29] IMAGEMAGICK STUDIO LLC. Imagemagick website. https://www.imagemagick.org, 2017.

[30] INTERNATIONAL TELECOMMUNICATIONS UNION. Digital compression and coding of continuous-tone still images: Requirements and guidelines, 1992.

[31] INTERNATIONAL TELECOMMUNICATIONS UNION. Digital compression and coding of continuous-tone still images: Jpeg file interchange format (JFIF), 2011.

[32] INVERNIZZI, L., KRUEGEL, C., AND VIGNA, G. Message in a bottle: Sailing past censorship. In *Proceedings of the 29th Annual Computer Security Applications Conference* (New York, NY, USA, 2013), ACM, pp. 39–48.

[33] INVERNIZZI, L., MISKOVIC, S., TORRES, R., KRUEGEL, C., SAHA, S., VIGNA, G., LEE, S., AND MELLIA, M. Nazca: Detecting malware distribution in large-scale networks. In *Proceedings of the Network and Distributed Systems Security Symposium* (2014).

[34] KOHEI, K., CHEN, J. C., AND JOCSON, J. Picture perfect: Crylocker ransomware uploads user information as png files. http://blog.trendmicro.com/trendlabs-security-intelligence/picture-perfect-crylocker-ransomware-sends-user-information-as-png-files/, 2016. Trend Micro.

[35] KWON, B. J., MONDAL, J., JANG, J., BILGE, L., AND DUMITRAS, T. The dropper effect: Insights into malware distribution with downloader graph analytics. In *Proceedings of the Conference on Computer and Communications Security* (2015).

[36] LI, B., AND CHEN, J. C. Updated sundown exploit kit uses steganography. http://blog.trendmicro.com/trendlabs-security-intelligence/updated-sundown-exploit-kit-uses-steganography/, 2016. Trend Micro.

[37] MARQUES, T. Png embedded - malicious payload hidden in a png file. https://securelist.com/png-embedded-malicious-payload-hidden-in-a-png-file/74297/, 2016. Kaspersky Lab.

[38] MCMILLEN, D. Steganography: A safe haven for malware. https://securityintelligence.com/steganography-a-safe-haven-for-malware/, 2017. IBM Security Intelligence.

[39] MILLMAN, R. Steganography attacks - using code hidden in images - increasing. https://www.scmagazineuk.com/steganography-attacks–using-code-hidden-in-images–increasing/article/680144/, 2017. SC Magazine UK.

[40] MOHAJERI MOGHADDAM, H., LI, B., DERAKHSHANI, M., AND GOLDBERG, I. Skypemorph: Protocol obfuscation for tor bridges. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 97–108.

[41] MOORE, T., LEONTIADIS, N., AND CHRISTIN, N. Fashion crimes: trending-term exploitation on the web. In *Proceedings of the Conference on Computer and Communications Security* (2011).

[42] MOSHCHUK, A., BRAGIN, T., DEVILLE, D., GRIBBLE, S. D., AND LEVY, H. M. SpyProxy: Execution-based detection of malicious web content. In *Proceedings of the USENIX Security Symposium* (2007).

[43] PAGANONI, S. Malicious javascript dataset. https://github.com/geeksonsecurity/js-malicious-dataset, 2017.

[44] PC'S XCETRA SUPPORT. Angler exploit kit steganography. https://pcsxcetrasupport3.wordpress.com/2017/02/06/angler-exploit-kit-steganography/, 2017.

[45] PEVNÝ, T., FILLER, T., AND BAS, P. Using high-dimensional image models to perform highly undetectable steganography. In *Information Hiding* (Berlin, Heidelberg, 2010), R. Böhme, P. W. L. Fong, and R. Safavi-Naini, Eds., Springer Berlin Heidelberg, pp. 161–177.

[46] PLOHMANN, D., CLAUSS, M., ENDERS, S., AND PADILLA, E. Malpedia: A collaborative effort to inventorize the malware landscape. *The Journal on Cybercrime & Digital Investigations 3*, 1 (2017).

[47] PROVOS, N. Defending against statistical steganalysis. In *Proceedings of the USENIX Security Symposium* (2001).

[48] PROVOS, N., AND HONEYMAN, P. Detecting steganographic content on the internet. In *Proceedings of the Network and Distributed Systems Security Symposium* (2002).

[49] PROVOS, N., AND HONEYMAN, P. Hide and seek: An introduction to steganography. *IEEE Security & Privacy 1*, 3 (2003).

[50] RAHBARINIA, B., BALDUZZI, M., AND PERDISCI, R. Real-time detection of malware downloads via large-scale url->file->machine graph mining. In *Proceedings of the Asia Conference on Computer and Communications Security* (2016).

[51] SCHWARZ, D. Zeusvm: Bits and pieces. Tech. rep., Arbor Networks, 2015.

[52] SEALS, T. Steganography sees a rise in 2017. https://www.infosecurity-magazine.com/news/steganography-sees-a-rise-in-2017/, 2017. Infosecurity Magazine.

[53] SHULMIN, A., AND KRYLOVA, E. Steganography in contemporary cyberattacks. https://securelist.com/steganography-in-contemporary-cyberattacks/79276/, 2017. Kaspersky Lab.

[54] SIVAKORN, S., ARGYROS, G., PEI, K., KEROMYTIS, A. D., AND JANA, S. Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations. In *2017 IEEE Symposium on Security and Privacy* (May 2017), pp. 521–538.

[55] SOOD, A. K., AND ENBODY, R. J. Targeted cyberattacks: A superset of advanced persistent threats. *IEEE Security & Privacy 11*, 1 (2013).

[56] STRINGHINI, G., KRUEGEL, C., AND VIGNA, G. Shady paths: Leveraging surfing crowds to detect malicious web pages. In *Proceedings of the Conference on Computer and Communications Security* (2013).

[57] WEINBERG, Z., WANG, J., YEGNESWARAN, V., BRIESEMEISTER, L., CHEUNG, S., WANG, F., AND BONEH, D. Stegotorus: A camouflage proxy for the tor anonymity system. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), ACM, pp. 109–120.

[58] WUSTROW, E., SWANSON, C. M., AND HALDERMAN, J. A. Tapdance: End-to-middle anticensorship without flow blocking. In *Proceedings of the USENIX Security Symposium* (2014).

[59] XYLITOL. Zeusvm and steganography. http://www.xylibox.com/2014/04/zeusvm-and-steganography.html, 2014.

[60] YU, T., FAYAZ, S. K., COLLINS, M., SEKARÂĂĂ, V., AND SESHAN, S. PSI: Precise security instrumentation for enterprise networks. In *Proceedings of the Network and Distributed Systems Security Symposium* (2017).

Notes

[1]Microsoft's 22nd Security Intelligence Report provides data for the first three months in 2017 separately while earlier reports aggregate data for each quarter. Due to the unknown overlap between the underlying systems, the 22nd report cannot be used to identify a trend with respect to the earlier reports.

[2]SHA256: `7b6cc23d545dea514628669a1037df88b278312↩f495b97869b40882ca554fa9a`