Hindawi
Security and Communication Networks
Volume 2022, Article ID 4477317, 12 pages
https://doi.org/10.1155/2022/4477317

Research Article

Efficient Detection and Recovery of Malicious PowerShell Scripts Embedded into Digital Images

Andreas Schaffhauser,[1] **Wojciech Mazurczyk**,[1,2] **Luca Caviglione**,[3] **Marco Zuppelli**,[3] **and Julio Hernandez-Castro**[4]

[1]*FernUniversität in Hagen, Hagen, Germany*
[2]*Warsaw University of Technology, Warszawa, Poland*
[3]*Institute of Applied Mathematics and Information Technologies, Genova, Italy*
[4]*University of Kent, Canterbury, UK*

Correspondence should be addressed to Andreas Schaffhauser; andreas.schaffhauser@htwsaar.de

Received 11 September 2021; Accepted 24 May 2022; Published 29 June 2022

Academic Editor: Vincenzo Conti

Due to steady improvements in defensive systems, malware developers are turning their attention to mechanisms for cloaking attacks as long as possible. A recent trend exploits techniques like Invoke-PSImage, which allows embedding a malicious script within an innocent-looking image, for example, to smuggle data into compromised devices. To address such a class of emerging threats, new mechanisms are needed, since standard tools fail in their detection or offer poor performance. To this aim, this work introduces Mavis, an efficient and highly accurate method for detecting hidden payloads, retrieving the embedded information, and estimating its size. Experimental results collected by considering real-world malicious PowerShell scripts showcase that Mavis can detect attacks with a high accuracy (100%) while keeping the rate of false positives and false negatives very low (0.01% and 0%, respectively). The proposed approach outperforms other solutions available in the literature or commercially through "as a service" model.

1. Introduction

According to a recent report by McAfee [1], the number of new malware samples grew by 11.5% in *Q*2 2020, partially fueled by the current COVID-19 pandemic. Among the various offensive mechanisms employed by attackers, PowerShell-based malware sharply increased by 117% from *Q*1 to *Q*2, mainly due to its cloaking abilities. For instance, commands can be generated dynamically, launched from memory, and easily obfuscated or injected, making both forensic analysis and detection significantly more arduous [2]. PowerShell scripts can also take advantage of the recent trend of exploiting information hiding techniques for embedding malicious commands or exfiltrating sensitive data within innocent-looking contents, such as digital images [3]. To this aim, one of the most recent and popular tools employed by criminals is Invoke-PSImage [4] which allows embedding malicious PowerShell scripts into digital images. The Invoke-PSImage technique has been already used to enhance several existing malware functionalities and to deploy multiple attacks, such as in the campaign against the Pyeongchang Olympic Games [5], in the diffusion of the Greystars ransomware [6], or in new variants of Ursnif [7]. It has also been deployed in the malicious IAmTheKing toolset [8] and used for the creation of a backdoor in the Bandook malware [9].

Since the adoption of steganography to conceal information in digital images is becoming increasingly popular among cybercriminals [10], being able to detect its usage is of paramount importance. However, this could be very challenging, as recognizing the existence of embedded data heavily depends on the adopted algorithm and the image format [3, 10, 11]. Moreover, in many real-world scenarios, the volume of images that has to be inspected could lead to

scalability issues [3]. Concerning threats leveraging Invoke-PSImage, the lack of efficient detection techniques makes the development of new methodologies an essential research topic [2]. Therefore, this work introduces Mavis (https://en.Wikipedia.org/wiki/Mavis_Batey), a tool for discovering malicious PowerShell scripts embedded in digital images via the Invoke-PSImage method. To this aim, we developed different detection algorithms leveraging the characteristics of the injection process (e.g., how colour channels are altered) as well as features of the scripts (e.g., recurrent patterns disclosing a textual content). Compared to similar approaches, our tool offers the additional advantage that it can also estimate the size of the malicious script embedded in the image and retrieve the payload. To test the effectiveness of Mavis, we created an ad hoc dataset composed of 45,000 images containing various malicious PowerShell scripts. Collected results indicate that Mavis has a detection rate close to 100%, with a very low false positive rate and a reduced computational footprint.

Summing up, the contributions of this work are as follows: (i) a set of detection, estimation, and recovery algorithms for efficiently thwarting Invoke-PSImage-based attacks, (ii) a vis-á-vis comparison with state-of-the-art techniques and commercial services, and (iii) a full public release of Mavis and the dataset used for the experiments (https://github.com/s3venup/Mavis).

The rest of the paper is structured as follows. Section 2 reviews past works on detecting steganographic threats, while Section 3 introduces the considered attack model and background information. Section 4 showcases details on Mavis, and Section 5 presents the experimental methodology. Section 6 discusses the obtained results, whereas Section 7 compares Mavis with similar tools. Finally, Section 8 concludes the paper and proposes future research directions.

2. Related Work

Images and videos have been extensively used to conceal secret information. Specifically, they have been the target of techniques for appending arbitrary contents in metadata or at the end of a file, as well as proper steganographic algorithms for injecting information in the Least Significant Bit (LSB) or in the coefficients of the discrete cosine transformation on the basis of many compression algorithms [10, 12–14]. Moreover, sophisticated approaches using adaptive mechanisms have also been proposed to reduce the visible artifacts caused by the hiding processes (e.g., algorithms exploiting dependencies among adjacent pixels) [15].

Even if the literature abounds in ideas or frameworks for image steganalysis (see, e.g., [12, 13] and the references therein), there has always been a historical shortage of tools specifically tailored for revealing attacks or performing threat detection. To this extent, a popular tool is Stegdetect introduced in 2001 [16], unfortunately no longer updated. A similar case is represented by Stegsecret (https://stegsecret.sourceforge.net/), which had a great potential but has not been updated since 2007. In addition, tools do not typically provide a very general-purpose solution, and many are mostly aimed at visual attacks or techniques that can be detected with a simple fingerprint-based approach. Other detection tools exist, but they generally lack in reliability since they have not been extensively tested. There are also solutions that are based on commercial or proprietary code, which is against best practices; see, for example, the case of StegoHunt (https://www.wetstonetech.com/products/stegohunt-steganography-detection/) by WetStone Technologies.

As a consequence, the number of tools that can reliably detect Invoke-PSImage-based threats "out of the box" is very limited. To the best of our knowledge, only two tools can be considered suitable for handling this task. The first is StegExpose (https://github.com/b3dk7/StegExpose), which is a general-purpose LSB steganalysis tool orchestrating the results provided via four different detection methods, that is, Sample Pairs [17], RS Analysis by [18], Chi-Square Attack [19], and Primary Sets [20]. Each algorithm returns a score reflecting the likelihood of an image being malicious: such values are then combined and compared with an empirically set threshold. StegExpose implements two execution modes: default and fast. In the default mode, all algorithms are used in sequence and the outputs are combined. In the fast mode, the tool tries to speed up the analysis by skipping the images considered clean. To this aim, if the value returned by a single algorithm is very far from the decision threshold, the image is considered clean without further processing. The second tool is McAfee Steganography Analysis Tool (https://www.mcafee.com/enterprise/en-us/downloads/free-tools/steganography.html) (SAT), which is an online general steganalysis tool. In essence, it provides a report including whether a content can be considered suspicious, along with a statistical assessment in terms of a confidence metric, a score, and the time spent to analyze the image. Unfortunately, McAfee SAT has some limitations: it cannot process images larger than 1 Mbyte and the implemented algorithms are not publicly available, thus making it a black box.

3. Attack Model

As previously hinted, the Invoke-PSImage tool manipulates the colour values of digital images (e.g., PNG files) with the aim of embedding PowerShell scripts. Figure 1 depicts the reference attack model and the related scenarios observed in an extensive array of threats [10].

As a first step, the user is tricked, for instance, by opening an infected attachment or by visiting a compromised website. When successfully infected, the host of the victim tries to reach a remote server controlled by the attacker to register itself. Upon completing this stage, the attacker usually needs to transmit further malicious code without being spotted by a network intrusion detection system or blocked by a firewall. To this aim, the attacker can use Invoke-PSImage to embed a malicious PowerShell script into an innocent-looking digital image and transmit it to the infected host (see "step 1" and "step 2" in Figure 1). In general, standard security tools do not consider this class of threats or do not have enough computing/storage resources to deeply inspect all multimedia data exchanged through the network. Moreover, eliminating hidden contents could

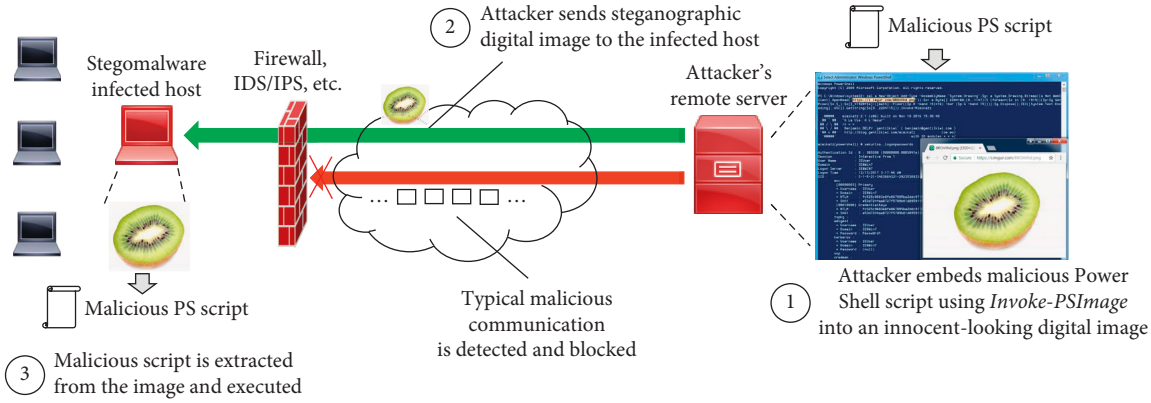

Figure 1: The Invoke-PSImage-based attack scenario.

require manipulating the original file, thus raising privacy and integrity concerns. As the last step of the attack, when the modified image reaches the infected device, the malware extracts the hidden script and executes it (see "step 3" in Figure 1).

To implement the covert communication, Invoke-PSImage subtly modifies digital images by embedding PowerShell scripts into pixels via altering their colour values. This can be done in two different modes. In Mode-1, the tool generates a synthetic image for containing the script. In this case, all 8 bits of the three-colour channels (i.e., red, green, and blue) are used to embed the malicious script. Thus, the resulting PNG file has approximately 50% of the size of the malicious original script. Instead, in Mode-2, the user needs to provide the cover image with a suitable size for injecting the PowerShell script. In this case, only the 4 least significant bits of the green and blue colour channels are used. Regardless of the selected mode, randomly generated values are used to pad the information injected into the colour channels. This happens, for instance, when the malicious script is smaller than the cover image or in Mode-2, which does not use the red colour channel. For performance reasons, such random values are generated only once, when the Invoke-PSImage tool is first launched. This behaviour leads to a recurring "pattern" within the resulting images. Additionally, no nonalphanumeric characters are added to the random values.

4. Detection Methodology

In this section, we describe the approach used by Mavis to detect the hidden payload, estimate its size, and perform the extraction. Specifically, Section 4.1 deals with attacks leveraging Mode-1 of Invoke-PSImage, whereas Section 4.2 focuses on Mode-2. Sections 4.3 and 4.4 provide details on the algorithms for size estimation and extraction of malicious payloads, while Section 4.5 discusses how the distinct functionalities of the embedded scripts can be classified.

4.1. Detection of Mode-1. To detect scripts embedded with Mode-1, the specific distribution of the RGB values can be used as a sort of "signature." In fact, as a first step, Invoke-PSImage calculates the size of the image required for the embedding process and initially sets all RGB values to zero. Next, it generates an alphanumeric, 128-character long, random string. For each character, the value is in the range $[0; 127]$. Due to such an embedding process, the resulting PNG file with hidden data has a very characteristic distribution of the RGB values. Figure 2 showcases the distribution of 5 million RGB values collected in 5,333 images. In this dataset, 99.51% of images have been processed using Invoke-PSImage in Mode-1 to inject malicious scripts in either an obfuscated or deobfuscated form. Images created automatically via Mode-1 have a resolution typically ranging from 8×8 to 64×64 pixels, depending on the size of the script. The benign images used were all 256×256 pixels. In order to compare the same number of pixels in malicious and benign images, the number of benign images must be quite small (i.e., only 0.49%).

Moreover, Figure 2 shows that the RGB values in images with embedded scripts contain no values in the range $[0; 8] \cup [11; 31] \cup [127; 255]$. Images containing deobfuscated scripts have a remarkable peak at RGB value of 48 (total number of occurrences: 548,490), which corresponds to the ASCII character "0." Conversely, images generated using obfuscated scripts have their global maximum at RGB value of 65, that is, "A" in ASCII, with a total number of occurrences of 1,618,144. This is caused by the utilization of Base-64 encoding for the scripts. Base-64-encoded scripts use the character "A" in 32.36% of all cases. In contrast to these observations, benign images display a more or less equal distribution, with frequent maxima in white and black colour values. The experimental results obtained by applying this observation for spotting hidden data will be presented in Section 6.

4.2. Detection of Mode-2. Mode-2 of the tool generates images by exploiting the green and blue channels as hidden carrier, whereas it uses the 4 LSBs of the red channel to store pseudorandom values generated at the beginning and repeated every 109 pixels. This can be used as a signature to spot the presence of a hidden script; that is, the detection can be carried out by searching for identical values over the red channel every 109 pixels.

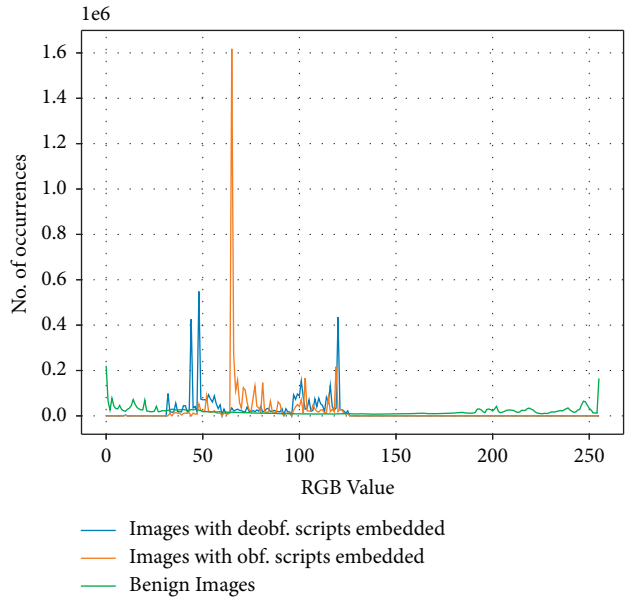

Figure 2: Distribution of 5,000,000 RGB values in benign and malicious images.

During our extensive experimental evaluation, only a completely black image triggered a false positive alarm using the aforementioned detection rule. However, such a case is highly unlikely in realistic, real-world images.

4.3. Size Estimation of the Embedded PowerShell Scripts. Apart from identifying if a digital image contains hidden data, it would be also beneficial evaluating the nature of the cloaked content. Alas, this is not always possible since the attacker could adopt obfuscation or antiforensics techniques, such as scrambling or encrypting the script. As an alternative, we can estimate the size of the PowerShell script and use such information to infer its complexity and the range of its offensive capabilities.

4.3.1. Mavis Size Estimation for Mode-1. In Mode-1, after the injection of the malicious script in the three-colour channels of the cover image, the rest of the RGB values are set to pseudorandom bytes. Since the random selection of the pseudorandom bytes is done via a modulo 113 operation, and the repetition of the pattern can be found after 113 values, the implemented algorithm compares the RGB value at index i with the RGB value at index $i + 113$. If these two byte values are equal, and this equality holds for all consecutive bytes until the end of the image, we assume that this is the start of the pattern used for detection. The size of the repeating pattern is then subtracted from the total file size to obtain the estimated size of the injected script. In cases where less than 114 random bytes have been injected after the malicious payload, we cannot determine the beginning of the repetition. Therefore, in these infrequent cases, the size of the script cannot be estimated in an accurate manner.

4.3.2. Mavis Size Estimation for Mode-2. The size estimation of the injected scripts using Mode-2 is based on the comparison of two consecutive lists containing 113 blue channel nibbles (i.e., 4 LSBs). The green and the blue channel hold always the same amount of nibbles, so only one of these two channels needs to be investigated to estimate the size of the hidden content. The first list contains 113 values from index $i \cdot 113$ to $((i + 1) \cdot 113) - 1$. The second list contains 113 values from $((i + 1) \cdot 113$ to $((i + 2) \cdot 113) - 1$. As long as both lists are not equal, the first list gets the values from the second, the second list gets the next 113 values of the blue channel, and the size of the estimated script is incremented by 113. If the equality is found for the first time, we search for the beginning of the repetition in the two lists from the previous round by reducing these two lists. In the result, the sizes of the two lists and the size of the estimated script are reduced by 1 as long as no equality is found. If the equality is spotted, the correct size of the injected script has been determined.

Due to the fact that only nibbles are compared, there is a possibility that the script nibble and the randomly injected byte nibble are coincidentally equal (i.e., a collision happens). In this case, the size of the script is underestimated and the length of the repeating pattern is overestimated. Because of the 16 possible states of the nibble, the scripts are most often underestimated by 8 bytes.

4.4. Recovering Embedded Scripts with Mode-1 or Mode-2. Apart from detecting malicious images, an important goal for Mavis is the extraction of embedded PowerShell scripts, for instance, to support forensics investigations. Since the creator of the Invoke-PSImage tool has not considered any cryptographic or scrambling mechanisms to secure the embedded information, the malicious content is always located at the beginning of the file, thus making it easily recoverable. This, jointly with the ability of Mavis of estimating the size of the payload, guarantees the recovery of the hidden contents.

Since in Mode-1 a whole byte is injected into the red, green, and blue channels, its ASCII value can be directly extracted from the image, from the start to the length of the estimated embedded data. For Mode-2, the algorithm needs to be slightly modified: in this case, only the 4 LSBs of the green and blue channels are used for data hiding purposes. The blue channel contains the 4 most significant bits of the script byte, while the green channel contains its 4 least significant bits. For recovering the specific ASCII value in Mode-2 images, we have to shift the 4 LSBs of the blue channel by 4 and add them to the 4 LSBs of the green channel.

4.5. Classifying the Functionality of a Script. Another important aspect of Mavis concerns the ability of determining the functionality of the hidden script, based only on its size. To design such a feature, we used the recent dataset of malicious scripts from [21], which contains real-life malicious scripts in both obfuscated and deobfuscated forms. Obtained numerical results will be presented in Section 6.3.

Table 1: Groups of obfuscation techniques as introduced in [22].

Obfuscation techniques group	Description
Encoding	These techniques rely on different encodings to obfuscate the code. In our experiments, we chose ASCII encoding-based obfuscation.
Token	Within this group, it is possible to obfuscate different tokens. Each token has different available obfuscation levels (e.g., 'argument' 1–4, 'command' 1–3, 'comment' 1, 'member' 1–4, 'string' 1-2, etc.). For our purposes, we selected all token obfuscation functions in random order at the highest obfuscation level.
String	Such methods perform obfuscation via string concatenation or reordering. For our evaluation purposes, we chose the string delimited and concatenated technique.
AbstractSyntaxTree	In this class, the obfuscation process uses AbstractSyntaxTree-based rules.

In more detail, we sorted the scripts based on their size and tried to determine whether there are some common keywords that can characterize their functionality. The rationale behind this is that, in general, the larger the size, the more complex the actions they perform. After this analysis, we were able to distinguish the three following categories:

(i) Memory execution: considers malicious scripts containing the *memset* keyword, which typically injects a shell script in binary form into the memory and executes it.

(ii) Shell execution: groups scripts that execute an application via a *shellexecute* command.

(iii) Malware download/rest: over 95% of the scripts in this group are related to the functionalities used to download the malware. In general, scripts belonging to this category are very simple, mainly aiming at establishing a communication path with a remote C&C server, downloading the malware, and executing it locally. However, this category also includes scripts that cannot be easily classified, for example, those that modify registry keys.

Then, for each class, we calculated the mean script size and the related standard deviation. Moreover, we used the Invoke-Obfuscation [22] tool to obfuscate the scripts and extend the remit of our experiments. The tool offers different obfuscation techniques, grouped into four main classes, as presented in Table 1.

Next, the sizes of the deobfuscated scripts were compared to their obfuscated counterparts. Note that the size of the obfuscated script is generally larger than the deobfuscated one. Moreover, we mapped the relationship (in terms of size and functionality) between deobfuscated and obfuscated scripts. For this reason, for each of the obfuscation classes mentioned above, we calculated the average multiplication factor $\overline{f_o}$, which is defined as $\overline{f_o} = \sum x_o/x_d/n$, where x_o is the size of the obfuscated scripts, x_d is the size of the deobfuscated scripts, and n denotes the total number of scripts. Note that such an approach requires that the obfuscation method used on the malicious script must be known in advance, so the correct multiplication factor is applied. Currently, several existing methods are able to infer the used obfuscation technique; see, for instance, the paradigmatic examples reported in [23–25]. Besides, one may convert an obfuscated script to its deobfuscated version by using tools like Revoke-Obfuscation [26] or via an approach based on the work presented in [27].

During our experiments, to infer the functionality of a script from its size, the following steps were carried out. First, we only examined whether the deobfuscated scripts can be correctly assigned to the correct keyword class based on their size using the closest mean size of the respective category. Then, we repeated this step for the obfuscated scripts; however, we accredited a script to the respective category if its size was closest to the mean size of the corresponding category multiplied by factor $\overline{f_o}$.

5. Datasets and Experimental Methodology

In this section, we first showcase the datasets that we utilized during our experimental evaluation. Then, we outline the methodology we followed during our study.

5.1. Preparation of the Datasets. To perform a thorough experimental evaluation of Mavis, we prepared a novel dataset by combining two different sources. The first is *iStego100K* (https://github.com/YangzlTHU/IStego100K), which originally contains 100,000 pairs of clean-stego images. This dataset is denoted in the following as the *digital images* dataset. Then, we considered malicious PowerShell scripts from [21], which contain 4,641 scripts in deobfuscated and 4,018 in obfuscated form. In the following, we refer to such a collection as the *malicious PowerShell scripts* dataset. The latter has been directly used to generate the dataset to test Mode-1. Specifically, it is composed of 4,641 and 4,018 images, generated starting from the deobfuscated and the obfuscated script of the malicious PowerShell scripts dataset.

Instead, to evaluate Mode-2, we created three datasets by considering images with different resolutions, that is, 1024×1024, 512×512, and 256×256:

(i) *clean* dataset: 15,000 images (5,000 images for each resolution),

(ii) *stego-obfuscated* dataset: 15,000 images (5,000 images for each resolution) containing obfuscated PowerShell scripts,

(iii) *stego-deobfuscated* dataset: 15,000 images (5,000 images for each resolution) containing deobfuscated PowerShell scripts.

Table 2: Defined performance metrics.

Performance metric	Description
Correct detection rate	The number of correctly identified cases for a given detection scenario.
False positives (FP) and *False negatives* (FN)	FP occurs when a clean image is identified as steganographically modified, while FN is when an image with an embedded malicious PowerShell script is classified as benign.
$\overline{t_d}$	The average time needed to perform the detection process, which is measured from the moment when the red channel is ready to be investigated, until the specific pattern indicating steganography usage is (or is not) identified.
$\overline{t_e}$	The average time needed to perform the size estimation of the embedded malicious script in the digital image. It is measured from the moment when the green and blue channels are prepared to be investigated until a prediction of the size is completed.
MAPE (mean absolute percentage error)	Specifies, on average, to what extent the estimated size of the injected script is correct. It is calculated as follows: MAPE = $1/n \sum_{i=0}^{n} \lvert As_i - Es_i/As_i \rvert$, where n denotes the number of steganographically modified files used for size estimation, As_i is the actual size of the script, and Es_i is its estimated size.

Table 3: Detection and performance results for Mavis.

Invoke-PSImage Mode-1					
Scripts	Correct detec.	FN rate	$\overline{t_d}$ (ms)	$\overline{t_e}$ (ms)	MAPE (%)
Deobfuscated	4641/4641 (100%)	0/4641 (0%)	7.75	1.65	10.03
Obfuscated	4018/4018 (100%)	0/4018 (0%)	19.18	5.23	0.12
Overall	**100%**	**0%**	**13.05**	**3.31**	**5.43**
Invoke-PSImage Mode-2					
Scripts	Correct detec.	FN rate	$\overline{t_d}$ (ms)	$\overline{t_e}$ (ms)	MAPE (%)
Deobfuscated (256 × 256)	5000/5000 (100%)	0/5000 (0%)	1.1	3.94	0.0009
Deobfuscated (512 × 512)	5000/5000 (100%)	0/5000 (0%)	4.11	3.83	0.0017
Deobfuscated (1024 × 1024)	5000/5000 (100%)	0/5000 (0%)	16.89	4.03	0.0022
Obfuscated (256 × 256)	5000/5000 (100%)	0/5000 (0%)	0.99	9.24	0.0006
Obfuscated (512 × 512)	5000/5000 (100%)	0/5000 (0%)	4.06	9.71	0.0005
Obfuscated (1024 × 1024)	5000/5000 (100%)	0/5000 (0%)	16.29	9.57	0.0006
Overall	**100%**	**0%**	**7.24**	**6.72**	**0.0011**
Scripts	Correct detec.	FP rate	$\overline{t_d}$ (ms)	$\overline{t_e}$ (ms)	MAPE (%)
Clean (256 × 256)	5000/5000 (100%)	0/5000 (0%)	1.15	N/A	N/A
Clean (512 × 512)	4999/5000 (99.98%)	1/5000 (0.02%)	4.44	N/A	N/A
Clean (1024 × 1024)	5000/5000 (100%)	0/5000 (0%)	16.21	N/A	N/A
Overall	**99.99%**	**0.01%**	**7.27**	**N/A**	**N/A**

To summarize, the total number of steganographically modified images for evaluating Invoke-PSImage Mode-1 is 8,659, whereas for Mode-2 it is 30,000. Moreover, for the sake of completeness, for both modes, we utilized 15,000 digital images from the clean images dataset.

5.2. Methodology. To evaluate the performance of Mavis and to have a comparison with other detection tools used as a benchmark, we assessed separately the two working modes implemented by the Invoke-PSImage tool.

The discussed datasets discussed in Section 5.1 have been processed by using the detection methodology introduced in Sections 4.1 and 4.2. We have analyzed two aspects: (i) whether the detection tool is able to correctly spot the presence of the hidden data within the inspected image and (ii) whether it is possible to estimate the size of the hidden data and to roughly infer its functionality.

To evaluate Mavis, we use the metrics defined in Table 2. To obtain numerical results, we conducted experiments by using ad hoc Python scripts running on a machine equipped with an Intel(R) Core(TM) i5-9400H CPU @2.5 GHz with 8 GB RAM.

6. Numerical Results

In this section, we present experimental results proving the effectiveness of Mavis. First, we showcase the performance in terms of detection, and then we discuss the ability of the tool to estimate the size of the script and infer its functionality.

6.1. Hidden Payload Detection. Concerning the detection of hidden malicious payloads, Table 3 reports results obtained by Mavis. As shown, for PowerShell scripts, hidden via Mode-1, Mavis was able to achieve 100% in detection accuracy with 0% false negatives. The time needed to perform the detection of deobfuscated scripts was ~2.5 times shorter than that in the case of obfuscated contents. This could be ascribed to the smaller nature of the deobfuscated contents compared to the obfuscated counterparts.

Similar considerations on the accuracy of the detection can be done when `Invoke-PSImage` is used in Mode-2. In this case, a detection delay of 7.24 ms is observed, but this does not prevent deploying Mavis in many realistic scenarios. Additionally, there is a linear relationship between the detection time and the size of the image. In this case,

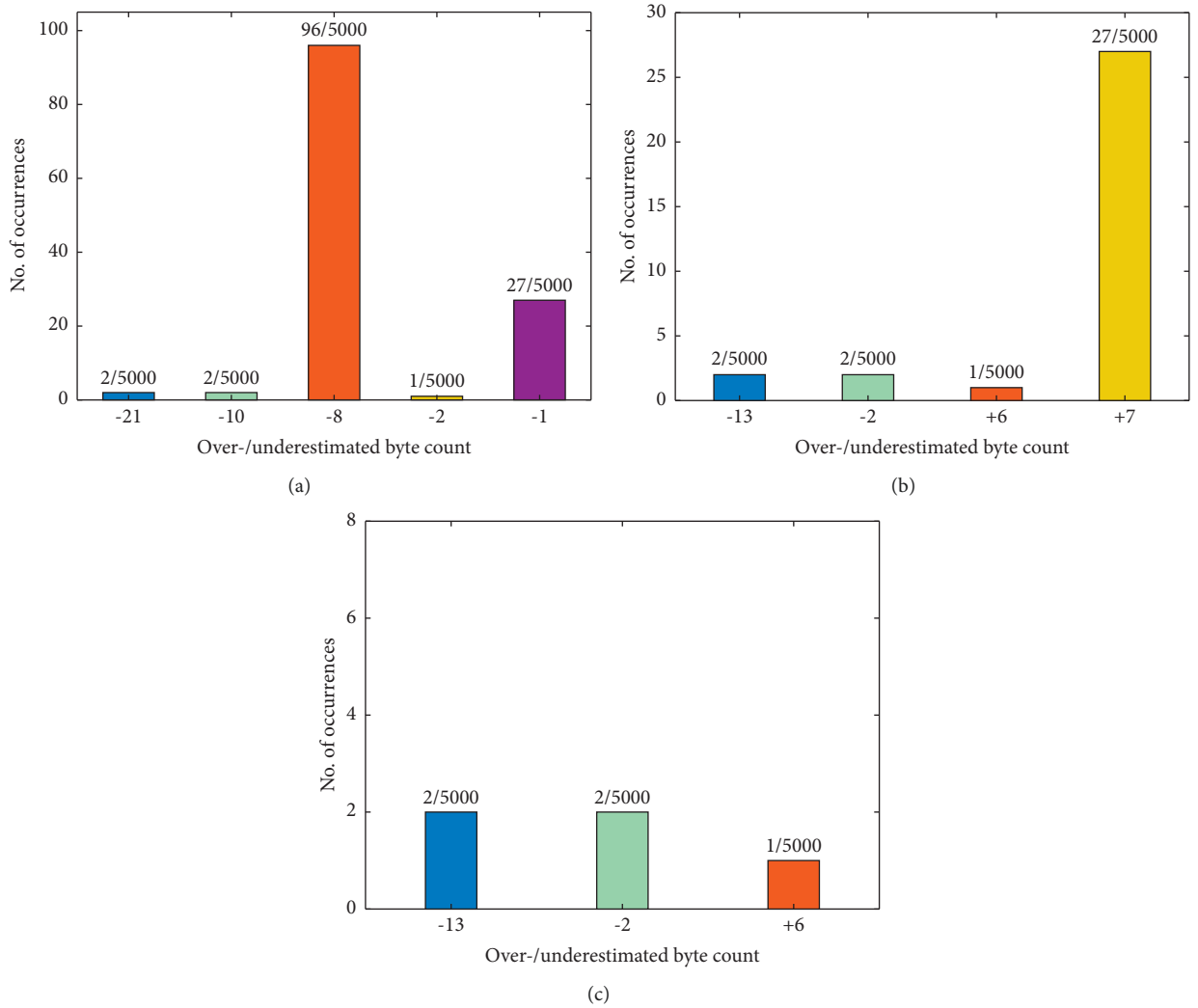

Figure 3: Analysis of the distribution of overestimations and underestimations on estimation errors.

obfuscation does not have a visible impact on the achieved performance.

For the sake of completeness, we also tested Mavis over the *clean* dataset. Results indicate that false positives are only ~0.01%.

6.2. Size Estimation of PowerShell Scripts. As shown in Table 3, Mavis was able to estimate the size of malicious scripts injected via Mode-1 with a MAPE of 5.43% in a time of ~ 3 ms, on average. The table also reports that obfuscated scripts are estimated with a higher accuracy. Yet, the time needed for size estimation is three times shorter for deobfuscated scripts than that for their obfuscated counterparts. In the case of Mode-2, Mavis achieved high values for the MAPE (i.e., 0.0011%, on average) at the price of a limited time for performing the estimation (i.e., 6.72 ms, on average). Additionally, the MAPE results are stable across groups and do not depend on the size of the image. Finally, the time needed to perform size estimation for obfuscated scripts is two times higher than that for their deobfuscated counterparts. Therefore, Mavis can be considered an efficient tool for estimating the size of malicious PowerShell scripts hidden via the Invoke-PSImage technique.

However, as mentioned in Section 4.3, the algorithm introduced for size estimation purposes could exhibit some limitations, which may lead to both under-/overestimations.

For example, let us consider the results for the 5,000 digital images with 256×256 resolution, where deobfuscated scripts have been embedded. Figure 3(a) illustrates the results of over-/underestimation. As shown, the real size of the script has been, in some cases, underestimated mostly by 1 or 8 bytes, but there are also some underestimations by -2, -10, or -21 bytes. We point out that, even in the presence of estimation errors, the size of ~ 97% scripts has been correctly identified.

To further improve the accuracy of the estimation process, we can explicitly consider the aforementioned biases. Specifically, if the initial estimation of the size is recognized as incorrect (e.g., scripts can be extracted in a sandbox and properly measured), we increase its size by 1 byte and check the outcome. If the size is still erroneous, then we increment the size by 7 bytes. From the computational perspective, this is

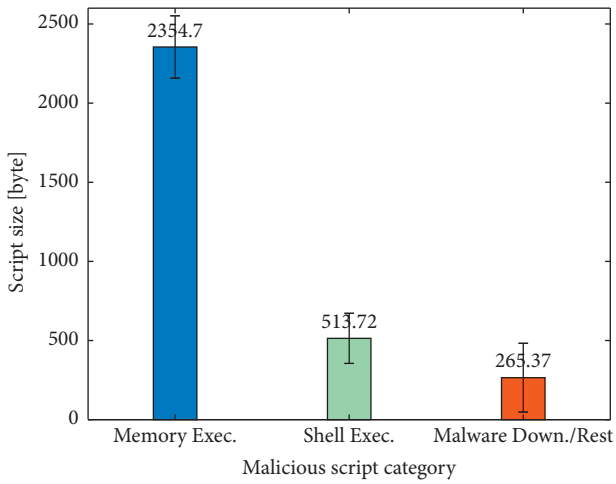

Figure 4: Script size and functionality class.

a negligible additional burden but allows improving performances to 99.8%. Moreover, reducing estimation errors also accounts for extracting the hidden script in a correct and complete manner. In Figure 3(a), it is visible that 96 scripts were underestimated by 8 bytes and 27 scripts were underestimated by 1 byte. By the exploitation of the 8-byte bias, as described above, we can decrease the number of inaccurate estimations from 128 to 32 but with an increasing overestimation of 7 bytes in 27 scripts (Figure 3(b)). Using this knowledge, we can further decrease the number of incorrectly estimated scripts to just 5 (Figure 3(c)).

Finally, we point out that the time needed to determine the size of the script can be reduced by using specific characteristics of the Invoke-PSImage injection process. Since the data is equally distributed over the targeted colour channels (i.e., every channel carries the half of a byte), it is only required to investigate one of the two channels to determine the size of the script.

6.3. Inferring Script Functionality via Its Size. To answer this research question, the first step was to remove the duplicated deobfuscated scripts from the dataset available in [21], leading to 2,355 unique scripts. Then, we determined the mean size of the previously defined three categories of scripts' functionality, that is, *Memory Execution*, *Shell Execution*, and *Malware Download/Rest*.

Figure 4 illustrates the obtained results. It is visible that the size of the scripts in the *Memory Execution* category is $\sim$ 4.5 times larger than the *Shell Execution* group. The main reason behind this is that the scripts belonging to the *Memory Execution* class contain malicious code in a binary form, which is injected directly into memory. This is a crucial characteristic that contributes to the resulting size. Since the category *Malware Download/Rest* mainly includes scripts used only to download further malicious components from the C&C server and execute them, their average size is $\sim$ 8.8 times smaller than that of those belonging to the *Memory Execution* class. We then evaluated whether we are able to assign the inspected script into the correct category just by its size. First, we

Table 4: Inferred script functionality by size.

Obfuscation method	Multiplication factor ($\overline{f_o}$)	Corr. predicted
No obfuscation	1	2,116/2,355 (89.85%)
ASCII encoding	4.19445	2,040/2,355 (85.14%)
Token-based	2.02852	1,999/2,355 (84.88%)
String-based	1.79499	1,939/2,355 (82.34%)
AbstractSyntaxTree	1.09672	2,005/2,355 (85.14%)
Overall	—	**85.77%**

experimented only with the deobfuscated scripts. Then, we also tested how many of the obfuscated scripts can be linked to the correct groups by using their mean sizes multiplied by the factor $\overline{f_o}$, depending on which obfuscation method was applied. The results for the calculation of $\overline{f_o}$ and the correctness of the size prediction are presented in Table 4.

In the case when no obfuscation was used, ~90% of the scripts were correctly classified. The ASCII encoding, token-based, and string-based obfuscation techniques performed similarly; that is, the functionality of ca. ~85% scripts was correctly inferred. Finally, a slightly worse result was achieved for the string obfuscation where ~82% of PowerShell scripts were correctly assigned. Based on these outcomes, it is evident that $\overline{f_o}$ is suitable for inferring the functionality of obfuscated scripts. Moreover, $\overline{f_o}$ varies greatly depending on the type of the used obfuscation technique. As an example, an obfuscated script using ASCII encoding is, on average, 3.82 times larger than an obfuscated script using AbstractSyntaxTree. As a consequence, the suggested approach seems to be promising, as more than 85% of the scripts can be assigned to their correct functional categories.

7. Comparison with Existing Tools

To prove the effectiveness of our approach, we compared Mavis with McAfee SAT and StegExpose. To avoid burdening the discussion, detailed results for these two tools are presented in Appendix A.

Concerning the experimental methodology, in the case of StegExpose, we used the complete datasets of steganographically modified and clean files of different resolutions as described in Section 5.1 (in total 45,000 files). However, due to the file size limitation of McAfee SAT, we narrowed the datasets to images of size 256×256 (stego and clean files), which means that only 15,000 files were subjected to evaluation.

Moreover, as the McAfee SAT tool is an online product, it was necessary to automatize the experimental campaign by developing a drag&drop control for the website. In this case, we used WebDriver for Chrome and prepared a custom Python script for sending the selected digital images to the McAfee website and then capturing the generated report.

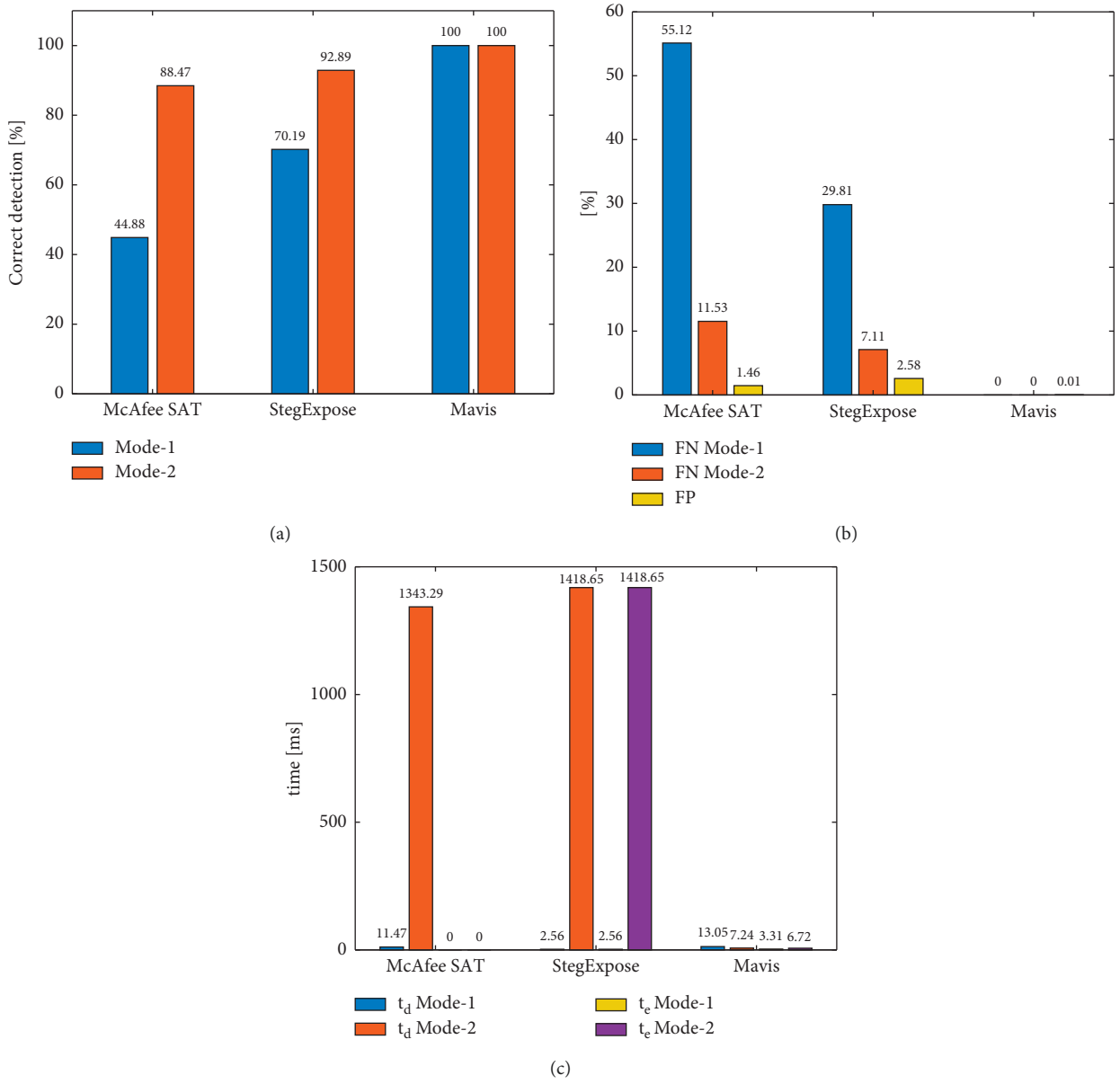

Figure 5: Comparison among the three methods.

For the case of StegExpose, we needed to modify its source code to have access to additional information, such as the time needed to perform the detection process.

For both tools, all measurements were stored in CSV files and then analyzed through ad hoc Python scripts to obtain the final averaged values. The experimental evaluation of McAfee SAT tool was performed on a machine equipped with an Intel(R) Core(TM) i5-9400H CPU @ 2.5 GHz with 8 GB RAM, while for StegExpose it was a device running Ubuntu 20.04 with an Intel(R) Core(TM) i9-9900KF CPU @ 3.60 GHz and 32 GB RAM.

Figure 5(a) illustrates the detection rates for each tool. It can be noted that Mavis outperforms the rest: the difference is especially noticeable for Mode-1 detection, for which McAfee SAT reached only $\sim$ 45% and StegExpose $\sim$ 70%, while it is 100% for Mavis. For Mode-2, the difference is smaller but still accounts for $\sim$ 10%. The FP and FN comparison is presented in Figure 5(b). It can be concluded that, again, Mavis achieved significantly better results (FN = 0% for both modes and FP = 0.01%) than the rest of the tools. McAfee SAT yielded particularly poor results, with FN rates as high as $\sim$ 55%. StegExpose is in this aspect much better with 11.53% for Mode-1 and 7.11% for Mode-2, but again Mavis significantly outperformed it.

Regarding the computational performance of each method, the experimental results are showcased in Figure 5(c). As shown, the time needed to detect Mode-1 of Invoke-PSImage is smaller for the case of StegExpose (2.56 ms), and it achieves the best performance compared against the other tools (for McAfee SAT it was 11.47 ms, while for Mavis it is 13.05 ms). However, for Mode-2, both StegExpose and McAfee SAT experienced a significant

Table 5: Detection results for StegExpose in "default" mode.

Invoke-PSImage Mode-1					
Scripts	Correct detec.	FN rate	$\overline{t_d}$ (ms)	$\overline{t_e}$ (ms)	MAPE (%)
Deobfuscated	4429/4641 (95.43%)	212/4641 (4.57%)	1.56	1.56	56
Obfuscated	1649/4018 (41.04%)	2369/4018 (58.96%)	3.72	3.72	91
Overall	**70.19%**	**29.81%**	**2.56**	**2.56**	**72**
Invoke-PSImage Mode-2					
Scripts	Correct detec.	FN rate	$\overline{t_d}$ (ms)	$\overline{t_e}$ (ms)	MAPE (%)
Deobfuscated (256 × 256)	4850/5000 (97%)	150/5000 (3%)	164.25	164.25	11614
Deobfuscated (512 × 512)	4579/5000 (91.58%)	421/5000 (8.42%)	824.05	824.05	31470
Deobfuscated (1024 × 1024)	4600/5000 (92%)	400/5000 (8%)	3308.11	3308.11	125795
Obfuscated (256 × 256)	4816/5000 (96.32%)	184/5000 (3.68%)	183.74	183.74	4380
Obfuscated (512 × 512)	4498/5000 (89.96%)	502/5000 (10.04%)	831.91	831.91	11848
Obfuscated (1024 × 1024)	4525/5000 (90.05%)	475/5000 (9.5%)	3199.81	3199.81	48231
Overall	**92.89%**	**7.11%**	**1418.65**	**1418.65**	**38890**
Scripts	Correct detec.	FP rate	$\overline{t_d}$ (ms)	$\overline{t_e}$ (ms)	MAPE
Clean (256 × 256)	4885/5000 (97.7%)	115/5000 (2.3%)	166.4	N/A	N/A
Clean (512 × 512)	4897/5000 (97.94%)	103/5000 (2.06%)	831.82	N/A	N/A
Clean (1024 × 1024)	4831/5000 (96.62%)	169/5000 (3.38%)	3261.14	N/A	N/A
Overall	**97.42%**	**2.58%**	**1419.79**	**N/A**	**N/A**

Table 6: Detection results for StegExpose in "fast" mode.

Invoke-PSImage Mode-1					
Scripts	Correct detec.	FN rate	$\overline{t_d}$ (ms)	$\overline{t_e}$ (ms)	MAPE (%)
Deobfuscated	3860/4641 (83.17%)	781/4641 (16.83%)	1.37	1.37	59
Obfuscated	1017/4018 (25.31%)	3001/4018 (74.69%)	0.44	0.44	93
Overall	**56.32%**	**43.68%**	**0.94**	**0.94**	**75**
Invoke-PSImage Mode-2					
Scripts	Correct detec.	FN rate	$\overline{t_d}$ (ms)	$\overline{t_e}$ (ms)	MAPE (%)
Deobfuscated (256 × 256)	3649/5000 (72.98%)	1351/5000 (27.02%)	122	122	10163
Deobfuscated (512 × 512)	2852/5000 (57.04%)	2148/5000 (42.96%)	480.27	480.27	25965
Deobfuscated (1024 × 1024)	2902/5000 (58.04%)	2098/5000 (41.96%)	1606.09	1606.09	104000
Obfuscated (256 × 256)	3694/5000 (73.88%)	1306/5000 (26.12%)	118.33	118.33	3841
Obfuscated (512 × 512)	2859/5000 (57.18%)	2141/5000 (42.82%)	364.44	364.44	9719
Obfuscated (1024 × 1024)	2792/5000 (55.84%)	2208/5000 (44.16%)	1694.7	1694.7	39816
Overall	**62.49%**	**37.51%**	**730.97**	**730.97**	**32251**
Scripts	Correct detec.	FP rate	$\overline{t_d}$ (ms)	$\overline{t_e}$ (ms)	MAPE (%)
Clean (256 × 256)	4905/5000 (98.1%)	95/5000 (1.9%)	6.32	N/A	N/A
Clean (512 × 512)	4966/5000 (99.32%)	34/5000 (0.68%)	16.44	N/A	N/A
Clean (1024 × 1024)	4912/5000 (98.24%)	88/5000 (1.76%)	140.67	N/A	N/A
Overall	**98.55%**	**1.45%**	**54.48**	**N/A**	**N/A**

Table 7: Detection results for McAfee SAT (only files of size 256 × 256 can be tested).

Invoke-PSImage Mode-1					
Scripts	Correct detec.	FN rate	$\overline{t_d}$ (ms)	Conf. level	Score
Deobfuscated	3064/4641 (66.02%)	1577/4641 (33.98%)	5.59	Low: 0, medium: 2055, high: 2586	544.07
Obfuscated	822/4018 (20.46%)	3196/4018 (79.54%)	17.35	Low: 0, medium: 3892, high: 126	100.83
Overall	**44.88%**	**55.12%**	**11.47**	**N/A**	**322.45**
Invoke-PSImage Mode-2					
Scripts	Correct detec.	FN rate	$\overline{t_d}$ (ms)	Conf. level	Score
Deobfuscated	4482/5000 (89.64%)	518/5000 (10.36%)	1343.29	Low: 0, medium: 2476, high: 2524	426.87
Obfuscated	4365/5000 (87.3%)	635/5000 (12.7%)	1337.54	Low: 0, medium: 2476, high: 2524	417.26
Overall	**88.47%**	**11.53%**	**1340.41**	**N/A**	**422.06**
Clean	4927/5000 (98.54%)	3/5000 (1.46%)	1097.37	Low: 0, medium: 4991, high: 9	32.03

decrease in processing performance with ∼ 1400 ms, while Mavis was slightly faster on average than for Mode-1, that is, 7.24 ms on average. A similar effect becomes apparent on the comparison of the time needed by the size estimation algorithm (note: McAfee SAT has no such feature). For Mode-1, both StegExpose and Mavis perform similarly; that is, they need ∼ 3 ms to complete the process. However, in Mode-2, the result yielded by StegExpose rapidly decays to ∼ 1400 ms, while it remains approximately stable for Mavis (∼ 7 ms). Finally, it must be noted that the malicious script size estimation algorithm in Mavis is significantly more precise than the one in StegExpose. For estimating the size of scripts embedded using Mode-1 of Invoke-PSImage, the MAPE value is 72% for StegExpose, while it is only 5.43% for Mavis. In Mode-2, the difference is even more remarkable as the MAPE for StegExpose skyrocketed to 38890%, and in the case of Mavis it improved to 0.0011%.

8. Conclusion and Future Work

In this work, we investigated threats leveraging the Invoke-PSImage tool to embed malicious PowerShell scripts into innocent-looking digital images. To cope with such an emerging class of attacks, we proposed a tool named Mavis. Obtained experimental results indicated the effectiveness of the proposed approach as well as its efficiency. Thus, Mavis could be considered as a valid framework to protect realistic, production-quality environments, especially if characterized by nonnegligible volumes of traffic.

Future work will aim at carrying out large-scale experiments by deploying Mavis in large-scale telecom/ISP networks. Apart from further evaluating the behavior of the tool, this will also allow us to quantify the real risk related to threats exploiting the Invoke-PSImage technique in the wild. Moreover, part of our ongoing research is devoted to further analyzing the relationship between script size, functionality, and different obfuscation techniques. In particular, we will aim at investigating if our solution can be applied to combat multilayered obfuscation approaches.

Appendix

Experimental Evaluation of the Existing Tools

In this appendix, we present the detailed detection results for the steganalysis tools considered in this work, that is, StegExpose and McAfee SAT, to show how they perform when trying to detect images modified by Invoke-PSImage.

A. StegExpose Results. Tables 5 and 6 report the measurements obtained by testing StegExpose against the complete dataset in both "default" and "fast" modes with a threshold = 0.2. The threshold value is set as suggested by the authors of the StegExpose tool [28], as it generally allows achieving the best trade-off between false positive and true positive rates.

Based on these results, it is noticeable but unremarkable that the "fast" mode performs better in terms of the time needed for the detection phase and estimation phase, as expected (e.g., $\overline{t_d}$ = 2.56 ms in the case of default mode and $\overline{t_d}$ = 0.94 ms in the case of fast mode). This is a consequence of the fact that when using the "fast" mode, StegExpose tries to skip several of the four algorithms. This leads to the lower precision. In fact, the "default" mode performs better; for example, it achieves 70.19% correct detection for images, where Mode-1 was applied, while the "fast" mode reaches only 56.32%. On the other hand, when dealing with clean images, the "fast" mode can reach a higher percentage of correctly classified images, that is, 98.55% versus 97.42% in almost 1/26 of time.

The MAPE value is not significantly different for both modes. This value declines with increasing image size. Finally, the results obtained for the obfuscated images are, in general, better than those for the deobfuscated images, especially in the case of the MAPE values.

B. McAfee SAT Results. Table 7 presents the results collected by testing the steganographic tool from McAfee. The tool is not able to process images with a resolution larger than 1024×768 or with a size greater than 1MB; thus the table contains the results obtained by testing only images of size 256×256. Moreover, the SAT tool does not provide any information about the injected script (e.g., its size); thus $\overline{t_e}$ and MAPE columns are omitted, and, therefore, comparisons for these two aspects are not possible in this case.

Compared to StegExpose (in both modes), McAfee SAT performs clearly worse for images generated with Mode-1. In fact, the McAfee SAT tool achieves an overall rate of correctly detected images of 44.88% (and a false negative rate of 55.12%) versus better results from StegExpose (for both modes). Concerning the results obtained by testing images of size 256×256 generated by Invoke-PSImage, McAfee SAT performs better than StegExpose in the fast mode (comp. Table 6) but worse than when the default mode is used (comp. Table 5).

Data Availability

The data used in this paper as well as the proposed Mavis tool are available at the repository: https://github.com/s3venup/Mavis.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work has been supported by the SIMARGL Project: Secure Intelligent Methods for Advanced RecoGnition of Malware and Stegomalware, with the support of the European Commission and the Horizon 2020 Program, under Grant Agreement no. 833042.

References

[1] McAfee, 2020, https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-nov-2020.pdf.

[2] D. Hendler, S. Kels, and A. Rubin, "Detecting malicious powershell commands using deep neural networks," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pp. 187–197, Republic of Korea, October 2018.

[3] L. Caviglione, M. Choras, I. Corona et al., "Tight arms race: Overview of current malware threats and trends in their detection," *IEEE Access*, vol. 9, pp. 5371–5396, 2021.

[4] B. Adams, Invoke-psimage tool, 2017.

[5] S. Ryan and J. Saavedra-Morales, *Malicious Document Targets Pyeongchang Olympics*, 2018, https://github.com/peewpw/Invoke-PSImage.

[6] S. Selvaraj, *Is your weblogic and websphere is safe from greystars ransomware*, 2018, https://medium.com/@sheyamselvaraj/is-your-weblogic-and-websphere-is-safe-from-greystars-ransomware-daad44166f0.

[7] Y. Ursnif, "Long live the steganography!," 2019, https://yoroi.company/research/ursnif-long-live-the-steganography/.

[8] I. Kwiatkowski, P. Delcher, and F. Aime, *Iam the king and the slothfulmedia malware family*, 2020, https://securelist.com/iamtheking-and-the-slothfulmedia-malware-family/99000/.

[9] *Check Point*, Bandook: Signed & delivered, 2020.

[10] W. Mazurczyk and L. Caviglione, "Information hiding as a challenge for malware detection," *IEEE Security & Privacy*, vol. 13, no. 2, pp. 89–93, 2015.

[11] W. Mazurczyk and L. Caviglione, "Steganography in modern smartphones and mitigation techniques," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 334–357, 2015.

[12] J. Fridrich, *Steganography in Digital Media: Principles, Algorithms, and Applications*, Cambridge University Press, Cambridge, UK, 2009.

[13] E. Zielińska, W. Mazurczyk, and K. Szczypiorski, "Trends in steganography," *Communications of the ACM*, vol. 57, no. 3, pp. 86–95, 2014.

[14] H. Patel and P. Dave, "Steganography technique based on dct coefficients," *International Journal of Engineering Research in Africa*, vol. 2, no. 1, pp. 713–717, 2012.

[15] W. Su, J. Ni, X. Hu, and J. Fridrich, "Image steganography with symmetric embedding using Gaussian Markov random field model," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 31, no. 3, pp. 1001–1015, 2021.

[16] N. Provos and H. Peter, "Detecting steganographic content on the internet," *Technical report, Center for Information Technology Integration*, 2001.

[17] S. Dumitrescu, X. Wu, and Z. Wang, "Detection of lsb steganography via sample pair analysis," in *Information Hiding*, F. A. P. Petitcolas, Ed., Springer, Berlin, Heidelberg, pp. 355–372, 2003.

[18] J. Fridrich, M. Goljan, and R. Du, *Reliable Detection of Lsb Steganography in Color and Grayscale Images*, Association for Computing Machinery, New York, NY, USA, 2001.

[19] A. Westfeld and A. Pfitzmann, "Attacks on steganographic systems," in *Information Hiding*, A. Pfitzmann, Ed., Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 61–76, 2000.

[20] S. Dumitrescu, X. Wu, and N. Memon, "On steganalysis of random lsb embedding in continuous-tone images," in *Proceedings of the International Conference on Image Processing (ICIP'02)*, vol. 3, Rochester, NY, USA, September 2002.

[21] D. Ugarte, D. Maiorca, F. Cara, and G. Giacinto, "Powerdrive: accurate de-obfuscation and analysis of powershell malware," in *Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment-16th International Conference*, Gothenburg, Sweden, June 2019.

[22] D. Bohannon, *Invoke-obfuscation v1.8*, 2018, https://github.com/danielbohannon/Invoke-Obfuscation.

[23] J. Raber and E. Laspe, "Deobfuscator: an automated approach to the identification and removal of code obfuscation," in *Proceedings of the 14th Working Conference on Reverse Engineering*, pp. 275-276, Vancouver, BC, Canada, October 2007.

[24] M. Park, G. You, S. je Cho, M. Park, and S. Han, "A framework for identifying obfuscation techniques applied to android apps using machine learning," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, vol. 10, no. 4, pp. 22–30, 2019.

[25] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet, and F. Mercaldo, "Detection of obfuscation techniques in android applications," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, Auguest 2018.

[26] D. Bohannon and L. Holmes: Revoke-obfuscation v1.0, 2020, https://github.com/danielbohannon/Revoke-Obfuscation.

[27] Z. Tang, K. Kuang, L. Wang et al., "A semantic-based approach for automatic binary code de-obfuscation," in *Proceedings of the 2017 IEEE Trustcom/BigDataSE/ICESS*, pp. 261–268, Sydney, NSW, Australia, Auguest 2017.

[28] B. Boehm, *Stegexpose-a Tool for Detecting Lsb Steganography*, arXiv preprint arXiv:1410.6656, 2014.