

# A Novel Behavioural Screenlogger Detection System

Hugo Sbai<sup>1</sup>[0000–0002–6411–1864], Jassim Happa<sup>2</sup>[0000–0002–0860–5130], and  
Michael Goldsmith<sup>1</sup>[0000–0001–7808–0600]

<sup>1</sup> University of Oxford, Oxford OX1 3QD, UK

<sup>2</sup> University of London, Royal Holloway, London WC1B 3RF, UK

**Abstract.** Among the various types of spyware, screenloggers are distinguished by their ability to capture screenshots. This gives them considerable nuisance capacity, giving rise to theft of sensitive data or, failing that, to serious invasions of the privacy of users. Several examples of attacks relying on this screen capture feature have been documented in recent years. Moreover, on desktop environments, taking screenshots is a legitimate functionality used by many benign applications, which makes screenlogging activities particularly stealthy. However, existing malware detection approaches are not adapted to screenlogger detection due to the composition of their datasets and the way samples are executed. In this paper, we propose the first dynamic detection approach based on a dataset of screenloggers and legitimate screenshot-taking applications (built in a previous work), with a particular care given to the screenshot functionality during samples execution. We also propose a tailored detection approach based on novel features specific to screenloggers. This last approach yields better results than an approach using traditional API call and network features trained on the same dataset (minimum increase of 3.108% in accuracy).

**Keywords:** Screenloggers · Screenshots · Malware detection.

## 1 Introduction

### 1.1 Context and Motivation

Spyware can be defined as software that gathers information about a person or organisation without their consent or knowledge and sends it to another entity [28]. The software is designed for secrecy and durability. A long-term connection to the victim’s machine is established by the adversary, and once spyware is installed on the victim’s device, it aims to steal information unnoticed.

Spyware is usually organised in multiple modules, each performing one or more malicious activities, with the ability to use them according to the attacker’s purpose [3]. Typical spyware modules include keystroke logging, screen logging, URL monitoring, turning on the microphone or camera, intercepting sensitive documents and exfiltrating them and collecting location information [28].

Among the aforementioned spyware modules, screenloggers have one of the most dangerous functionalities in today’s spyware as they greatly contribute to hackers achieving their goals.

Screenlogger users can be divided into two categories: financially motivated actors and state-sponsored attackers. The first category targets important industrial companies (e.g., BRONZE BUTLER [11]), online banking users (e.g., RTM [25], FIN7 [10], Svpeng [11]), and even banks themselves (e.g., Carbanak [25], Silence [26]). The second category, which is even more problematic, targets critical infrastructure globally. For instance, the malware TinyZbot [8], a variant of Zeus, has targeted critical infrastructure in more than 16 countries. More precisely, the targets can be democratic institutions; for instance, XAgent targeted the US Democratic Congressional Campaign Committee and the Democratic National Committee [19]. In Europe, Regin [26], took screenshots for at least six years in the IT network of the EU headquarters. Diplomatic agencies have also been compromised, for example by the North Korean malware ScarCruft [16]. US defence contractors have also been hit by screenloggers such as Iron Tiger.

Screenloggers have the advantage of being able to capture any information displayed on the screen, offering a large set of possibilities for the attacker compared to other spyware functionalities. Moreover, malware authors are inventive when maliciously using screen captures. Indeed, screen captures have a wide range of purposes. Some malware, such as Cannon and Zebrocy, only take one screenshot during their entire execution as reconnaissance to see if the victim is worth infecting [5] [20]. Others hide what is happening on the victim’s screen by displaying a screenshot of their current desktop (FinFisher [20], [30]) or take numerous screen captures for a close monitoring of the victim’s activity. This allows spyware hackers to steal sensitive intellectual property data (BRONZE BUTLER [4]), banking credentials (RTM [25], FIN7 [10], XAgent [19]), or monitor day-to-day activity of banking clerks to understand the banks’ internal mechanisms (Carbanak [9], Silence [26]).

The screenshot capability is sometimes the unique functionality used in some phases of an attack to observe while remaining stealthy. For instance, in the Carbanak attack targeting banking employees [28], attackers used the screengrabs to create a video recording of daily activity on employees’ computers. The hackers amassed knowledge of internal processes before stealing money by impersonating legitimate local users during the next phase of the attack.

These examples show that the screenshot functionality is widely used today in modern malware programs and can be particularly stealthy, enabling powerful attacks. Even in the case where no specific attack is performed, the simple fact of monitoring and observing all the victims’ activity on their device is a serious invasion of privacy. Moreover, screenshots are likely to contain personally identifiable information [24].

What makes the screenlogger threat even more problematic and stealthy is that, on desktop environments, the screenshot functionality is legitimate and, as such, is used by many benign applications (e.g. screen sharing for work or troubleshooting, saving important information, creating figures, monitoring em-

ployees). The necessity of capturing the screen has for instance increased with telework, even on sensitive machines. Teleworkers, including bank employees or lawyers, may need to control their office computer remotely from home, or to share their screens during online meetings. Therefore, countering the screenlogger threat cannot 'simply' be done by disabling the screenshot functionality on sensitive machines. Other natural approaches such as white-listing are prone to the ever more sophisticated strategies malware authors can deploy to inject malicious code into legitimate processes or bypass the user's consent.

Paradoxically, no work in the literature proposes a detection methodology adapted to the specifics of screenshot-taking malware. This is what we aim to do in this paper.

## 1.2 Contributions

More precisely, the contributions brought by this paper are:

- Training and testing of the detection model on a dataset composed exclusively of screenshot-taking malware and legitimate applications representative of the behaviours found in the wild: this allowed to identify the most effective existing detection features for screenlogger detection.
- Creation of features adapted to the screenlogging behaviour using novel techniques : instead of using hundreds of features and trusting a machine learning model to select the most discriminating ones, we propose to use new features that reflect a specific behaviour. The advantage is that malware authors will not be able to misguide the detection system without changing their core functionality. Indeed, existing detection models are prone to overfitting and can easily be misguided by malware authors by acting on features unrelated to the malicious functionalities of their programs.
- Samples execution methodology: in the malware detection field, it is common to automatically run thousands of malware samples in a controlled environment to collect features without interacting with the samples. Such an approach would unfortunately not work for screenloggers, as their malicious functionality needs to be triggered at run time through interaction with the malware program. To collect our features, we paid a particular care to ensure that each malicious or legitimate program worked as intended during its execution (was taking screenshots).

## 1.3 Paper outline

In this paper, we start by discussing the relevant literature (Section 2) and precisising the scope of our work through a system model (Section 3) and a threat model (Section 4). After that, we outline our experimental setup for screenlogger detection (Section 5). Then, we present a detection model based on state of the art features (Section 6) and a novel detection model including features specific to the screenlogging behaviour (Section 7). Finally, the performances of the two models are compared and discussed (Section 8).

## 2 Literature review

Several types of malware detection models can be found in the literature.

Signature-based methods give good results for known malware programs with a low false-positive rate. However, they are vulnerable to obfuscation techniques that are more and more used by modern malware [31]. Moreover, signature-based methods cannot detect malware integrating polymorphism and metamorphism mechanisms because the signature of the malware changes every time a machine is infected.

The drawback of anomaly-based detection techniques for detecting malware in computer systems is that those systems contain and execute many processes having many possible behaviours. This makes it difficult to define a "normal" behaviour and can result in a high false-positive rate.

Static behaviour-based detection fails to overcome obfuscation techniques [29]. Malware designers can use those techniques to disturb the analysis process and hide the malicious behaviour of the program.

Dynamic behaviour-based detection can overcome obfuscation techniques, as it analyses the runtime behaviour of the malware. Therefore, this technique is widely used in recent malware detection works, which consider diverse types of dynamic features such as API calls [13], [12], [22] and network traffic [18], [6], [23]. However, it was shown that the performance of a dynamic behaviour-based detection model greatly depends on the dataset it was trained on. Indeed, the features selected as the most discriminative vary according to the malware type [13], [12]. Hence, if the dataset does not contain any screenshot-taking malware or very few, the behaviours related to screenlogging would not be taken into account.

To the best of our knowledge, only one malware detection work explicitly mentions screenloggers [15]. This work focused on detecting spyware and, more specifically, keyloggers, screen recorders, and blockers. The authors proposed a dynamic behavioural analysis through the hooking of kernel-level routines. More precisely, the presented method is designed to detect screenloggers under the Windows operating system. To this end, it hooks the `GetWindowDC` and `BitBlt` functions. Then, to classify a screenshot-taking program as spyware or benign, they used a decision tree considering the following features: frequency of repetition, uniqueness of the applicant process, state of the applicant process (hidden or not) and values of parameters in the system calls. The results showed that the proposed method could detect screenloggers with an accuracy of 92% and an error rate of 7%.

However, this method suffers from several weaknesses. Relying exclusively on API calls may not be sufficient to distinguish screenloggers from legitimate screenshot-taking applications. Indeed, by investigating existing screenloggers, it is possible to notice that they may exhibit various behaviours, including the fact that the screenshots frequency may be different from one screenlogger to another. The frequency can be a few seconds, few minutes, or configurable by the adversary. Screenshots can also be taken irregularly at each user event. Legitimate screenshot-taking applications are also diverse. Some of them, such as

screensharing applications, need to take screenshots at a high frequency, while others like parental control make take screenshots at a lower frequency, and others like screenshots editing applications may take screenshots occasionally. Therefore, relying only on API calls may lead to high false positives and false negatives rates on an extensive dataset containing different types of screenloggers as well as benign screenshot-taking applications. It is not mentioned whether the dataset used to test the method proposed in [15] contained benign screenshot-taking programs, and there is no information about the nature and diversity of screenloggers. Moreover, the authors perform the hooking on only two functions, namely GetWindowsDC and BitBlt, whereas there are other ways of taking screenshots. For example, it is possible to use the GetDC function instead of GetWindowsDC in order to obtain the device context.

### 3 System Model

The targeted systems are desktop environments. The main reason why our work focuses on computer operating systems is that the screenshot functionality is a legitimate functionality offered to any application. In contrast, on smartphones, the principle is that apps cannot take screenshots of other apps, and the only way to accomplish this is to exploit specific vulnerabilities or to divert some libraries. However, many limitations exist for these techniques, such as permission required from the user at the beginning of each session, or a recording icon displayed in the notification bar. In sum, the architecture designs of mobile systems and computer systems are fundamentally different, which may lead to different solutions.

Targeted victims may be any individual or organisation, ranging from typical laptop users to small companies or powerful institutions. The victims are not particularly security aware, which implies they are not necessarily cognizant of the existing threats and will not install a specific protection against screenshots, such as a specific viewer to open documents in a secure environment, which prevents screenshots.

## 4 Threat Model

### 4.1 General Description

Our threat model is composed of a victim, an attacker and spyware with a screenshot functionality.

In this model, a screenshot is defined as a reproduction in an image format of what is displayed on the screen, even if all pixels may not be visible. Screenloggers must rely on a functionality offered by the operating system to perform their attack.

The adversary’s goals are diverse. They can range from general activity monitoring, which requires to see the whole screen, to sensitive data theft, which can be limited to some areas of the screen.

## 4.2 Operating process

Attackers may infect a system using common methods such as trojans, social engineering or through a malicious insider. The adversary has no physical access to the victim’s device (except in the case of a malicious insider). They have no knowledge about the system and tools installed on it before infection. We also assume they have not compromised the victim’s device at a kernel level. Apart from that, the attacker can use any technique to evade detection, including hiding by injecting api calls into system or legitimate processes, dividing its tasks between multiple processes, making the API calls out of sequence, spaced out in time, or interleaved with other API calls.

To reach their objective, attackers take screenshots of the victim’s device. The data may be either (1) extracted automatically using OCR tools inside the victim’s device locally, then sent to the attacker’s server using the victim’s network interface or (2) extracted, also using OCR tools, on the attacker’s server after screenshots have been transferred from the victim’s machine to the attacker’s as compressed image files. The screenshots can also be analysed manually by the attacker. Moreover, the screenshots may be taken and sent at regular or irregular rates.

## 5 Experimental setup

### 5.1 Malicious and benign datasets

In a previous work [27], we constructed the first dataset dedicated to malicious and legitimate screenshot-taking applications.

To ensure that this dataset was as representative and complete as possible, we included all the behaviours mentioned in the security reports referenced on the MITRE ATT&CK screen capture page [21].

Regarding legitimate applications, we collected samples of five categories of legitimate screenshot-taking applications: screen sharing, remote control, children/employee monitoring, screencasting and screenshot editing. Each of these categories exhibits different screenshot-taking and sending behaviours.

Our dataset contains 106 malicious samples and 87 legitimate samples. Although these numbers might seem low compared to the thousands of samples traditionally used in general malware detection works, they correspond to the number of samples used in detection works that target specific categories of malware [17]. Moreover, these numbers are explained by the particular care that must be given for each sample at runtime, as presented in the following section.

### 5.2 Experimental framework

In a previous work [27], we realised that none of the screenlogger samples found on available malware datasets (e.g. VirusShare, VirusTotal) were taking screenshots at runtime. This was mainly due to the specificities of screenshot triggering

(need to receive a command from the malicious server, need to open certain applications,...). This means that, even if generalist malware detection works might have been tested on screenlogger samples (among thousands of other malware samples), the screenshot functionality was probably not observed because no attention was paid to screenshot-triggering.

Therefore, the samples we selected for our malicious dataset had to include both the client and server parts.

Our malicious samples were run in two Windows 10 virtual machines to allow the client and server parts to communicate and trigger the screenshot functionality. Legitimate applications were also run in two machines when it was required for screenshot-triggering.

During their execution, the behaviour of malicious and benign samples was monitored using API Monitor and Wireshark.

To implement and test our detection models, we used the Weka framework, which is a collection of machine learning algorithms for solving real-world data mining problems [1].

More precisely, we used it to process the run-time analysis reports, select the best detection features, select the classification algorithms, train and test the models, and visualise the detection results.

## 6 Basic detection approach

To prove the effectiveness of our novel detection model, it was first necessary to construct a model based on features from the malware detection literature. These features were extracted (Section 6.1) and transformed (Section 6.2). Then a machine learning model was trained and tested (Section 6.3) to select the most effective features (Section 6.4).

### 6.1 Feature extraction

When running the samples from our malicious and benign datasets in a controlled environment, we collected reports on two aspects of their behaviours: API calls (API Monitor reports) and network activity (Wireshark reports).

**API calls** This category of features was extracted from the reports produced by API Monitor.

The first feature we used consisted in counting the number of occurrences of each API call. For each malicious and benign API call report, the numbers of occurrences of the API calls it contains was extracted in a .csv file.

In the literature, we found that malware programs try to dissimulate their malicious functionality by introducing benign API calls to their API call sequences. A popular way of performing malware detection using API calls is to use the number of occurrences of API call sequences rather than API calls taken alone. For this, the concept of N-grams is used. N-grams are sequences of N API calls made successively by the studied program.

As a result, we also extracted features based on the number of occurrences of 2-gram and 3-gram API calls sequences. The values of  $N$  were intentionally kept low for two reasons: (1) the number of features increases exponentially with  $N$ , and (2) the detection performance often decreases as  $N$  increases.

**Network traffic** Using the .pcap files produced by Wireshark and the Argus tool to isolate network flows [2], we extracted 47 network features found in the literature. These features belong to four categories:

- Behaviour-based features ([4], [6]): these features represent specific flow behaviours. For instance, they include the source and destination IP addresses.
- Byte-based features ([18]): these features use byte counts. For instance, they include the average number of bytes from source to destination.
- Packet-based features([18], [4], [6]): these features are based on packet statistics. For instance, they include the number of small packets (length  $\leq 400$  bytes) exchanged and the number of packets per second.
- Time-based features([18], [4], [6]): these features depend on time. For instance, they include the minimum time a flow was idle before becoming active.

## 6.2 Detection algorithm

Our detection model uses the Random Forest algorithm ([7]). This algorithm trains several decision trees and uses a majority vote to classify observations. Each decision tree is trained on a random subset of the training dataset using a random subset of features.

The main shortcoming of decision trees is that they are highly dependant on the order in which features are used to split the dataset. Random Forest addresses this issue by using multiple trees using different features.

We tested several parameters to optimise the performances of the model:

- Number of trees in the forest (by default 100).
- Number of randomly selected features for each tree.
- Maximum depth of the trees (by default unlimited).
- Minimum number of instance per leaf (by default 1 but can be raised to prevent overfitting).

## 6.3 Model training and testing

To train and test our model, we used the  $k$ -fold cross-validation method (with  $k = 10$ ). This method consists in dividing our dataset into  $k$  blocks of the same size. The blocks all have the same proportions of malware and legitimate applications. For each block, we train the model on the  $k - 1$  other blocks and test it on the current block. The final detection results are obtained by adding the results of each block.

Using cross-validation, we trained and tested our model using first API call features only, then network features only, and, finally, using both categories of features.



#### 6.4 Feature selection

Due to the high number of features used, to avoid overfitting, it was necessary to select the most useful ones. A feature is useful if it is informative enough for our classification task, that is, if it enables to effectively distinguish between malicious and benign behaviours.

For this task we used the Recursive Feature Elimination method ([14]). Given a number of features to select, this method iteratively trains our Random Forest model using cross-validation and removes the least important features at each iteration. The importance of a feature is given by the average of its Gini impurity score for each decision tree in which it is used.

The Gini impurity of a feature that splits the samples at a node of a decision tree reflects how 'pure' are the subsets produced by the split. In our case, a subset is purer if it contains mostly screenloggers or mostly legitimate screenshot-taking applications. For instance, a subset containing 75% malware and 25% legitimate applications is purer than a subset that contains 50% malware and 50% legitimate applications. The impurity of a subset is given by the formula:

$$p(\text{malware}) * (1 - p(\text{malware})) + p(\text{legitimate}) * (1 - p(\text{legitimate}))$$

That is:  $2 * p(\text{malware}) * p(\text{legitimate})$

The Gini impurity of a feature is the weighted average of the impurity scores of the subset it produces. The weights are computed using the number of samples contained in each subset.

When the features are numerical values (which is our case), instead of computing the impurity of the subsets produced by each single value, intervals are used. More precisely, the Gini impurity of the feature is obtained through the following steps:

- Step 1: The values of the feature are sorted.
- Step 2: The averages of each adjacent values are computed.
- Step 3: For each average value from Step 2, the Gini impurity of the feature if the samples were split using this value is computed.
- Step 4: The Gini impurity of the feature is the minimum among the Gini impurities from Step 3.

### 7 Optimised detection approach

The novel detection approach we propose is based on new features specific to the screenlogger behaviour.

Thanks to the comparison made in our previous work between malicious and legitimate screenshot-taking behaviours, we were able to identify promising features for screenlogger detection. These features target specific behaviours that can allow to distinguish between screenloggers and legitimate screenshot taking.

For some of these features, we had to record the times at which screenshots were taken by the applications. To this end, we used screenshot API call sequences that we had identified in a previous work [27]. Indeed, there does not exist a single API call that can be called to take a screenshot, but rather a

succession of API calls that must be called in a given sequence, each one of them accomplishing a different task (e.g. retrieving the Device Context, creating a bitmap, copying the content of one bitmap into another). Different functions can be called at each stage of the sequence, which results in many sequences.

As the functions in the sequences take as parameters the return values of the previous functions, it is impossible for them to be called out-of-order. Moreover, as the return values are kept in memory until they are used as parameters, the screenshot is detected even if the API calls are spaced in time.

For the features where we needed this information, we wrote a script that ran through the API calls reports looking for screenshot sequences and recording their time stamps.

### 7.1 Interaction with the user

Contrary to screenloggers, a majority of legitimate screenshot-taking applications require an interaction with the user to start taking screenshots.

To extract this feature, we had to identify the API calls which result from user interaction. We found that, on Windows, some API calls involved in user interaction can be called on other applications' windows. As such, they could easily be called by a malware program pretending to interact with the user, whereas in fact, it does not even have a window.

Other API calls, mainly those involved in drawing on the window can only be called by the application that created the window. If they are called by another application, their return value is *false*. Therefore, we monitor this second category of functions and, even if they are called, we verify their return value.

### 7.2 Visibility of the screenshot-taking process

Unless they infiltrate themselves in legitimate processes, all the malicious samples of our dataset take screenshots through background processes hidden to the user. Legitimate screenshot-taking applications, apart from children/employee monitoring and some applications that create a background process for the screenshot-taking (e.g. TeamViewer), use foreground processes. Thus, the fact that the screenshots are taken by a background process increases the probability of malicious activity.

### 7.3 Image sending

A major part of legitimate screenshot-taking applications do not send screenshots over the network, contrary to our malware samples (no malware with the local OCR exploitation feature was found). However, due to the limited monitoring time (3 minutes), we cannot tell for sure that the screenshots taken by a given application will never be sent. Indeed, some malware can for instance schedule the sending of screenshots. In such a case, even if image packets are not sent during the monitoring time, it can be that these packets will be sent later.

Therefore, our 'Image sending' feature only reflects whether or not screenshots are sent during the monitoring time, and cannot be used to affirm that an application does not send the screenshots it takes. Moreover, determining whether a network packet contains an image is only possible when the packet is not encrypted.

#### 7.4 Remote command triggering

An important characteristic shared by almost all screenloggers is that their screenshot-taking activity is triggered by a command received from their C&C server.

Two kinds of screenshot-triggering commands can be distinguished: commands for continuous capture of the screen and punctual commands for a single screenshot. In the first case, only one command is received at the beginning of the screenshot session, whereas in the second case, a command is received before every screenshot event. To cover both cases, we chose to consider that the screenshot-taking activity is triggered by a command even if only one screenshot is preceded by the reception of a network packet.

We had to determine an adequate duration between the reception of the command and the screenshot. Indeed, we only consider that the screenshot was triggered by the network packet if this packet is received within a given time-window  $T$  before the screenshot api call sequence. Concretely, for each screenshot taken, we control if:

$$t(\text{screenshot}) - t(\text{lastNetworkMessage}) < T$$

Note that, to measure this feature accurately, it was necessary that the API calls and network reports be generated at the exact same time.

By analysing our samples, we found that the maximum duration between the command and the screenshot is 46 772 ms, the minimum duration is 0.0059 ms, the average duration is 83.044 ms and the median duration is 33.115 ms. We conducted experiments with these different values for  $T$ .

Even if it was not found in our dataset, we account for the case where the process receiving the command is different from the process taking the screenshots.

To the best of our knowledge, our detection model, through this feature, is the first to make a correlation between two kinds of events (reception of a command and screenshot API call sequences) for malware detection.

#### 7.5 Asymmetric traffic

One of the packet-based network feature we found in the literature is the ratio between the number of incoming packets and the number of outgoing packets. This feature fails to capture the asymmetric traffic displayed by most screenloggers as opposed to legitimate screenshot-taking applications (e.g. video call with screen sharing). Indeed, in the case of screenloggers, the asymmetry lies in the quantity of data exchanged, and not necessarily in the number of packets.

It may be that the number of incoming and outgoing packets are equal, for example in the case of punctual screenshot commands. In such a case, however, the quantity of data received from the C&C server is significantly lower than the quantity of data sent by the victim machine. Therefore, instead of measuring the ratio between the number of incoming and outgoing packets, we use the ratio between the numbers of bytes exchanged in both directions.

## 7.6 Captured area

During our study, we observed that almost all malware capture the full screen as opposed to legitimate applications which may target more specific areas of the screen depending on their purpose. As a result, we implemented a 'captured area' feature which takes three values: full screen, coordinates and target window.

We had to identify, in our screenshot API call sequences, the elements that show what area of the screen is captured. However, there is not only one way to capture a given area of the screen, but several. For instance, to capture a zone with given coordinates, one might get a cropped DC from the beginning using the GetDC function with the desired coordinates as parameters, or take the whole DC and do the cropping afterwards when copying the content of the screen in the destination bitmap using BitBlt's arguments.

Therefore, for each of the three values of the 'captured area' feature, we listed the possible API call sequences which might be used.

Note that we consider that an application capturing more than the three quarters of the screen's area captures the full screen. This is to avoid malware programs pretending that they capture a precise area when, in fact, only few pixels are removed from the whole screen.

## 7.7 Screenshot frequency

The last screenlogger-specific feature we created is the frequency of screenshots. We consider that an application takes screenshots at a given frequency if we find the same time interval between ten screenshots. Indeed, some malware programs offer to take punctual screenshot as well as continuous screen capture. Therefore, it is possible that not all the screenshots be taken at the same time interval.

Each time a screenshot API call sequence is found, we record its time stamp. Then, we subtract the timestamps of consecutive sequences and compare the intervals obtained. If more than ten intervals are found to be equal, the feature takes the value of this interval. Screenshots taken using different sequences are accounted for in this frequency calculation.

Some malware programs try to evade detection by dynamically changing the screenshot frequency using random numbers. To cover this case, we consider that the intervals are equal if they are within 15s of each other.

## 8 Results and comparison

### 8.1 Performance measurements

Malware detection is a binary classification problem with two classes: malware and legitimate application.

The measures used to assess the performances of our detection models are the following:

- True Positives(TP): Number of malware programs classified as malicious.
- False Positives (FP): Number of legitimate applications classified as malicious.
- True Negatives (TN): Number of legitimate applications classified as legitimate.
- False Negatives (FN): Number of malware programs classified as legitimate.
- Accuracy: Given by the formula  $\frac{TP+TN}{TP+TN+FP+FN}$ . Accuracy does not discriminate between false positives and false negatives.
- Precision: Given by the formula  $\frac{TP}{TP+FP}$ . Precision is inversely proportional to the number of false positives.
- Recall: Given by the formula  $\frac{TP}{TP+FN}$ . Recall is inversely proportional to the number of false negatives.
- F-score: Given by the formula  $\frac{2*Precision*Recall}{Precision+Recall}$ . Contrary to accuracy, F-score decreases more rapidly if false positives or false negatives are high (i.e. precision or recall are low).

In the case of malware detection, it is crucial that all malware programs be detected, to avoid them causing important damage. On the other hand, classifying a legitimate application as malware, even if it can be inconvenient for the user, might not be as critical. As a result, we give a particular importance to the false negatives and recall metrics.

### 8.2 Basic detection approach

Table 1 contains the results we obtained for the first detection approach using features found in the literature.

**Table 1.** Detection results for the basic approach using features from the literature

Features	Accuracy	False Negatives	False Positives	Precision	Recall	F-Measure
network	94.301%	0.038	0.080	0.936	0.962	0.949
1-gram	92.228%	0.038	0.126	0.903	0.962	0.932
2-gram	88.601%	0.104	0.126	0.896	0.896	0.896
3-gram	83.938%	0.123	0.207	0.838	0.877	0.857
(1+2)-gram +network	94.301%	0.038	0.08	0.936	0.962	0.949
1-gram + network	94.301%	0.028	0.092	0.928	0.972	0.949

We can observe that network features seem to give better results overall than API call features. Regarding API calls, using sequences of two and three calls significantly decreases the performances of the model, with more than 10% of malware classified as legitimate (vs 3.8% when individual API calls are used).

Combining network features and API call features does not improve the results compared to using network features alone.

Additionally, using Recursive Feature Elimination, we identified the most relevant API calls for screenlogger detection:

- strcpy\_s (Visual C++ Run Time Library)
- ntreleasemutant (NT Native API)
- \_isnan (Visual C++ Run Time Library)
- getobjectw (Graphics and Gaming)
- rtltimefields (NT Native API)

We also identified the most relevant state of the art network features:

- Bytes per packet
- Total number of bytes in the initial window from source to destination
- Total number of bytes in the initial window from destination to source
- Total number of bytes from source to destination
- Average number of bytes in a subflow from source to destination

### 8.3 Optimised detection approach

Table 2 contains the results we obtained for the second detection approach using the screenlogger-specific features we implemented.

**Table 2.** Detection results for the optimised approach using our specific features

Features	Accuracy	False Negatives	False Positives	Precision	Recall	F-Measure
Specific features	97.409%	0.009	0.046	0.963	0.991	0.977

We can see that the detection performance is improved on all metrics: with only 7 features, our model outperforms the first model based on hundreds of standard features. That is because our features capture specific malicious behaviours.

Moreover, a malware author would not be able to act on these features to mislead the classifier without changing the malicious functionality. Indeed, to mislead traditional classifiers based on numerous features, malware authors leverage overfitting by acting on features unrelated to the core functionality of their programs. When all the features target a specific behaviour, as in our case, this cannot be done.

## 9 Conclusion

In this paper, we built a first Random Forest detection model using only API calls and network features from the literature. This model was trained and tested using our malicious and benign datasets. Using Recursive Feature Elimination with Gini importance, we identified the most informative existing features for screenlogger detection.

Then, we built a second model including novel features adapted to the screenlogging behaviour. These features were collected using novel techniques. Particularly, we can cite:

- Using API call sequences to identify specific behaviours. Contrary to existing works which only look at API called in a direct succession using the notion of n-grams, we wrote scripts which keep track of the API calls return values and arguments to characterise some behaviours even if the calls are not made directly one after the other. Numerous different sequences involved in the screenshot-taking process were identified by analysing malware and legitimate applications. These sequences were also divided into three categories depending on the captured area.
- Making a correlations between API calls made by an application and its network activity. During their execution, the API calls and network activity of our samples were simultaneously monitored. This allowed us to extract features such as the reception of a network packet before starting the screenshot activity or the sending of taken screenshots over the network.

When adding these novel features to the detection model, the detection accuracy increased by at least 3.108%. Indeed, it is well known that a detection model based on less features is less likely to fall into overfitting. Moreover, a detection model based on features which have a logical meaning and reflect specific behaviours, is less prone to evasion techniques often used by malware authors.

More generally, our results show that, for some categories of malware, a tailored detection approach might be more effective and difficult to mislead than a generalist approach relying on a great number of seemingly meaningless features fed to a machine learning model.

In the future, we could extend our detection model to infection to allow for an earlier and more effective detection. The detection model could also be integrated into a defense-in-depth solution against screenloggers, including prevention mechanisms.

## References

1. Albert, B.: Weka 3: Machine learning software in java, <https://www.cs.waikato.ac.nz/ml/weka/>
2. Argus, O.: Argus, <https://openargus.org>
3. Bahtiyar, S.: Anatomy of targeted attacks with smart malware: Targeted attacks with smart malware. *Security and Communication Networks* **9** (02 2017). <https://doi.org/10.1002/sec.1767>

4. Beigi, E., Jazi, H., Stakhanova, N., Ghorbani, A.: Towards effective feature selection in machine learning-based botnet detection approaches. 2014 IEEE Conference on Communications and Network Security, CNS 2014 pp. 247–255 (12 2014). <https://doi.org/10.1109/CNS.2014.6997492>
5. Bogdan, B.: Six years and counting: Inside the complex zacinlo ad fraud operation, bitdefender, <https://labs.bitdefender.com/2018/06/six-years-and-counting-inside-the-complex-zacinlo-ad-fraud-operation/>
6. Boukhtouta, A., Mokhov, S., Lakhdari, N.E., Debbabi, M., Paquet, J.: Network malware classification comparison using dpi and flow packet headers. *Journal of Computer Virology and Hacking Techniques* **11**, 1–32 (07 2015). <https://doi.org/10.1007/s11416-015-0247-x>
7. Breiman, L.: Random forests. *Mach. Learn.* **45**(1), 5–32 (Oct 2001)
8. Charline, Z.: Viruses and malware: Research strikes back, <https://news.cnrs.fr/articles/viruses-and-malware-research-strikes-back>
9. Cybersecurity, N.J., Cell, C.I.: Zbot/zeus, <https://www.cyber.nj.gov/threat-center/threat-profiles/trojan-variants/zbot-zues>
10. David, E., S., Nicole, P.: Bank hackers steal millions via malware, <https://www.nytimes.com/2015/02/15/world/bank-hackers-steal-millions-via-malware.html>
11. Ecular, X., Grey, G.: Cyberespionage campaign sphinx goes mobile with anubis-spy, <https://www.trendmicro.com/enus/research/17/1/cyberespionage-campaign-sphinx-goes-mobile-anubis-spy.html>
12. Han, W., XUE, J., Wang, Y., Huang, L., Kong, Z., Mao, L.: Maldae: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics. *Computers Security* **83** (02 2019). <https://doi.org/10.1016/j.cose.2019.02.007>
13. Han, W., Xue, J., Wang, Y., Liu, Z., Kong, Z.: Malinsight: A systematic profiling based malware detection framework. *Journal of Network and Computer Applications* **125** (11 2018). <https://doi.org/10.1016/j.jnca.2018.10.022>
14. Jason, B.: Recursive feature elimination (rfe) for feature selection in python, <https://machinelearningmastery.com/rfe-feature-selection-in-python/>
15. Javaheri, D., Hosseinzadeh, M., Rahmani, A.: Detection and elimination of spyware and ransomware by intercepting kernel-level system routines. *IEEE Access* **PP**, 1–1 (12 2018). <https://doi.org/10.1109/ACCESS.2018.2884964>
16. Josh, G., Brandon, L., Kyle, W., Pat, L.: Squirtdanger: The swiss army knife malware from veteran malware author thebottle, <https://unit42.paloaltonetworks.com/unit42-squirtdanger-swiss-army-knife-malware-veteran-malware-author-thebottle/>
17. Labs, S.: The ctu-13 dataset. a labeled dataset with botnet, normal and background traffic, <https://www.stratosphereips.org/datasets-ctu13>
18. Lashkari, A.H., A.Kadir, A.F., Gonzalez, H., Mbah, K.F., A. Ghorbani, A.: Towards a network-based framework for android malware detection and characterization. In: 2017 15th Annual Conference on Privacy, Security and Trust (PST). pp. 233–23309 (2017). <https://doi.org/10.1109/PST.2017.00035>
19. Lukas, S.: New telegram-abusing android rat discovered in the wild, welivesecurity by eset, <https://www.welivesecurity.com/2018/06/18/new-telegram-abusing-android-rat/>
20. Mikey, C.: Xagent malware arrives on mac, steals passwords, screenshots, iphone backups, <https://appleinsider.com/articles/17/02/14/xagent-malware-arrives-on-mac-steals-passwords-screenshots-iphone-backups>



21. Mitre: Screen capture, <https://attack.mitre.org/techniques/T1113/>
22. Mohaisen, D., Alrawi, O., Mohaisen, M.: Amal: High-fidelity, behavior-based automated malware analysis and classification. *Computers Security* **52** (04 2015). <https://doi.org/10.1016/j.cose.2015.04.001>
23. Nari, S., Ghorbani, A.: Automated malware classification based on network behavior. pp. 642–647 (01 2013). <https://doi.org/10.1109/ICCNC.2013.6504162>
24. Pan, E., Ren, J., Lindorfer, M., Wilson, C., Choffnes, D.: Panoptispy: Characterizing audio and video exfiltration from android applications. *Proceedings on Privacy Enhancing Technologies* **2018**, 33–50 (10 2018). <https://doi.org/10.1515/popets-2018-0030>
25. Research, K.L.G., Team, A.: The great bank robbery: Carbanak cybergang steals \$1bn from 100 financial institutions worldwide, <https://www.kaspersky.com/about/press-releases/2015-the-great-bank-robbery-carbanak-cybergang-steals-1bn-from-100-financial-institutions-worldwide>
26. Response, S.S.: Regin: Top-tier espionage tool enables stealthy surveillance, <https://www.databreaches.net/regin-top-tier-espionage-tool-enables-stealthy-surveillance/>
27. Sbaï, H., Happa, J., Goldsmith, M., Meftali, S.: Dataset construction and analysis of screenshot malware. In: 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). pp. 646–655 (2020). <https://doi.org/10.1109/TrustCom50675.2020.00091>
28. Shahzad, R., Haider, S., Lavesson, N.: Detection of spyware by mining executable files. pp. 295–302 (02 2010). <https://doi.org/10.1109/ARES.2010.105>
29. Shijo, P., Salim, A.: Integrated static and dynamic analysis for malware detection. *Procedia Computer Science* **46**, 804–811 (12 2015). <https://doi.org/10.1016/j.procs.2015.02.149>
30. Stefan, O.: The missing piece – sophisticated os x backdoor discovered, securelist by kaspersky lab, <https://securelist.com/the-missing-piece-sophisticated-os-x-backdoor-discovered/75990/>
31. You, I., Yim, K.: Malware obfuscation techniques: A brief survey. In: 2010 International Conference on Broadband, Wireless Computing, Communication and Applications. pp. 297–300 (2010). <https://doi.org/10.1109/BWCCA.2010.85>