


# Sniper: Countering Locker Ransomware Attacks Through Natural Language Processing

Abdulrahman Abu Elkhail, *Member, IEEE*, Anya Bacha  *Member, IEEE*, and Hafiz Malik, *Member, IEEE*

**Abstract**—Mobile systems have evolved into versatile devices that end users depend on for carrying out their daily tasks. Unfortunately, the mobile sector has recently fallen prey to a series of ransomware campaigns designed to lock users out of their devices and extort them for payment. In response to these challenges, we propose a novel runtime system that dynamically restores device access by undoing the effects of locker ransomware. A key observation made by this work is that attackers rely on the display of a ransom note on the victim's device to demand payment. Based on this observation, we develop a solution that combines the monitoring of mobile app activity with a natural language processing (NLP) unit that harnesses transformers to detect the appearance of ransom notes. We extensively validate the robustness of our solution against more than five thousand ransomware samples and show that our solution reliably recovers from all the malicious samples that we tested, including overlay screen and change pin-code ransomware. Finally, an evaluation of our proof-of-concept implementation shows minimal performance impact while running a mix of mobile benchmark applications.

**Index Terms**—Ransomware, malware, natural language processing, transformers, mobile security, Android, intrusion detection system, energy efficiency.



## 1 INTRODUCTION

From virtual reality that swoons users in the metaverse to on-the-go business carried out by road warriors, smartphones have evolved into the Swiss Army Knife of computing that end users depend on for a wide range of tasks. The year 2022 alone witnessed more than 255 billion mobile application downloads [1] with 73% of such apps destined to Android devices [2]. Unfortunately, the ubiquity of this platform has garnered significant interest from cybercrime gangs in recent years. For instance, Google officially announced that it removed more than one million risky and untrustworthy applications from its Play Store [3]. This occurred despite all the security checks its applications must undergo before they are made publicly available. Such risks are further amplified by downloads from third party app stores that offer minimal screening for their published apps. An experiment conducted by Allix et al. [4] demonstrated that up to 75% of application downloads from third party stores included malware. This trend underscores the urgency for solutions that can intrinsically safeguard mobile systems from malicious content.

Unfortunately, the mobile sector has recently fallen prey to a series of aggressive ransomware attacks designed to lock users out of their devices and extort them for payment. From enticing campaigns conceived to ensnare victims to creative obfuscation strategies concocted to evade detection, cybercriminals have demonstrated notable success in compromising mobile devices. According to Kaspersky, more than 4.2 million Americans have suffered ransomware attacks on their mobile phones [5]. Another cybersecurity firm confirmed that a slew of smartphone consumers were

denied access to their devices after being hoodwinked by CovidLock, an invasive form of locker ransomware that masquerades itself as a legitimate Coronavirus tracker app [6]. Sadly, the woes of this type of malware show no signs of abating. In 2019 alone, an unprecedented 68,000 new mobile ransomware samples were discovered. The sheer number of ransomware variants in recent years has made it challenging for solutions to scale their coverage and remain effective. This trend underscores the importance of exploring solutions that can detect and recover from such attacks.

In response to these challenges, researchers proposed several defenses [7], [8], [9], [10], [11], [12], [13], [14] designed to shield systems from locker ransomware. A large body of this research [7], [8], [9], [10], however, focused on the detection of locker ransomware in traditional computing systems. For example, Kharraz et al. [7] employed a virtual environment to monitor user activity within standard PC systems. Ozsoy et al. [8] suggested the use of hardware performance counters as a way of fingerprinting ransomware behavior. Other solutions [9], [10] leveraged system level features, such as opcode sequences, API calls, and registry key operations for classifying ransomware. Other work examined the detection of this kind of malware on mobile platforms [11], [12], [13], [14]. For instance, [11] explored the aspect of combining different heuristics from the CPU, memory, and I/O subsystems for classifying ransomware. More work [12], [13], [14] investigated the effectiveness of static analysis in exposing ransomware when analyzing resources, such as Dalvik bytecode and XML-based permission files. Unfortunately, these solutions are vulnerable to obfuscation techniques that involve repackaging apps with malicious content directly presented in native machine code [15]. Virtually, all prior work we are aware of lack the ability to recover mobile devices from locker ransomware after infection.

*The authors are with the University of Michigan, Dearborn, MI, 48128. This work was supported in part by the National Science Foundation under grant CNS-1947580.*

To address the shortcomings of the aforementioned defenses, researchers proposed an assortment of recovery techniques [16], [17], [18], [19], [20]. For example, [16] utilized out-of-place writes in solid-state-drives to restore maliciously encrypted data. Other work [19], [20] suggested the use of backups for recovering maliciously impacted data. All of these techniques, however, are limited to recovering data in systems compromised by cryptographic ransomware. They also require manual intervention by the user and often suffer from long data recovery periods. Virtually, all of the prior work we are aware of lack the ability to recover mobile devices from locker ransomware.

This paper presents a novel runtime system that dynamically restores device access by undoing the effects of locker ransomware. Unlike prior work, our solution seamlessly safeguards compromised devices from such attacks without the need for a manual recovery process or user intervention. A key observation made by this work is that attackers rely on the display of a ransom note on the victim’s device to demand payment. Starting from this observation, we propose a design that combines the monitoring of mobile app activity with a natural language processing (NLP) unit that harnesses transformers to detect the appearance of ransom notes. We thoroughly evaluate the robustness of our solution against more than 5K ransomware samples that span more than 17 families. We show that our solution reliably recovers from all the malicious samples that we tested, including overlay screen and change pin-code ransomware. We show that our experimental proof-of-concept exhibits minimal performance impact while running a mix of mobile benchmark applications [21].

Overall, this paper makes the following contributions:

- Proposes a novel runtime defense that protects mobile devices from the effects of locker ransomware.
- Makes the observation that combining natural language processing with transformers can be harnessed to detect the appearance of ransom notes and consequently identify the presence of locker ransomware.
- Discusses the impact of adversarial machine learning while demonstrating the robustness of our solution against such attacks.
- Presents an end-to-end runtime solution that incurs negligible overhead across a wide range of standard benchmarks.
- Characterizes the resiliency of our design against more than 5K ransomware samples and shows that our solution robustly recovers from the effects of all the samples we tested.

The rest of this paper is organized as follows: Section 2 presents background information. Section 3 discusses the threat model. Section 4 illustrates the design of the proposed defense system. Section 5 presents the methodology and experimental framework used in this work. Section 6 discusses the results of our evaluation. Section 7 details related work; and Section 8 concludes.

## 2 BACKGROUND

### 2.1 Locker Ransomware

Locker ransomware is a form of malware that maliciously appropriates a victim’s device while denying access to its

legitimate user. On mobile systems, this is accomplished by changing the device’s pin code upon gaining root privilege. Another common approach is to permanently overlay a new window that prevents access to the system resources and user installed apps. The malware then extorts the victim to pay a ransom in return for restoring access to the infected device. Payment is often accomplished through cryptocurrency based services since they offer end-to-end anonymity of financial transactions. Unfortunately, paying the ransom does not always guarantee restoration of the infected device.

Locker ransomware generally undergoes five primary stages as part of its lifecycle. These stages include: exploitation and infection, delivery and execution, information exchange, destruction, and extortion. A wide range of methods have been reported for the initial stage of exploitation and infection [22]. However, a combination of social engineering techniques that leverage phishing through social media, instant messaging, and seemingly benign apps, are common ways for luring victims into launching ransomware onto their devices [4]. Once such malware is deployed on a given device, it swindles the victim into granting it administrator privileges. At this point, the malware proceeds to the next stage by establishing a communication channel with a command and control (C&C) server. The channel is mostly used to share details about the infected device and retrieve a set of commands to be executed on the target. Once the necessary information has been exchanged between the C&C server and the infected device, the malware proceeds to the destruction stage that results in the denial of access to the device. The process is concluded by presenting a ransom note to the victim that describes the payment details and recovery instructions.

### 2.2 Transformers

Transformers have demonstrated significant promise in solving NLP tasks [23], [24], [25]. A core component of this architecture relates to its encoder-decoder structure that in turn consists of other subcomponents.

**Encoder and Decoder Stacks.** The encoder serves the purpose of mapping sequences of symbols  $(x_1, \dots, x_n)$  to a sequence of continuous representations  $z = (z_1, \dots, z_n)$ . It consists of  $N$  identical blocks where each block contains two layers: multi-head attention and position-wise fully connected network layers. Each of these layers is surrounded by residual connections and followed by layer normalization in order to speed up training. Input is presented in the form of positional encoded word embeddings in order to keep track of the order of the input sequence. The decoder, on the other hand, assumes the task of generating the output sequence of symbols  $(y_1, \dots, y_n)$ , from a sequence of representations  $z$ . Unlike the encoder, the decoder includes a third layer that performs multi-head attention on the encoder’s output.

**Attention.** Attention is the process of mapping a query and key-value pairs to an output, where the query, keys, values, and output are all vectors. The attention within a given transformer that involves a query, key, and value matrices  $Q$ ,  $K$ , and  $V$ , is computed using the scaled dot-product illustrated in equation (1). The  $\sqrt{d_k}$  term is used for scaling where  $d_k$  is the dimension of the input  $K$ . This mechanism of computing the attention is repeated several times in parallel

for different linear projections of the input. The output is then concatenated and multiplied by an additional weight matrix resulting in what is called multi-head attention. The output of the multi-head attention layer is then used as input into a position-wise fully connected network layer.

$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (1)$$

**Position-wise Feed-Forward Network.** The encoder and decoder of a given transformer contain a fully connected feed-forward network. This step consists of two linear transformations with a ReLU activation in between. The position-wise feed-forward network is expressed using equation (2).

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2 \quad (2)$$

**Embeddings and Softmax.** Input and output tokens are converted to dimensional vectors using learned embeddings. The final decoder output runs through a learned linear transformation followed by a softmax layer to convert the outputs to subsequent token probabilities.

After the introduction of transformers [23], [24] proposed the use of Bidirectional Encoder Representations from Transformers (BERT) as an enhancement for solving NLP tasks. This architecture consists of a bidirectional encoder that learns information from both the left and right sides of a word’s context during the training phase. In addition to enhancements made by BERT, a more recent architecture was introduced, namely, XLNet [25]. Unlike BERT, XLNet leverages a generalized autoregressive pre-training method that enables learning bidirectional contexts more efficiently. Our work examines both of these architectures and evaluates their suitability for detecting ransom notes.

### 3 THREAT MODEL

In this section we detail the assumptions we make about the attackers and how it applies to our design. Our solution is designed to protect mobile systems against ransomware that deliberately locks victims out of their devices. In general, we make the assumption that ransomware is installed through the proper mechanisms already supported by the device’s platform. For instance, an attacker can publish seemingly benign .apk files on a mobile application provider, such as Google Play Store that victims unwittingly install onto their devices. Overall, we make the assumption that any of the following ransomware types can be launched on a system:

- **Overlay Screen Ransomware.** This type of malware is assumed to have the ability to pop a new window that overlays on top of other apps, and consequently, prevent the victim from utilizing their device [26]. Attackers generally rely on social engineering techniques, such as phishing emails and websites, social media, torrent sites, instant messaging apps, and third-party app stores to infect a victim’s device with ransomware.
- **Lock Screen and Pin Code Ransomware.** This type of ransomware is assumed to have the ability to change the pin code of a victim’s lock screen, and consequently, deny access to the device. Akin to the aforementioned category, attackers rely on social engineering strategies for infecting users [27].

In addition to the above, we make the assumption that the attacker is aware of the architecture of our NLP module, and is therefore, capable of conducting adversarial text attacks to evade detection. Another possible attack involves physical access to a device. This method, however, contradicts an important objective ransomware aims to achieve. Ransomware campaigns are often concerned with compromising as many users as possible in order to remain profitable. As such, we do not envision physical attacks being a practical approach in this case.

## 4 THE SNIPER DEFENSE SYSTEM

We propose Sniper, a novel system that dynamically restores device access by undoing the effects of locker ransomware. A key observation made by this work is that attackers rely on the display of a prominent ransom note on the victim’s device to demand payment. Starting from this observation, we devise a solution that combines app activity monitoring with an intrinsically AML hardened NLP unit that doesn’t require retraining. The NLP unit harnesses transformers to accurately detect the appearance of ransom notes. Our design incorporates three primary components: an activity and service monitor that monitors newly installed apps and their associated actions that could result in a device being locked; a text classification unit for detecting ransom notes on an infected device; and a recovery unit that restores device access when locker ransomware is detected. An overview of our defense is shown in Figure 1.

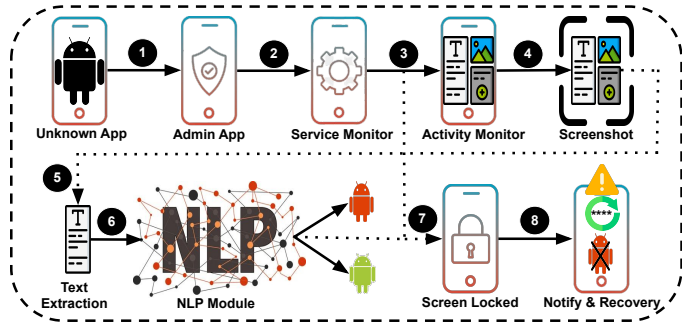


Fig. 1: Overview of the ransomware defense system.

### 4.1 Activity and Service Monitor

Our solution is designed to monitor newly installed apps and their associated actions that can result in a mobile device being locked. An important characteristic that distinguishes locker ransomware from other apps relates to its dependence on administrator privileges. This is because actions, such as locking a screen, setting password rules, and erasing data from a device often necessitate administrative access. As a result, step 1 of our detection process begins with tracking administrator privilege requests.

In Android, communication between running apps and the underlying OS is accomplished through intents. As such, our design monitors system broadcasts that involve the `DEVICE_ADMIN_ENABLED` intent as a way of tracking deployed apps and their associated privileges. Once a given app has received the

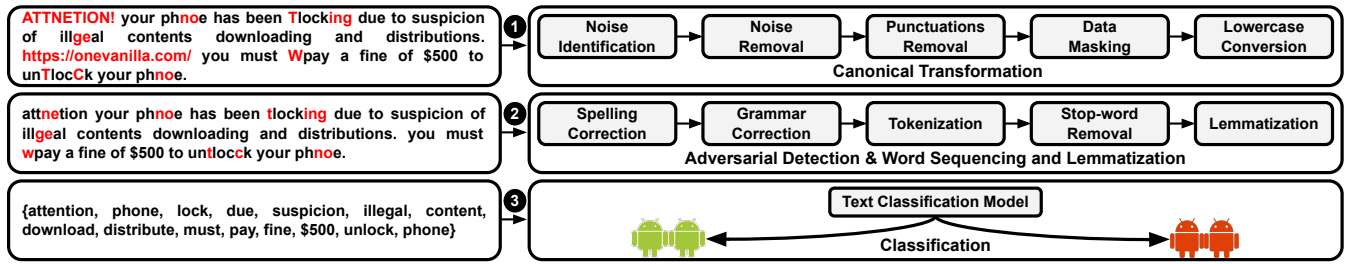


Fig. 2: Main components of the NLP module with an example of how text from a ransom note is progressively transformed.

DEVICE\_ADMIN\_ENABLED intent in response to an administrator request (ACTION\_ADD\_DEVICE\_ADMIN), privilege is deemed granted, and the app is tagged for further analysis. At this point, our design proceeds to the next step and monitors foreground service requests that are issued by the newly established administrator app. In addition to the aforementioned, our solution tracks pin code change requests. This is used to enable recovery and pin restoration in the event that the app is deemed malicious.

Unlike cryptographic ransomware that often runs as a background service, locker ransomware uses a different approach. It aggressively asserts itself as a prominent activity that overtakes the victim’s screen as a mechanism for denying access to the compromised device. As such, foreground service requests represent another important feature for identifying locker ransomware behavior. Our design accounts for this activity under step ② of Figure 1. In the event that an app elects itself to run as a foreground process, our design tags the app as suspicious and proceeds to step ③. At this stage, our solution observes all of the activities the suspicious app displays to the user interface. To make our design efficient, our defense restricts its analysis to newly displayed activities and content updates that modify the device’s user interface. To accomplish this, our design monitors the TYPE\_WINDOW\_STATE\_CHANGED and TYPE\_WINDOW\_CONTENT\_CHANGED events that are triggered when either a new window is launched or updated content is loaded. Hence, any time any of the aforementioned events are triggered, our solution captures the displayed content in the form of an image. It then extracts any text that may be embedded within the captured image and sends it to the NLP module for classification. These actions are illustrated in Figure 1 as steps ④ – ⑥.

## 4.2 Text Classification

The text classification subsystem serves the purpose of detecting ransom notes on an infected device. Once text has been extracted from the device’s user interface, the data is consumed by an NLP unit for classification. To safeguard our design from text based attacks, we augment our system to be adversarially aware. As such, we harden our solution to be resilient against various adversarial machine learning (AML) attacks including character level attacks, word level attacks, and sentence level attacks. To support this approach, any text that is received by the NLP module undergoes three main phases: canonical transformation, adversarial detection, and word sequencing with lemmatization. An

overview of the main components that make up this module is shown in Figure 2.

**Canonical Transformation.** This phase entails transforming input data into a standard form that can be consumed by the classification model. It begins with the removal of noisy symbols that could impact the text classification process. For instance, HTML tags and other non-printable characters are removed from the original input. Similarly, symbols that are commonly used for punctuation, such as hyphens and accents, are eliminated from the text. Upon completion of the aforementioned pre-processing, the canonical transformation phase proceeds to masking parts of the data (data masking). This entails the removal of identifiers, such as uniform resource locators (URLs) and email addresses. The overall canonical transformation phase is concluded with a text standardization step. During this step, all of the retained text is made uniform by converting it to lowercase characters. This step simplifies the parsing of data in the later phases of the NLP module.

**Adversarial Detection.** One of our design objectives is to ensure sufficient protection against AML attacks. Research has shown that adding small perturbations to input are sufficient for an attacker to misclassify the input with high confidence [28]. As such, an attacker can fool a detection system by carefully positioning perturbations in the form of characters and words into a ransom note. For instance, an attacker could leverage text from a previously released ransomware and replace a subset of its words with synonyms. Similarly, an attacker could harness different character manipulation techniques to evade detection without compromising the victim’s ability to comprehend the ransom note [29]. This can be in the form of substitution, swap, insertion, and even deletion of characters in a given word.

To address the aforementioned concerns, our design takes into account the linguistic correctness of a given ransom note. As a result, any text that is received by this component is subjected to a series of rules that perform spelling and grammar corrections. To this end, our design discovers errors in text using a set of predefined rules referred to as bad rules. Such errors are then corrected using another set of rules, namely, good rules. More specifically, our design accomplishes error detection and correction by breaking the input into chunks. Once split into chunks, each word in a given chunk is assigned a part-of-speech (POS) tag. The generated chunks are then compared against a rigorous set of error rules (bad rules). If a match occurs, an error is detected, and is subsequently corrected using the appropriate good rule. This approach ensures that ad-



verserially manipulated words through character swaps, deletions, and insertions are rectified prior to the text being used in the classification stage.

**Word Sequencing and Lemmatization.** This stage assumes the role of tokenizing the data received from the adversarial detection phase into a sequence of meaningful words. The primary purpose of this step is to encode words, so that the context of a given phrase is preserved. The module also includes the removal of stop words (unimportant words), such as “the,” “is,” and “are” that don’t add contextual meaning to a given sentence [30]. Stop word removal also has the benefit of improving the latency of the overall system since this results in less words being processed by the remaining stages of our design [31].

Once the stop words have been removed, the collected words (tokens) are transferred to the lemmatization step. This step assumes the role of mapping each word to its uninflected root (lemma). Since a given word could appear in multiple forms, it is essential to determine a common root that represents all the variations associated with a given word [32]. Therefore, lemmatization ensures that words, such as “gone” and “going” are mapped to a single root of the word, “go.” This step also has the benefit of hardening the system against adversarial text attacks since this approach reduces the search space available to an adversary. Once this step has been completed, the lemmatized text is fed into a pre-trained NLP model for final classification as either benign or malicious.

### 4.3 Device Recovery and Pin Restoration

Our solution is designed to restore device access in the event that locker ransomware is detected. To this end, once the NLP module flags input text as being malicious, the entire app is consequently treated as malicious (step 7). At this stage, further action is taken by our design to restore access to the compromised device (step 8). Our recovery process involves a series of actions including permission revocation, app deletion, and user notification. For instance, if an app has been deemed malicious (e.g. created a malicious pop up that prevents device access), our design responds by immediately revoking the administrator privilege associated with the app. It then proceeds to deleting the app and all of its associated files. Once the app has been deleted, a system reboot is issued. This approach safeguards the user from accidentally paying the ransom. Once the device has been rebooted, the app is no longer considered active and an alert is sent to the user to inform them about the app removal.

Unlike the screen overlay approach that prevents the user from selecting other apps, some ransomware families deny access to their victims by modifying the screen lock pin code of the device. To safeguard against this form of ransomware, our previously discussed service monitor assumes the role of tracking pin code change requests. As such, whenever a ransom note is detected and a pin code request was recorded, our solution resets the pin to a default value (e.g. 0000). To keep the user informed about this action, pin code resets are followed by a notification to the impacted user. The notification is used to inform the user about the newly established pin. On the other hand, if a given app is deemed benign in step 7 and the pin code has not

been changed, the previously captured image in step 4 is discarded and a new evaluation cycle is initiated (transition from step 7 back to step 3). Our design continues to evaluate the running foreground app that has administrator privileges over a programmable duration (e.g. 30 seconds). If the app does not exhibit any malicious behavior over the aforementioned programmable period, the app is deemed benign.

## 5 METHODOLOGY

Training experiments were conducted on a system equipped with two Intel Xeon Gold 6152 processors, 768 GB of main memory, and an NVIDIA Tesla V100 GPU. We re-purposed a diverse set of models that are popular in the field of NLP to perform ransomware detection. In our experiments, we re-purposed a convolutional neural network (CNN) [33], a standard recurrent neural network (RNN) [34], a long short term memory (LSTM) based RNN [35], and a gated recurrent unit (GRU) based RNN [36]. In addition, we employed two transformer based models, namely BERT [24] and XLNet [25]. All of our models were pre-trained using the Google News word embedding [37] that corresponds to a 3 million-word corpus represented in the form of a 300-dimensional vector. We used TensorFlow 1.12.0 with Python 3.6 for training the CNN and RNN based models. BERT and XLNet, on the other hand, were trained using Pytorch 1.2.0 [38]. All of the models were trained with the Adaptive Moment Estimation optimizer (Adam). We explored different configurations including various batch sizes, learning rates, dropout rates, and epochs in order to identify the optimal settings for each model. To construct our dataset, we extracted ransom notes from 5,524 ransomware samples that were supplemented with 8,000 benign sentences from the Internet Movie Database (IMDB). We dedicated 80% of this dataset for training, 10% for validation, and 10% for testing. The final hyperparameters we deployed across the different models are summarized in Table 2.

We conducted experimental evaluations using 5,524 live samples that spanned 17 distinct ransomware families acquired from [39] and [15]. Our dataset consisted of both overlay screen and pin code ransomware. All of the ransomware samples we tested were executed on an Android 11 platform for a minimum of 10 minutes or until the display was locked. The Android image was rolled back to a pristine snapshot after every executed sample. This was performed to reduce the possibility of previously executed ransomware interfering with subsequent runs. The ransomware families we used and their corresponding characteristics are listed in Table 1. Our dataset also included unclassified samples that did not belong to any known families. These samples are denoted as “other” in Table 1. In addition to ransomware samples, we tested 220 benign apps in order to assess the accuracy of our defense in terms of false positives. More specifically, we evaluated applications from multiple categories including health and fitness, productivity, social, entertainment, travel and local, and communication.

We implemented a prototype of our solution on a real platform using the Stratus C5 Elite smartphone that ran Android 11. We used the PaddleOCR v2.8.1 tool [40] to extract text displayed on the device’s user interface. Since

Family	Samples	Lock Screen		Payment Method		Threatening Message	Defeated	Discovered Month
		Overlay Screen	Change Pin Code	Traditional	Electronic			
Jisut	777	✓	✗	✓	✓	✓	✓	2014-6
Pletor	34	✓	✗	✗	✓	✓	✓	2014-6
SimpleLocker	946	✓	✗	✓	✗	✓	✓	2014-6
Aples	33	✓	✗	✗	✓	✓	✓	2014-7
Koler	483	✓	✗	✗	✓	✓	✓	2014-9
LockDroid	215	✓	✗	✗	✓	✓	✓	2014-10
Svpeng	22	✓	✗	✓	✗	✓	✓	2014-10
Slocker	699	✓	✗	✓	✗	✓	✓	2015-5
Xbot	13	✓	✗	✗	✓	✓	✓	2015-5
LockerPin	54	✗	✓	✓	✗	✓	✓	2015-9
Fusob	1,354	✓	✗	✗	✓	✓	✓	2015-10
PornDroid	241	✓	✗	✗	✓	✓	✓	2015-10
WipeLocker	7	✓	✗	✗	✓	✓	✓	2015-10
Charger	11	✓	✗	✗	✓	✓	✓	2017-1
WannaLocker	14	✓	✗	✗	✓	✓	✓	2017-6
DoubleLocker	16	✗	✓	✗	✓	✓	✓	2017-10
MalLocker	7	✓	✗	✗	✓	✓	✓	2020-10
Other	598	N/A	N/A	N/A	N/A	N/A	N/A	N/A

TABLE 1: Summary of ransomware families and their capabilities.

Hyperparameter	RNN, LSTM, GRU	CNN	BERT, XLNet
# of epochs	100	50	5
Batch size	64	32	32
Embedding dim.	300	300	768
Optimizer	Adam	Adam	Adam
Learning rate	0.001	0.001	1e-5
Dropout rate	0.5	0.5	0.1

TABLE 2: Summary of hyperparameters used in all models.

Suite	Benchmark
CPU	Integer Math (IM), Floating Point Math (FPM), Find Prime Numbers (FPNs), Random String Sorting (RSS), Data Encryption (DE), Data Compression (DC), Physics, Extended Instructions (EIs), Single Thread (ST)
Memory	Database Operations (DOs), Memory Read Cached (MRC), Memory Read Uncached (MRU), Memory Write (MW), Memory Latency (ML), Memory Threaded (MT),
I/O	Internal Storage Read (ISR), Internal Storage Write (ISW), External Storage Read (ESR), External Storage Write (ESW)
2D Graphics	Complex Vectors (CVs), Transparent Vectors (TVs), Solid Vectors (SVs), Image Rendering (IR), Image Filters (IFs)
3D Graphics	Simple Test (ST), Complex Test (CT), OpenGL

TABLE 3: Summary of performance benchmarks.

our models were trained to process text written in the English language, we leveraged the GoogleTrans 3.0.0 tool [41]. This allowed us to detect input text written in other languages and translate it into English prior to running it through the NLP model. Furthermore, we characterized the runtime and energy overhead of different models in order to identify their suitability for deployment on mobile devices. We ran a diverse set of test suites available in PassMark [21], a commonly used benchmark for measuring the performance of Android devices. This benchmark allowed us to characterize our design’s sensitivity to a comprehensive set of CPU, memory, I/O, and graphics workloads. The aforementioned test suites are listed in Table 3.

## 6 EVALUATION

### 6.1 Model Robustness

#### 6.1.1 Non-deep Learning Models

Although our study focused on the use of deep learning models, we initially explored lightweight classifiers, such as random forest, support vector machine, and logistic regression to understand their effectiveness in detecting locker ransomware. The results of this experiment along with the models we tested are shown in Figure 3. Overall, we observed that the models in the aforementioned Figure performed well when tested against a non-adversarial test set (*Non-Adv*). This test resulted in detection rates that ranged between 85.2% and 97.2% with a geometric mean of 95.0%. Although lightweight models are attractive for deployment in resource constrained mobile devices, it is important to understand their robustness against adversarial machine learning attacks. To this end, we augmented our non-adversarial test set (*Non-Adv*) with different types of perturbations to produce three AML test sets: Deep-WordBug (*DWB*) [42], Genetic Algorithm (*GA*) [43], and Probability Weighted Word Saliency (*PWWS*) [44].

Our first AML test set, *DWB*, is a test set that focuses on character level text attacks. It evades detection by manipulating key words in a given text through two phases. The first phase entails identifying the importance of words in a given text based on equation (3). In this equation, the importance of the  $i$ -th word in a given text ( $x_1, \dots, x_n$ )

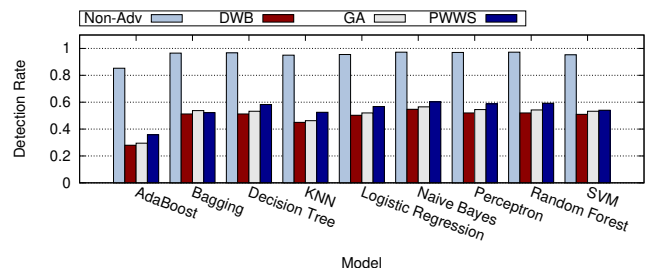


Fig. 3: Summary of detection rates for non-deep learning models with adversarial and non-adversarial test sets.

is computed using the  $Score(x_i)$ .  $\lambda$  is a hyperparameter, and  $F$  is the model’s function that maps an input  $X$  to a label  $Y$ . An example of adversarial text generated using this approach is shown in Figure 4.

$$Score(x_i) = [F(x_1, \dots, x_i) - F(x_1, \dots, x_{i-1})] + \lambda[F(x_i, \dots, x_n) - F(x_{i+1}, \dots, x_n)] \quad (3)$$

Overall, our results show that the non-deep learning models listed in Figure 3 were less effective in detecting adversarially designed inputs. For instance, the aforementioned models yielded detection rates that ranged between 28.0% and 54.8% when evaluated against the *DWB* test set. In general, the detection rates did not exceed 54.8%. This correlates to more than a 40% drop relative to what we observed with the non-adversarial test set (*Non-Adv*). In our experiments, we found that one of the simplest models, naïve bayes, exhibited the best performance which yielded a mere detection rate of 54.8%. Popular algorithms, such as SVM performed even more poorly, since it was incapable of detecting no more than 51.0% of the adversarially designed malicious samples.

In addition to *DWB*, we explored the use of a genetic algorithm (*GA*) to produce supplemental adversarial samples. More specifically, we used the nearest neighbor replacement algorithm defined in equation (4) for generating our *GA* based test set. In this equation,  $X_{adv}$  represents the ideal adversarial sample that would deceive a target model  $f$  into giving false results.  $P^{(g-1)}$ , on the other hand, denotes the  $(g - 1)^{th}$  generation. Unlike *DWB*, *GA* is a word level attack that makes use of the Euclidean distance to reduce the difference in semantics between the original word and its adversarial replacement. An example of adversarial text generated using the *GA* approach is shown in Figure 4.

$$X_{adv} = P^{(g-1)}_{\arg \max f(P^{(g-1)})_{target}} \quad (4)$$

Similar to the *DWB* experiment, our *GA* results yielded detection rates that ranged between 29.5% and 56.5%. On average, we observed a detection rate of 50.4% across the different models. Analogous to the *DWB* experiment, naïve bayes offered the best performance (56.5%) while adaptive boosting performed the worst (29.5%). Analogous to character level attacks, our *GA* results demonstrate that attackers can adversarially compromise non-deep learning models through word level attacks.

<b>Original</b>
ATTENTION! your phone has been locked due to suspicion of illegal contents downloading and distributions. You must pay a fine of \$500 to unlock your phone. → <b>Positive (82%)</b>
<b>Adversarial (DWB)</b>
ATTENTION! your phone has been locked due to <b>ss</b> picion of illegal contents downloading and distributions. You must pay a fine of \$500 to unlock your phone. → <b>Negative (72%)</b>
<b>Adversarial (GA)</b>
<b>NOTICE!</b> your phone has been locked due to suspicion of illegal contents downloading and distributions. You must pay a fee of \$500 to unlock your phone. → <b>Negative (83%)</b>
<b>Adversarial (PWWS)</b>
ATTENTION! your phone has been locked due to suspicion of illegal contents downloading and distributions. You must pay a <b>fee</b> of \$500 to unlock your phone. → <b>Negative (85%)</b>

Fig. 4: Original text example and the corresponding adversarial samples generated using *DWB*, *GA*, and *PWWS*.

Probability Weighted Word Saliency (*PWWS*) is another approach that our study considered for producing adversarial data. Similar to *GA*, *PWWS* is a word-level attack that evades detection by replacing words with their synonyms using WordNet 2. This approach also relies on an optimization procedure for finding the appropriate substitutions by maximizing the word saliency from a set of selected synonyms. Although *PWWS* is a word-level attack, we extend this method to also perform sentence level attacks by perturbing the ordering of the words in a given phrase.

The optimization procedure used in this method is illustrated in equation 5. In this equation,  $R(w_i, L_i)$  represents the best replacement synonym  $w_i^*$  of the  $i^{th}$  word in a given text  $x$ .  $x'_i$  is obtained by replacing the  $i^{th}$  word in  $x$  with each candidate synonym.  $P(Y|x)$  is the probability for classifying text  $x$ , such that the target model can be fooled. An example of adversarial text generated through *PWWS* is shown in Figure 4.

We observed that the non-deep learning models in Figure 3 were slightly resilient in detecting adversarial attacks, but still performed poorly. The aforementioned models yielded detection rates that ranged between 35.8% and 60.3% when evaluated against the *PWWS* test set. On average, the models showed a 3.8% and 5.8% improvement in detection rates relative to *DWB* and *GA*, respectively. However, despite this slight improvement, our results show that non-deep learning models are still vulnerable to adversarial samples generated using the *PWWS* approach. Overall, our results underscore the need for superior models that can offer better protection against AML generated data sets.

$$R(w_i, L_i) = \arg \max P(Y_{true}|x) - P(Y_{true}|x'_i) \quad (5)$$

### 6.1.2 Deep Learning Models

We examined a diverse set of deep learning models that were trained to detect locker ransomware. We evaluated a convolutional neural network (CNN) [33], a standard recurrent neural network (RNN) [34], a long short term memory (LSTM) RNN [35], and a gated recurrent unit (GRU) RNN [36]. In addition, we employed two transformer based models, BERT [24] and XLNet [25]. All of the aforementioned models were evaluated against commonly used quality metrics that include: accuracy, true positive rate (TPR), false positive rate (FPR), recall, F-score, and area under the curve (AUC). Furthermore, all of the models were tested against

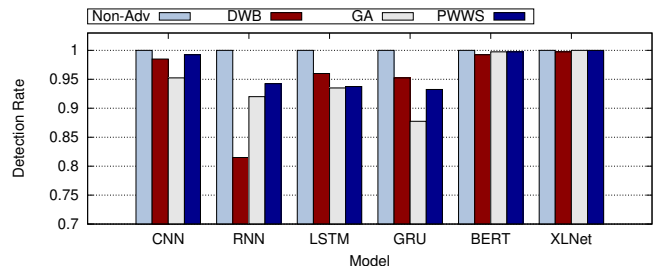


Fig. 5: Summary of detection rates for deep learning models with adversarial and non-adversarial test sets.

Model	Accuracy	TPR	FPR	Recall	F-score	AUC
CNN (Non-Adv)	1	1	0	1	1	1
CNN (DWB)	0.997	0.985	0	0.985	0.992	0.992
CNN (GA)	0.990	0.952	0	0.952	0.975	0.976
CNN (PWWS)	0.998	0.992	0	0.992	0.996	0.996
RNN (Non-Adv)	1	1	0	1	1	1
RNN (DWB)	0.963	0.815	0	0.815	0.898	0.907
RNN (GA)	0.984	0.920	0	0.920	0.958	0.960
RNN (PWWS)	0.988	0.942	0	0.942	0.970	0.971
LSTM (Non-Adv)	1	1	0	1	1	1
LSTM (DWB)	0.992	0.960	0	0.960	0.979	0.980
LSTM (GA)	0.987	0.935	0	0.935	0.966	0.967
LSTM (PWWS)	0.987	0.937	0	0.937	0.967	0.968
GRU (Non-Adv)	1	1	0	1	1	1
GRU (DWB)	0.990	0.952	0	0.952	0.975	0.976
GRU (GA)	0.975	0.877	0	0.877	0.934	0.938
GRU (PWWS)	0.986	0.932	0	0.932	0.965	0.966
BERT (Non-Adv)	1	1	0	1	1	1
BERT (DWB)	0.998	0.992	0	0.992	0.996	0.996
BERT (GA)	0.999	0.997	0	0.997	0.998	0.998
BERT (PWWS)	0.999	0.997	0	0.997	0.998	0.998
XLNet (Non-Adv)	1	1	0	1	1	1
XLNet (DWB)	0.999	0.997	0	0.997	0.998	0.998
XLNet (GA)	1	1	0	1	1	1
XLNet (PWWS)	1	1	0	1	1	1

TABLE 4: Summary of quality metrics of NLP models using Non-Adv and DWB, GA and PWWS adversarial test sets.

the same non-adversarial and adversarial datasets: *Non-Adv*, *DWB*, *GA*, and *PWWS*.

Our results indicate that all of the deep learning models achieved ideal performance under the *Non-Adv* dataset (100% detection rate). Unlike the non-parametric models, none of the deep learning designs yielded any false positives or negatives while using the non-adversarial dataset. This correlates to a 5% improvement relative to the rates achieved by the non-parametric models previously depicted in Figure 3. In addition, we observed a significant improvement in the ability to detect adversarially designed ransom notes. On average, the deep learning models demonstrated a 45% increase in the ability to detect malicious samples that span the *DWB*, *GA*, and *PWWS* datasets.

Figure 5 reveals that XLNet outperformed the remaining models with BERT being a close second. Our results demonstrate that the transformer based models offered better performance over their CNN and RNN counterparts across all the metrics depicted in Table 4. For instance, XLNet and BERT detected 99.7% and 99.2% of the *DWB* samples. This correlates to XLNet and BERT having one and three false negatives, respectively. The CNN model, on the other hand, experienced a slightly lower detection rate (98.5%) which corresponds to six malicious samples going undetected. Unlike the transformer and CNN models, however, combining our solution with RNNs resulted in suboptimal detection rates. Although such models previously represented the state-of-the-art in the field of NLP, the standard, LSTM, and GRU based RNNs misclassified 74, 16, and 19 *DWB* samples, respectively. This data suggests that RNNs are less effective in combating sophisticated forms of locker ransomware that employ AML techniques to evade detection.

A similar trend was observed with the *GA* and *PWWS* datasets. Overall, our transformer based models performed slightly better while using the aforementioned datasets. We observed that XLNet classified all of the *GA* and *PWWS* samples correctly. Similarly, BERT exhibited near ideal detection rates by misclassifying only one malicious sample

from each dataset. The CNN model, on the other hand, experienced a 3% drop while using the *GA* dataset (95.2%). The CNN model, however, improved its detection rate to 99.2% while using the *PWWS* dataset. This correlates to only three samples being undetected. On the other hand, the RNN models manifested suboptimal detection rates that ranged between 87.7% and 94.2%. This correlates to an average that is slightly above 92% when evaluated with samples from the *GA* and *PWWS* datasets. We attribute the high detection rates of transformer models like BERT and XLNet to their ability to capture bidirectional context. Unlike RNNs that process text sequentially, and CNNs that rely on local patterns, transformers handle entire sentences simultaneously, providing a more comprehensive understanding. The self-attention mechanism enables transformers to understand complex relationships and long-range dependencies between words in a given sentence.

To further evaluate the robustness of the deep learning models, we trained these models using a training set that consisted of older ransomware samples released between 2014 and 2015. We then tested the trained models against a test set (unseen ransomware) that comprised of samples from 2017, 2020, and other. A summary of the detection rates for these models against the unseen ransomware test set is illustrated in Figure 6.

Overall, we observe that XLNet outperformed all the other models, achieving a detection rate of 99.6%, closely followed by BERT with a detection rate of 99.3%. These results reinforce that transformer-based models are particularly effective at identifying ransomware, even when faced with previously unseen samples. In comparison, the CNN model exhibited a slightly lower detection rate of 98.6%. On the other hand, the RNN-based models were less adept at detecting unseen ransomware samples. For instance, the RNN, GRU, and LSTM models achieved detection rates of 93%, 96.6%, and 97.8% respectively when tested against the unseen ransomware test set.

Altogether, our results show that XLNet exhibited supe-



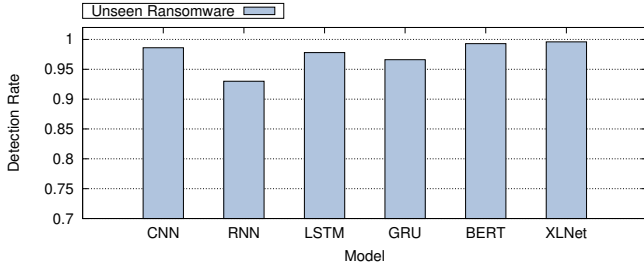


Fig. 6: Summary of detection rates for deep learning models against unseen ransomware test set.

rior performance in detecting unseen ransomware samples and demonstrated the most resilience against adversarial attacks. Although the BERT and CNN models produced competitive detection rates, XLNet consistently outperformed all of the models across a diverse set of evasion techniques that span, character (*DWB*), word (*GA*), and sentence (*PWWS*) level attacks. Although devising defenses that can scale to detect unseen forms of ransomware while remaining robust against AML attacks is critical, other factors, such as energy and runtime must be considered in the context of mobile platforms.

### 6.1.3 Model Runtime and Energy

The performance overhead of our defense is primarily a function of the runtime associated with the text classification module. To this end, we conducted runtime and energy efficiency experiments across the different NLP models in order to understand their suitability for deployment on mobile devices. Figure 7 shows the runtime of different NLP models based on measurements from our prototype system.

Overall, we observed that XLNet achieved the best performance in terms of runtime. On average, XLNet only required 21 ms to run text classification tasks to completion. BERT, on the other hand, incurred an additional 5% overhead while attempting to classify the same tasks. This overhead increased significantly while using the CNN and RNN models. For instance, the CNN model required almost 27 ms to run to completion. This correlates to 27% and 21% increases in runtime relative to the transformer based models, XLNet and BERT. In general, the RNN models proved to be the least efficient, offering the worst execution times. For example, the GRU model took 33 ms to classify text, on average. This corresponds to a 56% increase relative to what we observed with XLNet. Similarly, the LSTM and standard RNN models experienced comparable overheads. We observed 33 ms and 31 ms for the LSTM and standard RNN models, respectively. This is not surprising given that a typical RNN neuron requires 8x the number of weights and multiply-accumulate ops of a standard CNN cell [45].

Figure 8 shows the energy consumption of the previously discussed NLP models. Overall, our energy results correlate to our runtime findings. We observed that the transformer based models offered the best energy efficiency while the RNN models offered the worst. To put things in perspective, both XLNet and BERT consumed 129 mj and 136 mj, respectively. This overhead increased to 164 mj while using a CNN. However, significantly more energy

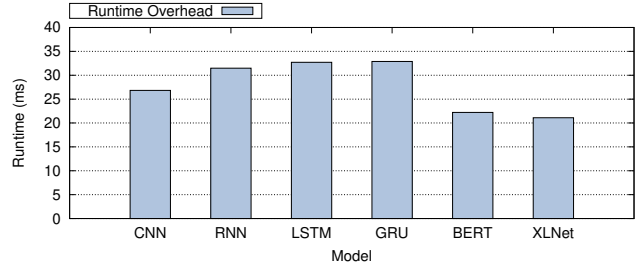


Fig. 7: Summary of the runtime for different NLP models.

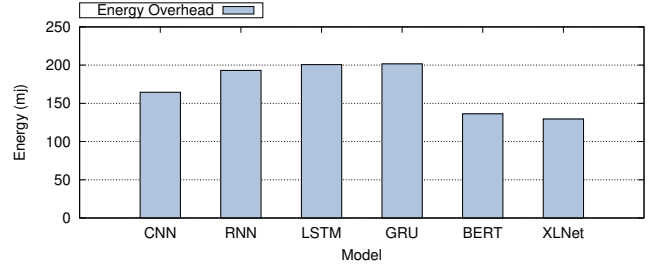


Fig. 8: Summary of the energy for different NLP models.

was consumed while using a GRU. For instance, the GRU expended 20% more energy (202 mj) compared to its CNN counterpart. Although the LSTM and standard RNN models consumed slightly less energy, they still required 201 mj and 193 mj, respectively. Overall, we found that in addition to robustness to AML attacks, XLNet is the most energy efficient NLP model. This makes XLNet the most suitable choice for deployment on mobile systems that are often resource and energy constrained.

Our defense has three main sources of runtime and energy overhead. These sources correspond to performing canonical transformation, adversarial detection, and text classification. Figures 9 and 10 show a breakdown of the runtime and energy overhead for our design. The first source of runtime and energy overhead relates to the canonical transformation phase. This phase is responsible for transforming input data into a standard form that can be consumed by the classification model. We measured runtime and energy costs of 38 ms and 233 mj. The second source of overhead relates to the adversarial detection phase. This phase involves performing spelling and grammar correctness to ensure linguistically sound text data. This phase marks the biggest source of overhead within our design. We observed runtime and energy costs of 72 ms and 443 mj. This is 1.9x the runtime and energy consumption of the canonical transformation phase. However, despite the high overhead, this phase is critical to our design since it represents the core component for defending against AML attacks. The final source of overhead within our design relates to classification. On average, this phase required 21 ms and 129 mj to classify 10K sentences from a mix of ransomware and benign applications while using the XLNet model. Overall, our end-to-end design required 131 ms and 805 mj to perform complete detection of ransomware samples. It is important to note that in most cases, a given user attempting to launch an app will not result in any of these phases being executed. The aforementioned overhead is introduced only

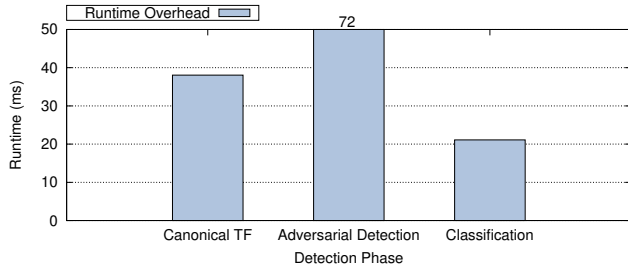


Fig. 9: Runtime breakdown of different NLP phases.

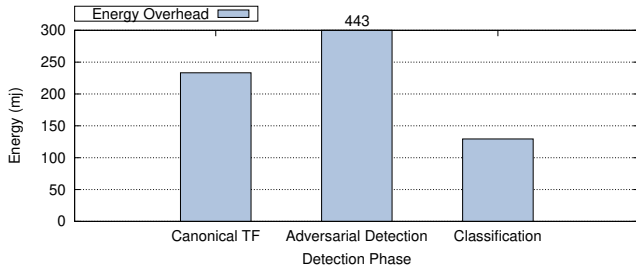


Fig. 10: Energy breakdown of different NLP phases.

when a launched app is deemed suspicious.

## 6.2 Workload Characterization

**Computational Workloads.** We ran a wide range of experiments that aimed to evaluate the performance impact of our system on CPU workloads. The CPU workloads we used are listed in Table 3. Figure 11 summarizes the overhead of our solution on CPU workloads relative to an unsecured baseline. In general, most of the CPU-bound workloads incurred an insignificant amount of overhead when dispatched on our solution. We found that our system incurred a negligible overhead that was well under 2% across most programs with the exception of *Integer Math* (IM) and *Data Encryption* (DE). IM is a workload that aims to evaluate how fast a CPU can perform arithmetic operations, such as addition, multiplication, and division. DE, on the other hand, is a workload that encrypts and hashes blocks of random data using algorithms, such as AES, ECDSA, and SHA256. Unlike the other programs, the IM and DE workloads incurred 2.2% and 3.0% reduction in performance, respectively. We attribute this reduction to our design continuously running in the background in order to monitor service requests issued by a user’s apps.

**Memory Workloads.** We also ran various workloads that aimed to evaluate the impact of our solution on the memory subsystem. The memory workloads we used for this purpose are listed in Table 3. Figure 12 summarizes the performance overhead of our solution on memory centric workloads relative to an unsecured baseline. On average, our solution exhibited a performance impact that was below 3.2%. The majority of the workloads encountered overheads that were well below 3% with the exception of the *Database Operations* (DOs) workload, which experienced a 9.0% overhead. We attribute this reduction in performance to the size of our NLP module. Since our design relies on a pre-trained model, a substantial amount of memory is needed

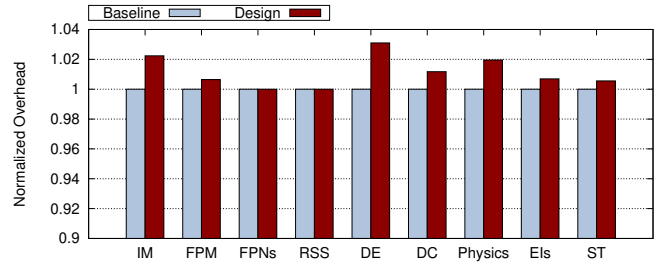


Fig. 11: Performance overhead on CPU workloads.

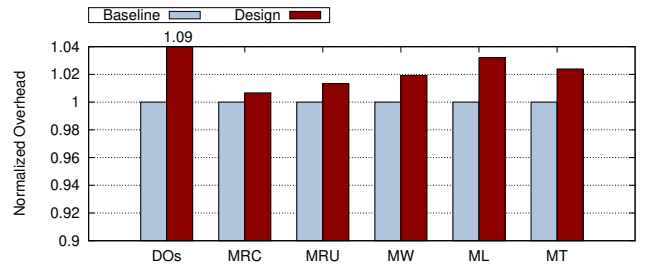


Fig. 12: Performance overhead on memory workloads.

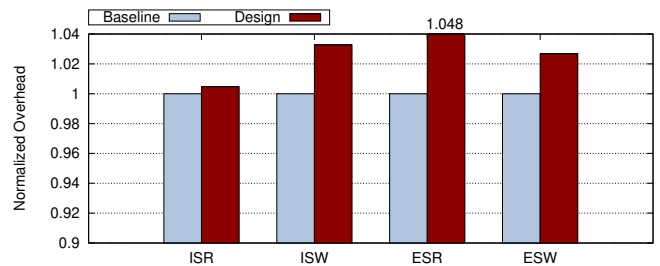


Fig. 13: Performance overhead on I/O workloads.

in order to fit all of the hyperparameters in memory. This can induce low memory conditions when combined with memory intensive apps, such as database applications.

**I/O Workloads.** We also conducted tests that assessed the performance impact of our design on I/O. The I/O workloads we used can be found in Table 3. Figure 13 summarizes the performance overhead of our solution on the I/O subsystem relative to an unsecured baseline. We observed an average performance decline that was less than 2.8%. The majority of workloads experienced overhead that was well below 3%, with the exception of *External Storage Read* (ESR) and *External Storage Write* (ESW), which had performance losses of 4.8% and 3.3%, respectively. This overhead is only applicable when new apps are installed and thus evaluated for suspicious behavior.

**Overall Performance Impact.** Figure 14 illustrates the overall performance impact of our design relative to an unsecured baseline for different subsystems. In general, our design had the least impact on the CPU subsystem with an average performance reduction that is under 1.2%. Additionally, we observed that our design exhibited a 2.8% loss in performance across I/O workloads. Similarly, our design experienced slightly elevated overheads while running memory bound workloads, averaging a 3.2% drop in performance. On the other hand, our design had no impact

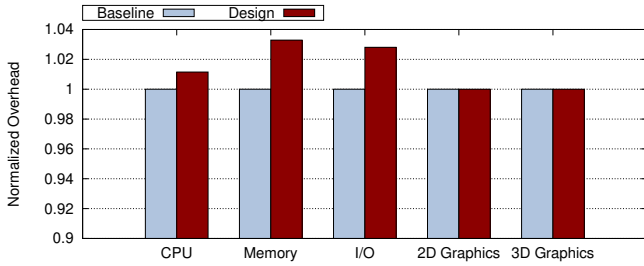


Fig. 14: Performance overhead of overall system.

on the performance of graphics workloads. This is expected since our design does not rely on the device’s GPU for any of its analysis. Overall, our design experienced a reduction in performance that was less than 1% when considering all of the subsystems together. Such negligible overheads demonstrate the efficiency of our proposed solution.

### 6.3 Discussion

Our study entailed conducting a comprehensive set of experiments designed to evaluate the ability of our solution in detecting and recovering from locker ransomware. Our validation included over 200 benign apps and more than 5K ransomware samples that spanned 17 ransomware families. The ransomware families we used are shown in Table 1.

Altogether, our results demonstrate that our design is reliable in differentiating between benign and ransomware activity. Our solution accurately classified all of the malicious samples that we ran, including overlay screen and change pin-code ransomware. Similarly, our design demonstrated resiliency with regards to misclassifying benign apps. Our system was able to correctly classify all of the benign apps without triggering any false positives. During this process, we observed that less than 1% of the benign apps we tested requested administrator privileges. This correlates to only two apps out of a total of 220 applications that requested such privileges. More specifically, Microsoft Authenticator [46], a multifactor authentication app that enables users to securely sign into their accounts was one of the applications that requested administrator privileges. This app requests administrator privileges in order to collect GPS data and enable location based authentication. The second app that required administrator privileges was Find My Device [47]. Since this is an app that allows users to locate their lost devices and lock them until they are found, administrator privileges are requested. Our solution was also able to correctly classify such apps without yielding any false positives. The results of the entire experiment are summarized in Table 5.

<i>Evaluation</i>	<i>Results</i>
<b>Total Samples</b>	5,744
<b>Ransomware Samples</b>	5,524
<b>Benign Applications</b>	220
<b>False Positives</b>	0.0%
<b>False Negatives</b>	0.0%

TABLE 5: Summary of false positive and negative results.

Dataset	Tesseract-OCR	EasyOCR	PaddleOCR
Clean ransomware	80.89%	82.95%	99.48%
Low noise ransomware	58.11%	78.29%	99.44%
Medium noise ransomware	49.02%	68.75%	99.21%
High noise ransomware	46.50%	62.34%	98.25%
Captcha version 2 [50]	45.26%	61.77%	85.82%

TABLE 6: Summary of text accuracy for various OCR tools across different datasets.

In addition, our solution was able to recover from all of the ransomware samples we tested, including overlay screen and change pin-code ransomware. We observed that all of the ransomware samples we encountered requested administrator privileges upon installation, in order to control the screen lock action. In the case of overlay screen ransomware, all of the samples leveraged the foreground service for displaying their ransom notes to the user interface. For instance, when we tested Svpeng [26], a ransomware that infiltrates Android smartphones through a fake Adobe Flash update message, it locked the UI display and popped up a bogus FBI message that demanded a \$200 ransom. In addition to Svpeng, our system was able to detect malicious activity from all samples within 131 ms of the screen being locked. In all cases, our solution successfully restored access to the infected system within the aforementioned period by revoking the administrator privileges, deleting the offending app, and issuing a notification to the user.

We also observed that some ransomware samples changed the pin code in order to prevent users from accessing their devices. For example, we tested Lockerpin [27], a ransomware that changes the pin of an infected device followed by a \$500 ransom demand. Our design was able to detect such activity within 131 ms of the screen being locked. It removed the malicious app and all of its related files after revoking administrator privileges, reset the screen lock pin code with a new pin code of 0000, and notified the user about the app removal and the newly established pin.

We also conducted a series of experiments to evaluate different OCR tools and their robustness in extracting text from images produced by locker ransomware. We examined commonly used open source tools, such as Tesseract-OCR v5.0.0 [48], EasyOCR v1.7.1 [49], and PaddleOCR v2.8.1 [40]. Our evaluation process began with testing the aforementioned tools on a dataset of 5,524 clean ransomware images, which were collected from over 17 ransomware families. To further assess the robustness of these OCR tools under more challenging conditions, we injected the clean ransomware dataset with different levels of noise. More specifically, we applied Gaussian noise, salt-and-pepper noise, and Gaussian blur to the clean ransomware dataset at different intensities. This allowed us to produce three different noise-level datasets: low-level noise (10% intensity), medium-level noise (25% intensity), and high-level noise (50% intensity). An example of an image obtained from the Aple ransomware after being exposed to different noise levels is shown in Figure 15. Moreover, to evaluate the robustness of the chosen OCR tools when exposed to images with sophisticated backgrounds, we collected 1,070 complex background images sourced from CAPTCHA dataset version 2 [50]. The findings of these experiments are presented in Table 6.





Fig. 15: Ransom note from the Aples ransomware at different noise levels (a) Clean image with no noise added. (b) Low-level noise with 10% intensity. (c) Medium-level noise with 25% intensity. (d) High-level noise with 50% intensity.

Table 6 presents a summary of text accuracy for Tesseract-OCR, EasyOCR, and PaddleOCR across various datasets, including clean ransomware images, noise-injected ransomware images at different levels, and CAPTCHA images with complex backgrounds. The results show that PaddleOCR consistently outperformed the other two OCR tools in terms of accuracy, achieving a near-perfect accuracy of 99.48% on clean ransomware images. The same tool also maintained high accuracy even under noisy conditions, achieving a 98.25% accuracy on high-level noise ransomware images. Additionally, PaddleOCR exhibited exceptional robustness across a variety of testing scenarios, including complex CAPTCHA images with sophisticated backgrounds, where it achieved an accuracy that exceeded 85%. Unlike ransomware images, which are deliberately crafted to be easily readable to ensure that victims can clearly understand the demands and pay ransoms, CAPTCHA images are intentionally designed to be challenging to decipher. This approach in turn affects the victim's ability to comprehend messages concocted using this algorithm. Despite these challenges, PaddleOCR proved to be effective in processing ransomware images. This was the case even under noisy conditions and deliberately obfuscated CAPTCHA images. These results underscore the versatility of this tool and its ability to scale to diverse text recognition tasks.

In contrast, EasyOCR displayed moderate performance, with an accuracy of 82.95% on clean ransomware images, 62.34% on high-level noise images, and 61.77% accuracy on CAPTCHA images with complex backgrounds. Tesseract-OCR, on the other hand, showed lower accuracy rates across all datasets, with the highest being 80.89% on clean

ransomware images, 46.50% on high-level noise images, and 45.26% on CAPTCHA images. These findings further support PaddleOCR's robustness as an open-source tool for text extraction purposes. While some commercial OCR tools [51], such as Google Cloud Vision and AWS Textract, claim superiority over open-source tools, PaddleOCR remains an effective option for open-source applications and the development of proof-of-concept solutions.

To assess the impact of text extraction tools on the accuracy of our overall solution, we evaluated our design using Tesseract-OCR which exhibited the worst performance. We then tested our deep learning models against three datasets with increasing noise levels (low, medium, and high), in addition to a clean dataset. The results of this experiment are illustrated in Figure 16. Overall, we observed that all the models achieved perfect detection on the clean dataset. However, the effectiveness of the models varied under different noisy conditions. For instance, RNN-based models exhibited the most pronounced decline, with detection rates dropping from 81.0% to 63.0% as noise increased. CNN experienced a gradual decrease in its detection rates from 87.0% under low noise to 72.5% under high noise conditions, reflecting moderate noise resilience. In contrast, transformer-based models like XLNet and BERT demonstrated superior robustness, maintaining high detection rates even under high noise conditions. For instance, XLNet achieved 97.0%, 94.0%, and 86.0% for low, medium, and high noise levels, respectively. BERT, on the other hand, achieved 96.0%, 93.0%, and 85.0% when exposed to the same noise levels. This suggests that transformer-based models are significantly more effective at handling noisy data compared to CNN and RNN-based models. This enhanced per-



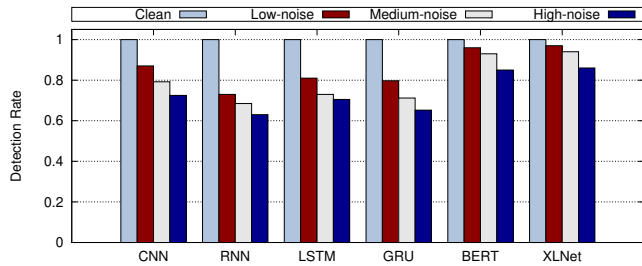


Fig. 16: Summary of detection rates for deep learning models using Tesseract-OCR tool across various noise levels.

formance is attributed to the transformer’s ability to capture bidirectional context and leverage its self-attention mechanism for a deeper understanding of complex relationships and establishing long-range dependencies between words. Unlike RNNs, which process text sequentially, and CNNs, which focus on local patterns, transformers evaluate entire sentences simultaneously, providing a more comprehensive and nuanced understanding of the text at hand.

## 7 RELATED WORK

**Ransomware Detection.** Several studies [11], [12], [13], [14], [56], [59], [60], [61] explored various detection techniques against Android ransomware. For example, [59] relied on UI analysis on mobile devices to distinguish between benign and malicious behavior. Other techniques [12], [13], [61] used disparate forms of static analysis including API calls, text embedded within APK files, and bytecode present in DEX files for performing detection. Unfortunately, such solutions can be thwarted by embedding malicious content in encrypted form [62]. More work [14], [60] considered permissions for fingerprinting malicious apps. In general, the aforementioned methods rely on analyzing high-level bytecode or permissions files, which are ineffective against obfuscation techniques that repackage apps with malicious content directly in native form [15]. Other work [28], [63] proposed detecting ransomware at the native instruction level. However, such work lacks that ability to recover from ransomware attacks. In addition, [11] examined system resources associated with a given process including CPU, memory, and I/O activity for detection. However, relying on usage thresholds have been shown to be unreliable in predicting malicious activity [64]. Other work [56] proposed a hybrid approach in which apps are initially assessed statically by examining embedded text, images, API calls, and permissions present within APK files. API call sequences are then monitored at runtime in the event that the static analysis phase classifies a given app as suspicious. However, given the vulnerability of static analysis solutions to different obfuscation techniques [15], [62], the dynamic layer can be easily rendered ineffective. Overall, all prior work we are aware of is incapable of recovering mobile systems from ransomware attacks once a device has been compromised. Unlike prior work, we present a comprehensive system that seamlessly safeguards compromised devices from locker ransomware without the need for manual recovery.

**Ransomware Recovery.** To complement the deficiencies of prior work, [16], [17], [18], [20], [65] proposed ransomware

recovery solutions. For example, [16] used out-of-place writes in solid-state-drives to recover impacted data. Another framework [17], proposed the use of a key vault created to store secret keys generated on a given platform for later use in order to decrypt maliciously impacted files. Other work [18] suggested quarantining backup data on an inaccessible volume to safeguard against backup spoliation attacks. Similarly, [20] suggested the use of backup techniques as a defense against ransomware attacks. Unfortunately, continuously backing up mobile devices is not a viable option due to the limited amount of accessible storage and computing resources. Furthermore, such solutions are mainly concerned with cryptographic ransomware while our work concentrates on locker ransomware.

**Comparison to Existing Defenses.** We compared Sniper against several state-of-the-art ransomware defense systems. The differences are outlined in Table 7. Overall, the majority of previous work focused on developing ransomware defenses for the Android platform [9], [12], [13], [14], [56], [58], while a few others [52], [53], [54] targeted Windows. In general, we find that most prior work [9], [12], [13], [52], [53] relied on different forms of static analysis, including the examination of API calls, images, strings embedded in APK files, and bytecode sequences present in DEX files, along with the use of supervised machine learning algorithms for performing detection. However, these approaches are vulnerable to evasion techniques, such as, embedding malicious content in encrypted form [62]. Other work [14] has explored combining the use of permissions with a customized machine learning algorithm for fingerprinting and detecting malicious applications. Unfortunately, methods that focus on analyzing high-level bytecode or permissions files are ineffective against obfuscation techniques that repackage applications with malicious content embedded directly in native code [15], [66]. Other approaches [54], [58] proposed the use of opcodes for training CNN-based models to perform detection. However, these methods are ineffective against obfuscation techniques [15], [62] that alter opcode sequences without changing malicious behavior. Other work by [56] employed a hybrid approach where an app’s embedded text, images, API calls, and permissions are analyzed statically. API calls are then monitored in the event that the static analysis phase flags the app as suspicious. However, due to the susceptibility of static analysis solutions to various obfuscation techniques [15], [62], the dynamic layer could easily be rendered ineffective. Unlike these solutions, our solution combines runtime monitoring of mobile app activity with an NLP unit that utilizes transformers for more robust detection.

In general, our evaluation consisted of executing a larger dataset size of 5,524 locker ransomware samples relative to other work [9], [12], [13], [14], [52], [53], [54], [56]. With the exception of work by [58], our dataset size for locker ransomware is  $2\times - 15\times$  larger than what was considered in the other studies. Although work by [58] used a slightly larger dataset size of 5,852 samples, the study only covered ransomware samples collected in September 2017. Unlike [58], our work considers samples collected over several years and across more than 17 ransomware families. Furthermore, in terms of detection capabilities, we find that previous work [9], [12], [13], [14], [52], [53], [54],

TABLE 7: Comparison of our solution (Sniper) with existing ransomware defense systems.

Work	Platform	Detection Method	Features	Dataset		Detection of Adv. Attacks	Detection of Unseen Ran.	Detection Rate			Device Recovery	PIN Rest.	Overhead Analysis
				Size	Source			Non-Adv.	Adv.	Unseen Ran.			
[52]	Windows	State-of-Art Machine Learning	API Calls, Static Code Analysis	582	VirusShare	✗	✗	0.963	-	-	✗	✗	✗
[53]	Windows	State-of-Art Machine Learning	API Calls, Static Code Analysis	360	VirusShare	✗	✗	0.971	-	-	✗	✗	✗
[54]	Windows	Self-Attention CNN model	Opcodes	1,787	VirusTotal	✗	✗	0.875	-	-	✗	✗	✗
[12]	Android	State-of-Art NLP models	API Calls, Static Code Analysis	650	VirusTotal	✗	✓	1.0	-	0.846	✗	✗	✗
[14]	Android	Customized Machine Learning	API Calls, Permissions	500	VirusTotal [12], [55]	✗	✗	0.980	-	-	✗	✗	✗
[56]	Android	State-of-Art Machine Learning	API Call Sequences, Static Code Analysis	1,928	[12], [13] [55], [57]	✗	✗	0.975	-	-	✗	✗	✗
[9]	Android	State-of-Art Machine Learning	API Calls, Static Code Analysis	666	VirusTotal	✗	✗	1.0	-	-	✗	✗	✗
[13]	Android	State-of-Art Machine Learning	Static Code Analysis	2,045	VirusTotal [12]	✗	✓	0.978	-	0.852	✗	✗	✗
[58]	Android	State-of-Art CNN models	Opcodes	5,852	Leopard Mobile Inc.	✗	✗	0.968	-	-	✗	✗	✗
<b>Sniper</b>	Android	State-of-Art NLP Transformer models	Activity and Service Monitor	5,524	VirusTotal [15]	✓	✓	1.0	0.997	0.996	✓	✓	✓

[56], [58] did not evaluate the robustness of their solutions against adversarial attacks. Our solution, on the other hand, provides a thorough discussion on the impact of adversarial attacks, underlining the necessity of hardening NLP models against such attacks. For instance, Sniper stands out in terms of its ability to detect adversarial attacks, achieving a high detection rate of 99.7%. It also achieves a perfect detection rate of 100% against non-adversarial ransomware, with a high detection rate of 99.6% against unseen ransomware. While two other solutions [9], [12] also achieved a perfect detection rate of 100% against non-adversarial ransomware, and [13] achieved a detection rate above 97%, [9] did not explicitly evaluate their solution against unseen ransomware. Additionally, both [12] and [13] solutions experienced a significant drop in their effectiveness when tested against unseen ransomware. They merely accomplished detection rates of 84.6% and 85.2%, respectively.

Another aspect we examined in our comparison relates to the ability to recover mobile systems and restore pins from ransomware attacks after a device has been compromised. Overall, we find that none of the other defenses listed in Table 7 demonstrated the capability to recover mobile systems and restore pins after being compromised. Our solution demonstrates its capability to seamlessly recover mobile systems and restore pins of compromised devices without manual intervention. Our solution was able to successfully achieve this across 5,524 ransomware samples obtained from more than 17 families. Furthermore, we find that all the defenses listed in Table 7 did not explicitly evaluate the performance overhead of their solutions. Our solution, on the other hand, goes beyond prior work by providing a thorough analysis of performance overhead, including runtime, energy consumption, and the impact on CPU, memory, and I/O while running a mix of standard mobile benchmarks. Our evaluation demonstrated that our proof-of-concept implementation incurs a minimal performance overhead of less than 1% when considering all subsystems together. Such negligible overheads underscore the efficiency of our solution in defending against locker ransomware attacks.

## 8 CONCLUSION

In this study, we propose, Sniper, a novel runtime defense that safeguards mobile devices from locker ransomware. We

combine a lightweight NLP module that efficiently detects ransom notes with a system that dynamically tracks runtime app behavior. We evaluate the robustness of our solution against more than 5K ransomware samples. Finally, we show that our solution incurs minimal performance impact while running a mix of mobile benchmark workloads.

## REFERENCES

- [1] Lara Ceci. Number of mobile app downloads worldwide from 2016 to 2022. <https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads>, 2023.
- [2] Mobile Operating System Market Share Worldwide. <https://gs.statcounter.com/os-market-share/mobile/worldwide>, 2022.
- [3] Alan Friedman. Google kept over a million bad apps out of the Play Store last year. [https://www.phonearena.com/news/google-kept-more-than-million-bad-apps-out-of-playstore\\_id139880](https://www.phonearena.com/news/google-kept-more-than-million-bad-apps-out-of-playstore_id139880), 2022.
- [4] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE, 2016.
- [5] Ivana Vijiñovic. Ransomware statistics in 2023: From random barrages to targeted hits. <https://dataprot.net/statistics/ransomware-statistics/>, 2023.
- [6] Chad Anderson. CovidLock Update: Deeper Analysis of Coronavirus Android Ransomware. <https://www.domaintools.com/resources/blog/covidlock-update-coronavirus-ransomware/>, 2020.
- [7] Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. {UNVEIL}: A {Large-Scale}, automated approach to detecting ransomware. In *25th USENIX security symposium (USENIX Security 16)*, pages 757–772, 2016.
- [8] Meltem Ozsoy, Caleb Donovan, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Malware-aware processors: A framework for efficient online malware detection. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 651–661. IEEE, 2015.
- [9] Brijesh Jethva, Issa Traoré, Asem Ghaleb, Karim Ganame, and Sherif Ahmed. Multilayer ransomware detection using grouped registry key operations, file entropy and file signature monitoring. *Journal of Computer Security*, 28(3):337–373, 2020.
- [10] Hanqi Zhang, Xi Xiao, Francesco Mercaldo, Shiguang Ni, Fabio Martinelli, and Arun Kumar Sangaiah. Classification of ransomware families with machine learning based onn-gram of opcodes. *Future Generation Computer Systems*, 90:211–221, 2019.
- [11] Sanggeun Song, Bongjoon Kim, and Sangjun Lee. The effective ransomware prevention technique using process monitoring on android platform. *Mobile Information Systems*, 2016, 2016.
- [12] Nicolás Andronio, Stefano Zanero, and Federico Maggi. Heldroid: Dissecting and detecting mobile ransomware. In *international symposium on recent advances in intrusion detection*, pages 382–404. Springer, 2015.

- [13] Davide Maiorca, Francesco Mercaldo, Giorgio Giacinto, Corrado Aaron Visaggio, and Fabio Martinelli. R-packdroid: Api package-based characterization and detection of mobile ransomware. In *Proceedings of the symposium on applied computing*, pages 1718–1723, 2017.
- [14] Hossam Faris, Maria Habib, Iman Almomani, Mohammed Eshtay, and Ibrahim Aljarah. Optimizing extreme learning machines using chains of salps for efficient android ransomware detection. *Applied Sciences*, 10(11):3706, 2020.
- [15] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 252–276. Springer, 2017.
- [16] Jian Huang, Jun Xu, Xinyu Xing, Peng Liu, and Moinuddin K Qureshi. Flashguard: Leveraging intrinsic flash properties to defend against encryption ransomware. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2231–2244, 2017.
- [17] Eugene Kolodenker, William Koch, Gianluca Stringhini, and Manuel Egele. Paybreak: Defense against cryptographic ransomware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 599–611, 2017.
- [18] Kul Prasad Subedi, Daya Ram Budhathoki, Bo Chen, and Dipankar Dasgupta. Rds3: Ransomware defense strategy by using stealthily spare space. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8. IEEE, 2017.
- [19] Muhammet Baykara and Baran Sekin. A novel approach to ransomware: Designing a safe zone system. In *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, pages 1–5. IEEE, 2018.
- [20] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. Shieldfs: a self-healing, ransomware-aware filesystem. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 336–347, 2016.
- [21] PassMark PerformanceTest Tool. [https://play.google.com/store/apps/details?id=com.passmark.pt\\_mobile&hl=en\\_US&gl=US](https://play.google.com/store/apps/details?id=com.passmark.pt_mobile&hl=en_US&gl=US).
- [22] Robert Lipovský, Lukáš Štefanko, and Gabriel Braniša. The rise of android ransomware, 2016.
- [23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [25] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019.
- [26] Eric Vanderburg. The evolution of a cybercrime: A timeline of ransomware advances. <https://www.carbonite.com/blog/article/2017/08/the-evolution-of-a-cybercrime-a-timeline-of-ransomware-advances>, 2017.
- [27] Swati Khandelwal. LockerPin Ransomware Resets PIN and Permanently Locks Your SmartPhones. <https://thehackernews.com/2015/09/android-lock-ransomware.html>, 2015.
- [28] Nada Lachtar, Duha Ibdha, Hamza Khan, and Anys Bacha. Ransomshield: A visualization approach to defending mobile systems against ransomware. volume 26, page 30, New York, NY, USA, 2023. Association for Computing Machinery.
- [29] John Morris, Eli Lifland, Jin Yong Yoo, Jake Grigsby, Di Jin, and Yanjun Qi. Textattack: A framework for adversarial attacks, data augmentation, and adversarial training in nlp. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 119–126, 2020.
- [30] Hans Peter Luhn. A statistical approach to mechanized encoding and searching of literary information. *IBM Journal of research and development*, 1(4):309–317, 1957.
- [31] Catarina Silva and Bernardete Ribeiro. The importance of stop word removal on recall values in text categorization. In *Proceedings of the International Joint Conference on Neural Networks, 2003.*, volume 3, pages 1661–1666. IEEE, 2003.
- [32] Péter Halácsy and V Trón. Benefits of deep nlp-based lemmatization for information retrieval. In *CLEF (Working Notes)*, 2006.
- [33] Yoon Kim. Convolutional neural networks for sentence classification.
- [34] Fernando Pineda. Generalization of back propagation to recurrent and higher order neural networks. In *Neural information processing systems*, 1987.
- [35] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [36] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.
- [37] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.
- [38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.
- [39] VirusTotal. <https://www.virustotal.com>.
- [40] PaddleOCR. <https://github.com/PaddlePaddle/PaddleOCR>.
- [41] GoogleTrans. <https://pypi.org/project/googletrans/>.
- [42] Ji Gao, Jack Lanchantin, Mary Lou Soffa, and Yanjun Qi. Black-box generation of adversarial text sequences to evade deep learning classifiers. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 50–56. IEEE, 2018.
- [43] Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani Srivastava, and Kai-Wei Chang. Generating natural language adversarial examples. *arXiv preprint arXiv:1804.07998*, 2018.
- [44] Shuhuai Ren, Yihe Deng, Kun He, and Wanxiang Che. Generating natural language adversarial examples through probability weighted word saliency. In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pages 1085–1097, 2019.
- [45] Mohammad Hossein Samavatian, Anys Bacha, Li Zhou, and Radu Teodorescu. Rnnfast: An accelerator for recurrent neural networks using domain-wall memory. *J. Emerg. Technol. Comput. Syst.*, 16(4), September 2020.
- [46] Microsoft authenticator. [https://play.google.com/store/apps/details?id=com.azure.authenticator&hl=en\\_US](https://play.google.com/store/apps/details?id=com.azure.authenticator&hl=en_US).
- [47] Google find my device. [https://play.google.com/store/apps/details?id=com.google.android.apps.adm&hl=en\\_US&pli=1](https://play.google.com/store/apps/details?id=com.google.android.apps.adm&hl=en_US&pli=1).
- [48] Tesseract-OCR. <https://tesseract-ocr.github.io/>.
- [49] EasyOCR. <https://github.com/JaidedAI/EasyOCR>.
- [50] Rodrigo Wilhelmy. Captcha dataset version 2. <https://www.kaggle.com/datasets/fournierp/captcha-version-2-images>, 2013.
- [51] Cem Dilmegani. Ocr in 2024: Benchmarking text extraction/capture accuracy. <https://research.aimultiple.com/ocr-accuracy/>, 2024.
- [52] Daniele Sgandurra, Luis Muñoz-González, Rabih Mohsen, and Emil C Lupu. Automated dynamic analysis of ransomware: Benefits, limitations and use for detection. *arXiv preprint arXiv:1609.03020*, 2016.
- [53] Md Mahbub Hasan and Md Mahbubur Rahman. Ranshunt: A support vector machines based ransomware analysis framework with integrated feature set. In *2017 20th international conference of computer and information technology (ICCIT)*, pages 1–7. IEEE, 2017.
- [54] Bin Zhang, Wentao Xiao, Xi Xiao, Arun Kumar Sangaiah, Weizhe Zhang, and Jiajia Zhang. Ransomware classification using patch-based cnn and self-attention network on embedded n-grams of opcodes. *Future Generation Computer Systems*, 110:708–720, 2020.
- [55] Koodous community. <https://koodous.com/>, 2016.
- [56] Amirhossein Gharib and Ali Ghorbani. Dna-droid: A real-time android ransomware detection framework. In *International Conference on Network and System Security*, pages 184–198. Springer, 2017.
- [57] Contagio mobile malware mini-dump. <http://contagiominiidump.blogspot.it/>, 2016.
- [58] TonTon Hsien-De Huang and Hung-Yu Kao. R2-d2: Color-inspired convolutional neural network (cnn)-based android malware detections. In *2018 IEEE international conference on big data (big data)*, pages 2633–2642. IEEE, 2018.
- [59] Jing Chen, Chiheng Wang, Ziming Zhao, Kai Chen, Ruiying Du, and Gail-Joon Ahn. Uncovering the face of android ransomware: Characterization and real-time detection. *IEEE Transactions on Information Forensics and Security*, 13(5):1286–1300, 2017.
- [60] Gautam Sharma, Anubhav Johri, Anurag Goel, Anuradha Gupta, et al. Enhancing ransomwareelite app for detection of ransomware in android applications. In *2018 Eleventh International Conference on Contemporary Computing (IC3)*, pages 1–4. IEEE, 2018.

- [61] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, 15(1):83–97, 2016.
- [62] Davide Maiorca, Davide Ariu, Iginio Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.
- [63] Nada Lachtar, Duha Ibdah, and Anys Bacha. Toward mobile malware detection through convolutional neural networks. *IEEE Embedded Systems Letters*, 13(3):134–137, 2021.
- [64] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. Hardware performance counters can detect malware: Myth or fact? In *Proceedings of the 2018 on Asia conference on computer and communications security*, pages 457–468, 2018.
- [65] Abdulrahman Abu Elkhail, Nada Lachtar, Duha Ibdha, Rustam Aslam, Hamza Khan, Anys Bacha, and Hafiz Malik. Seamlessly safeguarding data against ransomware attacks. volume 20, pages 1–16, 2023.
- [66] N. Lachtar, D. Ibdah, and A. Bacha. The case for native instructions in the detection of mobile ransomware. *IEEE Letters of the Computer Society*, 2(2):16–19, June 2019.