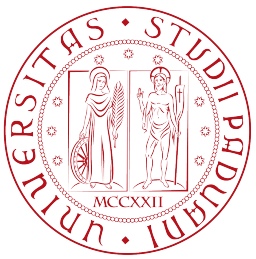


Università degli Studi di Padova

Dipartimento di Matematica
"Tullio Levi-Civita"

Corso di Laurea Magistrale in Informatica

PIN Inference on a Covered Hand Scenario: a Computer Vision Approach

Supervisor
Prof. Mauro Conti
University of Padua, Italy

Co-Supervisors
Dott. Matteo Cardaioli
University of Padua, Italy
GFT Italy, Italy
Dott. Stefano Cecconello
University of Padua, Italy

External Reviewer
Prof. Mark Manulis
University of Surrey, United Kingdom

Author
Eugen Saraci
1171697

September 2020

Sommario

Coprire la propria mano quando si digita il proprio codice PIN sul tastierino di un ATM è risaputo essere una buona tecnica per evitare truffe e furti di PIN; ciò infatti non permette al malintenzionato, o a un qualsiasi passante, di carpire le cifre che compongono il nostro codice segreto. Sebbene la nostra carta possa comunque essere clonata durante l'attacco, il malintenzionato non potrà effettuare alcuna operazione rilevante senza conoscere il PIN della stessa. La principale tecnica usata dai truffatori per rubare i PIN consiste nel nascondere, all'interno dell'ATM o in alcune sue componenti, una micro-camera diretta verso il tastierino, in modo da registrare tutto ciò che venga digitato dalle vittime. Tuttavia, se le vittime coprono attentamente la loro mano durante la digitazione, il malintenzionato non sarà in grado di capire quale sia il PIN digitato, dato che il tastierino non risulterebbe visibile alla micro-camera. In queste condizioni, per un essere umano risulta quasi impossibile riuscire a ricostruire il PIN digitato, tuttavia, utilizzando tecniche di *machine learning*, l'esito non è più scontato. In questa tesi, presentiamo un attacco innovativo che sfrutta la potenza delle reti neurali convoluzionali (CNN) per risalire al PIN digitato guardando solamente i movimenti della mano coperta, con una percentuale di successo pari a 1 PIN su 3, in uno scenario *user-independent*, in soli 3 tentativi.

Abstract

Covering your hand while entering a PIN on an Automated Teller Machine (ATM) is known to be good practice against card skimming attacks, as it prevents the attacker, or any onlooker, from learning the secret PIN, which is required to perform any operation on the ATM. The most common way the attackers steal PINs is by carefully concealing a small camera into the ATM components to record anything that is being typed on the keypad; the attacker can later watch the recorded footage to retrieve all the stolen PINs. However, if the victims carefully cover their hand while typing, the video footage will not be of any help to the attacker, as there would be no clear sight of the keypad. Therefore, recovering a PIN from this kind of footage would be close to impossible for the human attacker, however, if machine learning is involved in the process, the outcome is different. In this work, we present a novel attack that exploits the power of Convolutional Neural Networks (CNNs) to retrieve the victims' PIN just by looking at the movements of their covered hand, with a success rate of 1 PIN out of 3, in a user-independent scenario, when given only 3 attempts.

Contents

List of Figures

List of Tables

Listings

CHAPTER 1

Introduction

Personal Identification Numbers (or PINs) are still widely used as authentication schemes in many scenarios, such as ATMs, phone lock screens, or even door locks. Unfortunately, the action of entering a PIN is generally vulnerable to shoulder-surfing attacks [30]. While there are some countermeasures proposed in the literature to prevent this kind of attack [16, 19], a little effort is done to translate these defense mechanisms into practice. Some ATMs make use of a plastic shield around the keypad (Figure 1.1) to prevent precisely shoulder-surfing or pinhole camera attacks, moreover, for the same reason, customers are often taught to cover their hand while entering the PIN. However, while covering the hand will prevent the digits from being leaked, one might still try to analyze the hand movements made by the victim and infer which buttons are being pressed. Hand movements, in fact, have been shown to reveal critical information, especially when typing on a keyboard [4] or smartphone [25].

In the last decade, side-channel attacks have seen a surge in popularity thanks to machine learning techniques and tools being easy to use and affordable. Some of these attacks involve very precise video [2, 3] and audio [7, 9] analysis, which can only be performed by machine learning algorithms. In this work we analyze a scenario in which the attacker has placed a microcamera pointed at the keypad of an ATM, (this is commonly done for card skimming attacks); the microcamera films the victims while they enter their PIN on the keypad, however, we also assume that the victims adopt the aforementioned technique of covering their hand while

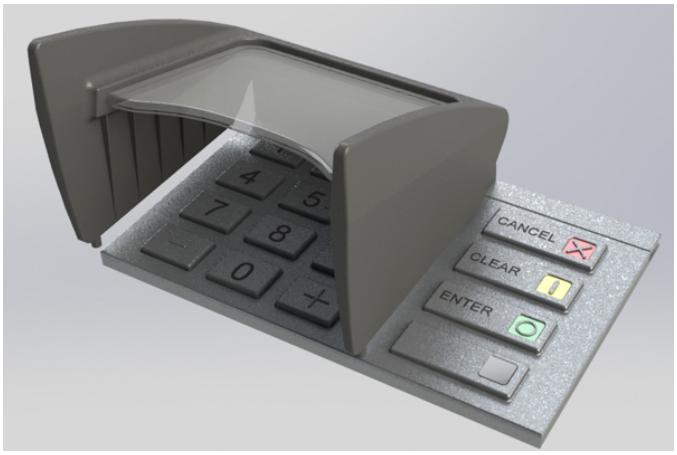


Figure 1.1: Plastic shield set around the keypad to prevent onlookers or hidden cameras from learning the PINs typed by customers. This image is taken from [32].

typing, thus preventing the microcamera from having a clear line of sight to the buttons of the keypad. The attacker can still analyze the recorded video, possibly remotely, in order to extract as much information as possible.

1.1 Card Skimming Attacks

Card-skimming is a category of attacks whose aim is to steal credit/debit card information from the victims. In this work, however, we only consider skimming attacks that target ATMs. These attacks usually required two different steps: (1) stealing the card information, and (2) stealing the PIN. The attacker accomplishes the first step by using a *skimmer*, a small device hidden in the card tray that reads the data of the cards as the victim inserts it. Sometime attackers will build a fake card tray almost indistinguishable from the real one and put it on top of the real card tray. In this way, it is much easier for the attacker to hide a skimmer inside the fake card reader (Figure 1.2).

The attacker generally accomplishes the second step in two possible ways, either by hiding a small pinhole camera in the ATM (Figure 1.3) or by putting a fake keypad on top of the real one (Figure 1.4). The pinhole camera points at the keypad and it records the victims' hand while entering the PIN, while the keypad overlay simply saves all keypresses. The attack is successful only if both steps reach

Figure 1.2: The figure shows a fake card tray with a skimmer inside it (the electronic chip). When the fake card tray is put in place, it is not easy to spot unless the victim is very careful to details. This image is taken from [15].

their objectives. The only real practical countermeasure against the keypad and card tray overlay is to pay attention to oddities in those components, while for the pinhole camera it is usually suggested to cover the keypad while typing (either with the other hand or the wallet).

Contributions This work aims at testing whether the movements made by the typing hand while being covered leak enough information for a deep learning model to understand which key is being pressed. To this end, we build an ATM replica to collect data and asked 43 volunteers to take part in our experiments. We asked each volunteer to type 100 randomly generated 5-digit PINs onto our fake ATM while the keypad was being recorded by a webcam. Participants were told to cover their typing hand while entering the PINs so that the webcam could not have a clear vision of the buttons being pressed. After cleaning the dataset, we trained a deep learning model with 70% of the data collected, being careful to partition the data in such a way that the model would never see, during training, some of the users. The idea is that, in this way, we can test the performance of the model in a user-independent context.

The final evaluation was performed on a total of 400 PINs typed by 4 users. The model managed to retrieve 22% of the PINs on the first attempt, and it reached

Figure 1.3: The red circle highlights a very small hole from which a pinhole camera was recording the keypad. This image is taken from [15].

Figure 1.4: The fake keypad on top of the real one logs all the keypresses. This image is taken from [15].

36% accuracy when given 3 attempts. After 20 attempts the model reached 50% accuracy, however, we deem the TOP-3 accuracy to be the most relevant metric to the attacker in this context, as ATMs only allow 3 PIN attempts before locking or disabling the card.

Organization The remainder of this work is organized as follows:

- Chapter 2 presents some of the related work, both from the security and deep learning areas;
- Chapter 3 introduces some preliminary notions about the attack and the main aspects of deep learning used throughout this work;
- Chapter 4 formalizes the attacker model;
- Chapter 5 explains the whole data collection process.
- Chapter 6 presents the models developed for this attack and the different approaches adopted for them;
- Chapter 7 shows the evaluation process and results of the models;
- Chapter 8 contains some final remarks about this work and proposes some extensions for this attack.

CHAPTER 2

Related Work

This chapter presents some of the related work dividing them in two categories. First we report some general side-channel attacks, then we cover only PIN and PIN pads attacks.

2.1 Side-Channel Attacks

Side-channel attacks are those attacks that specifically target the information gained by the *implementation* of a system rather than its logic. Most of the time, these attacks exploit channels like sound, timing, energy consumption, and electromagnetic emanations to learn the secrets of the system in use. The main cause behind the feasibility of side-channel attacks is the lack of attention to the details that do not directly concern the system, i.e., while much effort may be put into the verifying the logic and the cryptographic security of an algorithm, little to no effort is allotted to prevent attacks such as, e.g., timing attacks on the instructions executed by the algorithm. In [14], for example, the authors managed to crack RSA keys by carefully timing the operations performed by the key-generating algorithm. Another example of a timing attack is reported in [27], where the authors measured the timing between keystrokes in interactive SSH sessions, in an attempt to retrieve the typed passwords. In [1] the authors exploit the sound made a printer when printing a text document; without a priori knowledge on the context of the document, the authors managed to retrieve more than 70% of the text contained in it, just by analyzing the audio during the printing process. Another example

of an *acoustic* attack comes from [9], where the authors showed that they were able to infer which keys were being pressed by the victim, just by having a Skype call with them and recording the audio of the keystrokes. Notice that also human behavior can be defined as a side-channel of a system, especially if the analyzed behavior is a direct result of the system's requirements. For example, in [4], the authors analyze the hand movements of people typing on a keyboard and, by using basic computer vision techniques, they try to infer reconstruct the text being typed. In [26], the authors again analyze the hand motion during the PIN-entry process on smartphones. They showed that 50% of the PINs could be retrieved in just 1 attempt even when the keyboard is not visible. In [26] they also analyze the ATM scenario, which is substantially different from our approach, as, in their model, the PIN pad and the fingers of the victim are partially visible to the camera. In [29] the authors present a side-channel attack on tablets, which consists of analyzing the backside movements of the tablet itself to infer what is being typed by the victim. To do so, they select some peculiar features of the backside of the tablets (e.g. logos, side-buttons) and analyze their movement throughout the frames, to understand which area of the virtual keyboard is being pressed. Another interesting side-channel attack is reported in [10], where, by analyzing the encrypted traffic of some of the most common Android applications (e.g., Facebook, Twitter, Gmail), the authors are able to understand which actions are performed by the victims with a 99% accuracy. While this attack does not reveal the content of a specific action, it allows the attacker to know whether, e.g., the victim is sending an e-mail or posting a message on Facebook wall, which, through a correlation attack, might lead to the full de-anonymization of an online profile.

2.2 PIN and PIN pads attacks.

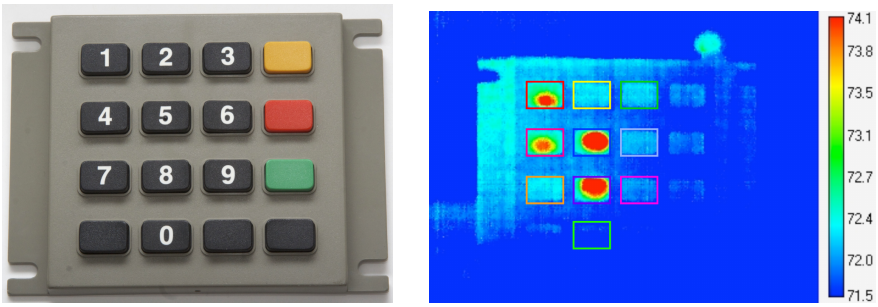
In the following section, we describe some of the most interesting side-channel attacks on PINs and PIN pads we found in the literature.

The first attack is performed on the PIN pad and it exploits the heat that is transferred from the hand to the keypad when the victim enters her PIN [22]. In

this attack, as soon as the victim has finished entering her PIN, the attacker points a thermal camera to the keypad; the thermal image not only shows which keys have been pressed but it also highlights the order in which the victim pressed them (Figure 2.1). The main advantage of this attack is that it does not require the attacker to do anything *while* the victim is typing her PIN, even though the attacker must act quickly (i.e., within seconds) for a higher success rate, as the heat on the keypad rapidly fades away (between 1 and 2 minutes). Another drawback of the attack is that its effectiveness depends on the material of the keypad, e.g., metal PIN pads completely nullify the attack because of their high thermal conductivity.

The second attack is a timing attack against PINs. In the scenario presented in [2], the attacker record the screen of an AMT while the victim is entering her PIN. When analyzing the recorded video, the attacker exploits the PIN masking symbols appearing on the ATM screen to extract precise timing information about the keystrokes. This allows the attacker to use a machine learning model to infer, starting from the inter-keystroke timing, which buttons (i.e. two consecutive buttons) were most likely typed by the victim. In Figure 2.2, we see how the attack would be performed in a real scenario.

The third and last attack is a timing/acoustic attack on PIN pads. In [7], the authors used the artificial sound made by the ATM whenever a button is pressed. The sound produced by ATM must be constant independently of which button is being pressed (to comply with the standard ISO 9564-1). This means one single "beep" will not help the attacker, however, the sound gives enough information to extract a timestamp of the keys being pressed, which allows the attacker to perform the timing attack reported also in [2].



(a) The plastic keypad used throughout the experiments in [22].

(b) Thermal camera view of the keypad after a victim has typed her PIN. The original PIN was 1485, which is also suggested by the heat localized in those buttons. The colored rectangles around the keys are just used to highlight the position of the buttons. This image is taken from [22].

Figure 2.1: The thermal camera detects which buttons have been pressed.

Figure 2.2: The left image shows clearly how the attacker can record the ATM screen while the victim is typing. On the right, we see the view from a bystander.

CHAPTER 3

Background Knowledge

In this chapter, we introduce some of the main features regarding Convolutional Neural Networks. We mainly focus on those features that we manipulated the most.

3.1 CNN - Convolutional Neural Networks

Convolutional neural networks (CNNs) are a kind of neural network mainly employed for applications in visual imagery. The main difference between CNN and traditional Multi-Layer Perceptrons (MLPs) is that CNNs employ at least one convolutional layer. In a convolutional layer, the main operation between inputs and weights is a linear operation named convolution, while in a fully connected layer the main operation is the matrix multiplication. The weights of the convolutional layer are called *filters* (or *kernels*, or *feature detectors*) and they all must have the same shape of the input. So, if the input is a 32×32 grayscale image, then the filters must also be two-dimensional, but no constraints are posed on their size (which usually is 3×3).

The number of filters of a convolutional layer constitutes a hyper-parameter of the model, therefore there is no clear way to decide how many filters should be instantiated in a single convolutional layer. Each filter is applied, through the convolution operation, to the input image as many times as it fits, and the result of this operation is a representation of the original image (possibly with a smaller size). For example, Figure 3 shows the result of a convolution between a 3×4 input image and a 2×2 filter. The filter is applied to the image starting from the top

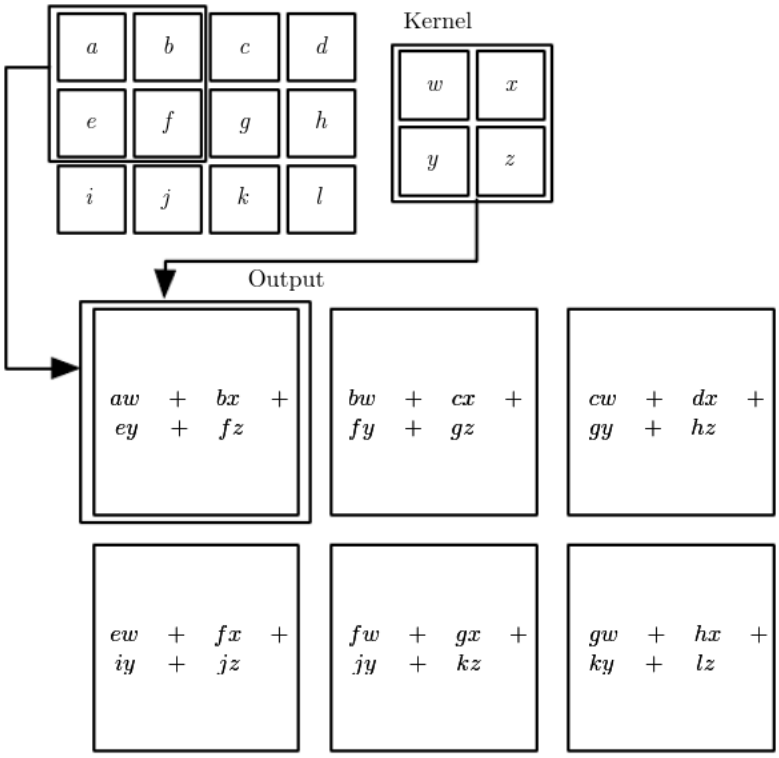


Figure 3.1: Example of a convolution between a 3×4 input image and a 2×2 filter. Each square in the output area represents a pixel of the output image. This image is taken from [13].

left corner to the bottom right, shifting each time by 1 pixel. The final result is a 2×3 image whose pixel values are the linear combination of the inputs.

Notice that in this basic example, the final image is smaller in size than the starting one and it will keep shrinking if more convolutional layers follow. To prevent the image from shrinking too much, we might make use of a technique called *zero-padding*. With zero-padding, before each convolutional layer, each side of the input image is enwidened with p columns and rows of black pixels (i.e., pixels with value 0), so that the resulting image does not suffer too much the dimensionality reduction. Notice, however, that p, the size of the padding, must be regulated based on the filter size, as both these elements determine the final output size. If the final image has the same size as the input image, then we call this a *same* convolution (Figure 3.2).

The final component that influences the size of the output image is the *stride*. The stride is the number of pixels in the step size of the filter, i.e., how much (in

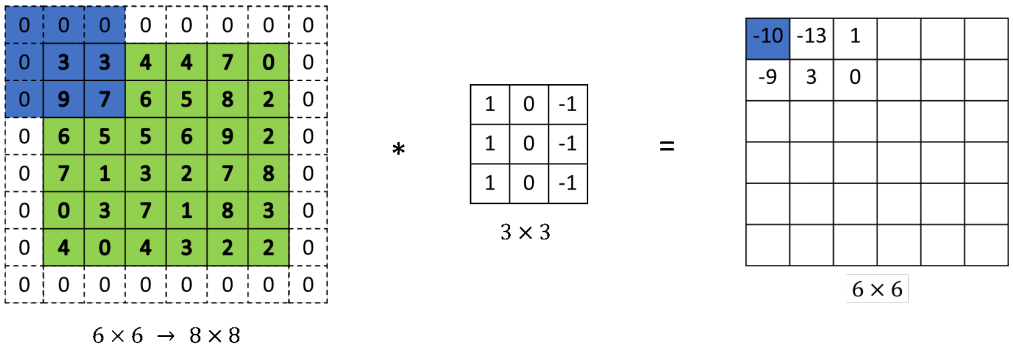


Figure 3.2: Zero-padding with $p = 1$ applied to a 6×6 input picture. When the 3×3 kernel is applied to the input image, the output image does not shrink, instead it keeps the *same* size. This image is taken from [8].

pixels) the filter shifts to the right (or down) with respect to the image after a convolution. The formula to compute the size of the output image is the following:

$$\left\lfloor \frac{m + 2p - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor$$

were m and n are the height and width of the input image; p is the size of the padding; f is the size of one side of the filter (we are assuming that the filter is squared), and s is the size of the stride. If we take the parameters from Figure 3.1 and apply those to the formula, we have that $m = 3$, $n = 4$, $p = 0$, $f = 2$, $s = 1$ which results in:

$$\left\lfloor \frac{3 + 2 * 0 - 2}{1} + 1 \right\rfloor \times \left\lfloor \frac{4 + 2 * 0 - 2}{1} + 1 \right\rfloor = 2 \times 3.$$

The main advantage of a convolutional network with respect to the traditional Multi-Layer Perceptron is efficiency. As stated before, in fully connected layers, the main operation is the matrix multiplication between the input and the weights of the hidden layers. If we have a 128×128 grayscale image and a fully connected layer of 100 units, we need to train $(128 * 128 + 1) * 100 = 1,638,500$[1] weights just on the first layer. With a convolutional layer, we can use 32 filters of size 3×3 and still get fewer weights to train (i.e., $(3 * 3 + 1) * 32 = 320$[2]). This happens because of two characteristics of the convolutional layers: *sparse connectivity* and *parameter*

[1]The +1 is needed to account for the *bias* of each hidden unit.
[2]The +1 is needed to account for the *bias* of each filter.

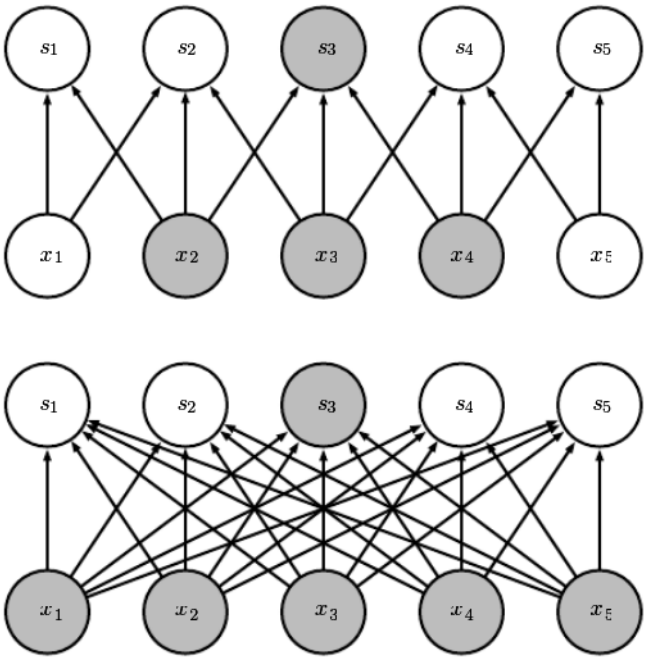


Figure 3.3: The image shows how the connections (i.e., weights) required for a fully connected layer (bottom) are many more than those required by a convolutional layer (top). The image is showing the connection between the inputs x_i and the outputs s_i; the black arrows represent the weights. Take, for example, the output unit s_3, we can see that in a convolution operation, s_3 only depends on the values of $x_{2,3,4}$, while, in a dense matrix multiplication, it depends on all the inputs $x_{1,\dots,5}$. This image is taken from [13].

sharing.

Sparse Connectivity Sparse connectivity refers to the fact that any output unit of the convolutional layer only depends on a subset of units of the input layer; this is not true in the case of fully connected layers, where, as the name suggests, each output units is dependent on all input units. Figure 3.3 shows an example of this.

Parameter Sharing Parameter sharing (or *tied weights*) refers to the fact that when computing the output of a convolutional layer, the weights of a filter are reused multiple times as it shits across the input image. In fully connected layers, a single weight is only used once throughout all the computation of the output. This peculiar feature of the convolutional layers implies that the weights of the filters do

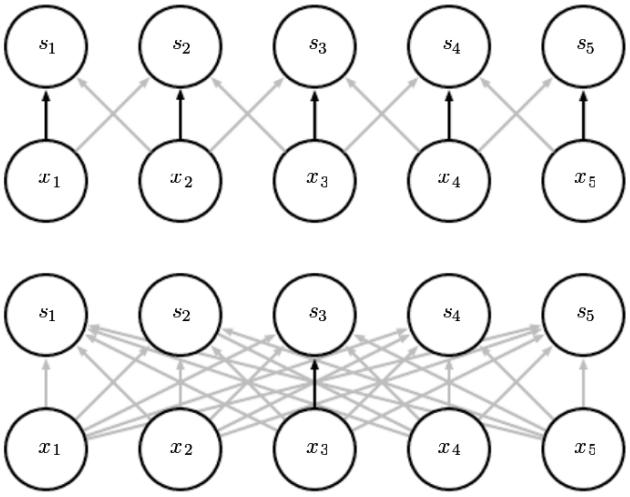


Figure 3.4: The image shows how the weights (arrows) used in a convolutional layer (top) are shared (black arrows) among different input units, while, in a fully connected layer (bottom), weights are used exactly once.

not change based on where they are applied to the image; i.e., the weight applied to one input is also applied to other input units elsewhere in the image. This works well with visual imagery, as it implies that a filter that detects a particular feature (e.g., vertical edges) needs only to be learned once, and it will be independent of where it is applied to the input image (this property is called *equivariance* to translation). Figure 3.4 shows an example.

3.1.1 Typical CNN Architecture

Convolutional neural networks, as stated before, employ convolutional layers as their main component, however, it is not rare to find fully connected layers too in other parts of the network. Fully connected layers are usually positioned at the end of the network, where all the feature extracted by the filters get flattened into a 1D array and then provided to such layer. Figure 3.6 shows an example of a typical CNN architecture.

Another important type of layer used in convolutional networks is the so called *Pooling* Layer. The pooling layer is usually inserted after one or multiple consecutive convolutional layers; its main task is to reduce the of parameters of the network by shrinking the image. To do so, the pooling layer uses one single filter (usually of

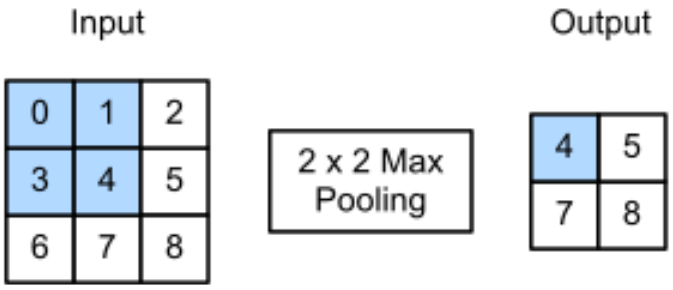


Figure 3.5: Example of the application of a 2×2 max pooling filter to a 3×3 input. This image is taken from [24].

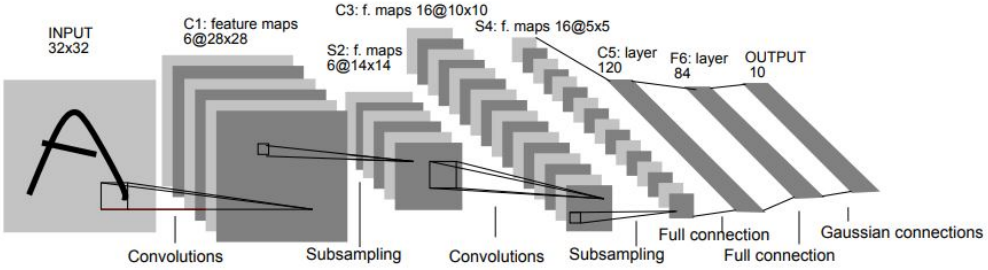


Figure 3.6: Typical CNN architecture that employs convolutional layers, pooling layers, and fully connected layers. Used in [18].

size 2×2) to scan the input image and produce at each step the output value which is obtained either by taking the maximum (*max pooling*, Figure 3.5), the average (*average pooling*) or the L2-norm (*L2-norm pooling*) among the input values (if the pooling filter has size 2×2, then the number of inputs is 4). Notice how even though a filter-like structure is used, the pooling filter does not have any trainable weight, in fact, the designer can only choose what kind of function to use when applying the pooling filter. Some authors think that the pooling layers only add to the complexity of the architecture without providing a real benefit from the functionality standpoint; in [28], the authors argue that the same effect produced by the pooling layers can be obtained by carefully setting the stride parameter on the convolutional layers.

CHAPTER 4

System Model

In this chapter, we formalize the attack scenario and the attacker himself. In this way, it should be clear what conditions are necessary for the attack to take place.

4.1 System Model

The attack is performed when a victim interacts with the keypad of an *unsafe* ATM and types her PIN in it. The ATM is considered to be *unsafe* when the attacker places a hidden webcam on hit to spy on the victim's actions. The attacker aims at learning the victim's PIN.

We make no assumptions about the type of webcam used by the attacker, but we assume that it can easily be hidden in an ATM and that it is the only way the attacker can manipulate the ATM. Figure 4.1 shows how small these cameras can get, additionally, if disassembled, the components can be organized to fit in very tight places.

We assume that the victim has no way to detect that an ATM is unsafe, but she still adopts basic countermeasures against card-skimming attacks, such as covering her hand while typing.

The attacker does not need to be there when the victim types her PIN, as he can freely have access to the recorded video of the webcam, either remotely, or at a different time. The attacker is able to retrieve the timestamps in which the victim has typed the single keys on the keypad, and he can do so by listening at the audio of the video recording. There are two different types of sound clues that can be

Figure 4.1: Example of a pinhole camera. The camera is usually disassembled to fit in tighter spaces.

exploited by the attacker: the first one is the *beep* sound made by the keypad when a key is pressed[1]; the second one is the sound of the physical button of the keypad being pressed. External noise does not prevent the attacker from extracting they keypresses, as the webcam is still very close to the keypad, therefore the sound can still be identified in the audio track.

If for any reason, the attacker has no way to retrieve the timestamps from the recorded audio (or if there is no audio at all), the attacker could place the camera in such a way that both the keypad and the screen of the ATM are visible: this allows him to extract the timing of the keypresses by looking at the PIN masking symbols appearing on the screen [3]. Common masking symbols are usually dots ($\bullet$) and asterisks ($*$).

Figure 4.2 shows the steps needed for the attack to occur.

[1]Not all keypads make this sound.

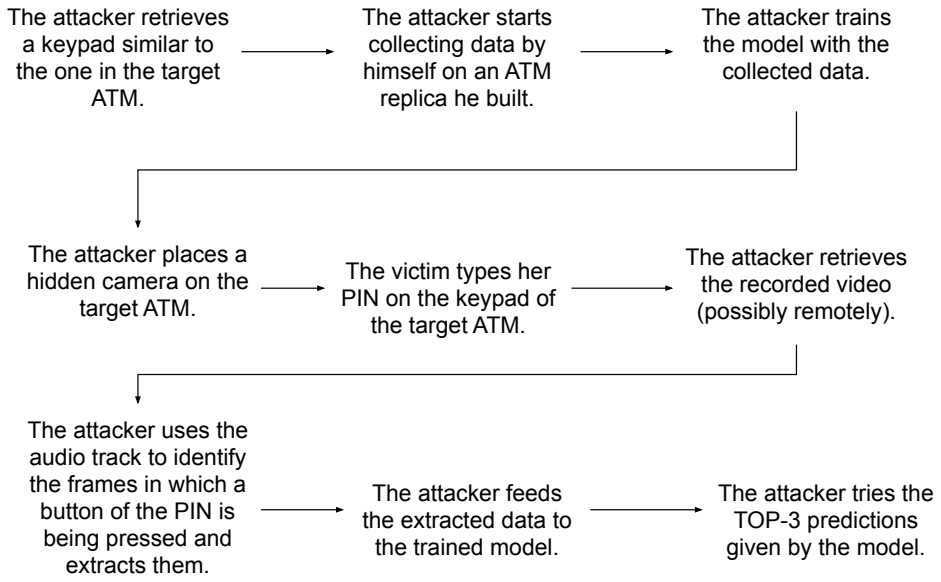


Figure 4.2: The attack shown step-by-step. The data collection process does not necessarily need to happen before the attacker steals the victim's PIN, however, it is a required step of the attack.

CHAPTER 5

Data Collection

In this chapter, we present the main aspects of the experimental setup and how the data collection was performed. In the last two sections, we describe the cleaning steps required to produce the final dataset used for the experiments. The data collection was performed inside one of the seminar rooms of the Department of Mathematics "Tullio Levi-Civita" at the University of Padua, during the first two weeks of July 2020.

5.1 Experimental Setup

The experiment setup is aimed at replicating the *interaction* between users and real ATMs as closely as possible, and to do so we try to preserve the same proportions (e.g., height from ground) found in publicly available ATMs. The most relevant aspects of our replica are therefore the position and appearance of the keypad, while the looks of the replica as a whole are not important in this context. The objective of the data collection process is to create a dataset of images in which a button of the keypad is begin pressed.

5.1.1 ATM Replica and Keypad

The ATM replica is simply a wooden frame with a monitor and a keypad in it. The wooden frame has a width of 60 cm, a height of 64 cm, and a depth of 40 cm [5]; at 15 cm of height from the frame's base, we inserted a wooden shelf on which we later positioned the keypad and the monitor. On the monitor, we display the randomly

(a) ATM replica.

(b) Keypad closeup. This image is taken from [21].

Figure 5.1: The ATM replica and a closeup of the keypad.

generated PIN and the text box on which the participant has to type it in. The keypad is connected to a PC using a USB cable, and it behaves exactly like a USB keyboard, which makes it easier to log the keys being pressed. The participants are instructed to only use the numeric keys and the `enter` button to submit the PIN. Figure 5.1a shows a closeup of the ATM replica, while Figure 5.1b shows a closeup on the keypad.

To keep the data collection as consistent as possible throughout the days, during the experiments, all of the blinds inside the room were closed, while all of the lights were turned on. The ATM replica was placed against the wall, and it was heavy enough to prevent any accidental movement by the participants during the experiments. In the unlikely case of the replica being moved in-between experiments, the images collected by the webcams would not be affected by it, as the webcams were solidly attached to the frame.

5.1.2 Webcams

To record the typing process, we set up three Logitech C920 HD Pro webcams [20]. The first one was positioned centrally, above the keypad, while the other two were set respectively in the top-left and top-right corner of the wooden frame; all three cameras were pointing at the keypad. To firmly attach the webcams to the frame,

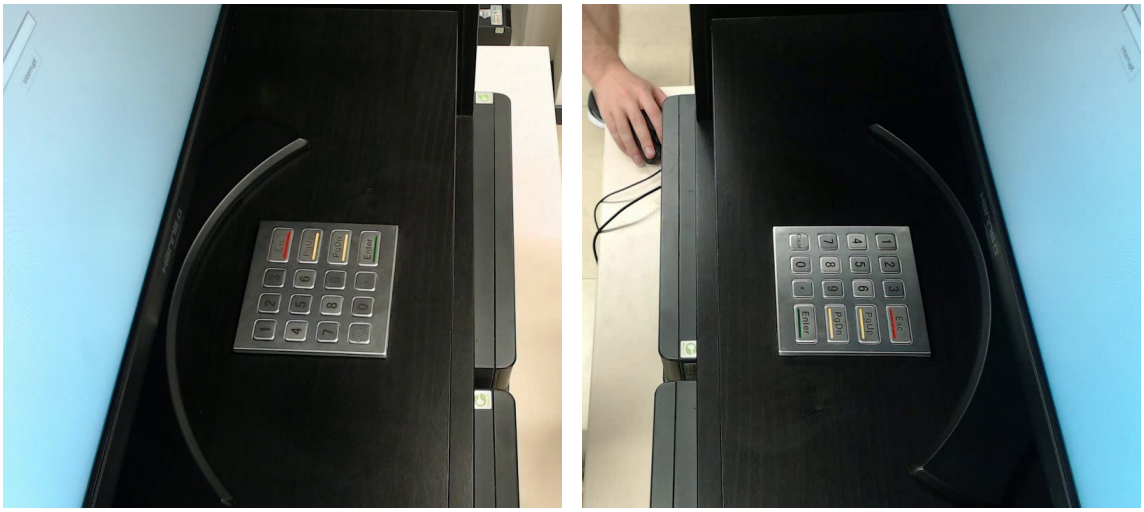
Figure 5.2: Central webcam view.

we used three clamps.

The initial idea was to have three different points of view (Figures 5.2, 5.3) so that we could evaluate which webcam, or a combination thereof, performed best for this attack scenario. However, as of this writing, only the central webcam has been tested thoroughly, while the top-left and top-right cameras have been employed only for some preliminary experiments, which is why the attack itself only involves one camera.

5.2 Experiment Process

Because of COVID-19 regulations, only 3 people were allowed to be in the testing room at the same time, but to avoid any kind of issues, we limited that number to 2: I and whichever candidate was scheduled for that time slot. Face masks were (and still are) mandatory inside the department building, additionally, every one was asked to use the hand sanitizer gel we provided before and after typing on the keypad; the keypad itself was also cleaned and disinfected after each experiment. Moreover, for the same reasons, we could only look for participants among the university employees, which heavily limited the amount of data we could collect



(a) View from the top-left webcam. **(b)** View from the top-right webcam.

Figure 5.3: Lateral webcams view.

during those days. Eventually, we managed to find 43 people who agreed to take part in the experiment.

Every participant was asked to type 100 randomly generated 5-digits PINs, and to make it easier for them, we allowed them to take a short break after each session of 25 PINs. We also required the participants to submit each PIN by pressing the `enter` key, which would allow them to skip to the next randomly generated PIN; on average, for the whole process, participants would take between 11 and 14 minutes to type all 100 PINs, including the short breaks, depending on how fast they would type. All participants were also instructed to carefully cover their hand while typing, which is a common technique used against shoulder surfing attacks and maliciously placed micro-cameras. Some participants were not familiar with this typing technique, but after being shown how it was done, they all learned quickly how to replicate it. Figure 5.4 shows the effectiveness of covering the typing hand.

5.2.1 Error Handling

The chosen error handling approach was to allow participants to make mistakes while typing, i.e., no error message was shown on screen if the submitted PIN did

(a) View from the top-left webcam.

(b) View from the central webcam.

(c) View from the top-right webcam.

Figure 5.4: Participants covering their hands while typing.

not match the PIN on the screen. Although this might sound counterproductive, we argue that it is not: in this context, what we really care about is the position of the hands in the exact moment a button is being pressed; it does not matter if the pressed button is correct with respect to the displayed PIN, as long as we know the actual button that has been pressed. In fact, the length of the PIN itself was not enforced by the software, this means that, possibly, some participants may have inadvertently submitted PINs with more or less than 5 digits.

5.3 Data Logging

To log all the keypresses coming from the keypad, a keylogger was installed on the control machine: when a new experiment started, all three cameras start recording, and, at the same time, the keylogger starts saving all the keypresses. After 25 PINs, the webcams and the keylogger save all the collected data to disk. The webcams save one video recording[1] each, while the keylogger saves six different files, two for each camera. A part of the first logfile is shown in Listing 5.1: the first value of each line represents the label of the key being pressed; the second value can either be `d` or `u`, and it represents respectively the `KeyDown` or the `KeyUp` event for that key; the last value shows (in seconds) when the event occurred with respect to the

[1] At 30 fps, in 720p.

beginning of the recording (slightly different for each camera). Part of the second file is shown in Listing 5.2: in this case, there is only one value, which is the frame number (of the video) on which the event occurred. This means that the event listed on the i-th line of the first logfile occurred on the frame number found on the i-th line of the second logfile.

```
key,    event,  timestamp
5,      d,      11.057425
5,      u,      11.097378
0,      d,      11.297442
0,      u,      11.345712
enter,  d,      11.665385
enter,  u,      11.713443
6,      d,      14.225381
6,      u,      14.265386
```

Listing 5.1: Keys, events, and timestamps recored by the keylogger and saved in first logfile for the central webcam.

```
frame_no
337
338
344
346
355
357
432
433
```

Listing 5.2: Frame numbers of the events reported in Listing 5.1, saved in the second logfile for the central webcam.

5.3.1 Data Cleaning

Unfortunately, of the 43 participants, 3 were left-handed; the final decision was to remove them from our dataset, as we deemed the amount of data not to be enough for the task. Keeping them in would also mean that the dataset would be heavily unbalanced in favor of the right-handed people, an undesirable property

that may skew the evaluation results. Nonetheless, we claim that if we were to have a left-handed-only dataset with the same amount of samples as the "right-handed-only" dataset, the results showed in Chapter 7 could easily be achieved by the same models presented in Chapter 6, as there is no substantial difference between the two tasks.

5.3.2 Preparing the dataset

As we were not satisfied with the structure and content of the two logfiles, we decided to write a Python script to merge them and add some additional data to the rows. Listing 5.3 shows how a whole PIN is represented in the "new" CSV dataset (header row is included for clarity). One immediate difference that can be noticed is that there is no more a distinction between `KeyUp` rows and `KeyDown` rows, additionally, the `enter` key has been ignored. Here follows a description of the new fields:

- `target`: the frame number of the `KeyDown` event of the target key (i.e., the key being pressed);
- `start`: the frame number of the `KeyUp` event of the previous key;
- `end`: the frame number of the `KeyDown` event of the following key;
- `t1`: time passed (in milliseconds) between frame `start` and frame `target`;
- `t2`: time passed (in milliseconds) between frames `target` and frame `end`;
- `case`: edge case flag; can assume values `0`, `1`, and `2`:
  - `0` means that there is no edge case;
  - `1` means that the target key is the first digit of a PIN, which also implies that there is no previous key, and therefore `start` must be equal to `target`;
  - `2` means that the target key is the last digit of a PIN, which also implies that there is no following key, and therefore `start` must be equal to `target`;

- `sid`: unique identifier of the session (i.e., block of 25 PINs);
- `uid`: unique identifier of the user (i.e., full name of the participant);
- `key`: the key being pressed.

```
target, start, end, t1,   t2,   case, sid,   uid,   key
750,    750,   765, 0.00, 0.50, 1,    48a7b, Eugen, 0
765,    751,   822, 0.46, 1.91, 0,    48a7b, Eugen, 3
822,    767,   831, 1.86, 0.29, 0,    48a7b, Eugen, 7
831,    824,   841, 0.25, 0.32, 0,    48a7b, Eugen, 8
841,    832,   841, 0.28, 0.00, 2,    48a7b, Eugen, 5
```

Listing 5.3: PIN representation in the new dataset.

At the end of this process, the number of rows in the final dataset was 20054. Notice that one row corresponds to just one keypress, in fact, in the final dataset, we look at data only as single keypresses and not as 5-digit PINs; the concept of PIN is reintroduced later, during the final testing phase. Figure 5.5 shows the same data, graphically.

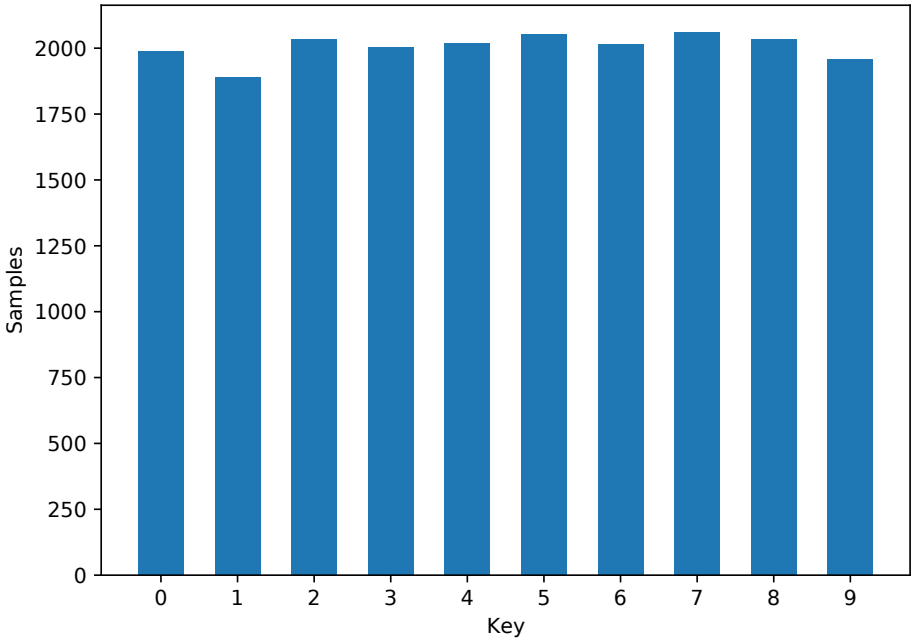


Figure 5.5: Amount of samples for each key.

The final step of the data collection process was to use the just mentioned CSV dataset to extract and save all the needed images to disk using *FFmpeg* [31], so that we could easily load it later for our deep learning models. However, we noticed that there was no need for the data to be saved on disk first, as, by using *OpenCV* [6], we could load the needed images in memory by extracting them directly from the video source.

CHAPTER 6

Models and Experiments

In this chapter, we present the models and the different approaches explored during the development of the attack. Here, we also show some of the practical challenges we had to deal with before even starting with the training phase.

6.1 Base model

The aim of the model that we developed is to determine which button is being pressed, given a sequence of images as input. Since we had to deal with both sequences and images (or more simply, videos), the first design choice that came to mind, was to use Convolutional Neural Networks (CNNs) [17], and Long Short-Term Memory Cells (LSTM) [12]. The convolutional part would work as a feature extractor for each frame of the sequence (e.g. detecting hands, fingers, keypad), while the LSTM part would detect movement and patterns through time. Additionally, to classify the result, we inserted a fully connected layer, followed by a 10-units output layer with a `softmax` activation function, in order to have a probability distribution over all the buttons. From now on, for simplicity, we will be referring to the CNN part as the "feature extractor" of the model, while the LSTM layers, the fully connected layers, and the output layer will be referred to as the "classifier" of the model. The resulting model is known in the literature as *Long-term Recurrent Convolutional Network* (LRCN) [11]. In Keras, such architecture can be implemented using the `TimeDistributed` wrapper throughout all the layers of the feature extractor, which causes the same convolutional filters to be applied to all the timesteps (or frames)

of the input sequence. Alternatively, the naive (and highly inefficient) approach would be to use different "flows" of convolutions for each frame, essentially creating n independent feature extractors that possibly detect unrelated features among the frames of the same sequence.

The feature extractor of the model has 4 convolutional layers (`Conv2D` in Keras) each followed by a pooling layer (`MaxPooling2D` in Keras). Each convolutional layer has a filter size of 3×3, while each pooling layer has a filter size of 2×2. The number of filters in the convolutional layers doubles up at each level, starting from 32 filters in the first layer, ending up at 256 filters in the fourth (and last) layer. The classifier has an LSTM layer with 128 units, followed by a fully connected layer of the same size and an output layer of size 10, which is the number of classes we are trying to classify. In the classifier, in the input of the fully connected layer, we also added the timing information found in the dataset, however, this aspect is discussed in §6.6. Other specific parameters of the model are discussed in §6.7, while Figure 6.1 shows a global overview of the model.

6.2 Dataset Partitioning

Before starting with the experiments, we divided the data into three partitions: training set, validation set, and test set. The percentages we chose for those partitions are 70% for the training set, 20% for the validation set, and 10% for the testing set. The basic approach to creating these sets would be to randomly assign samples from the dataset, however, this would make the attack user-dependent. In this context, a user-dependent attack is not realistic, as it would require the attacker to collect prior data about the typing behavior of the victims (including keypresses), and use that data to target the same victims for a future attack; this is an extremely unlikely scenario because there is a high chance of the attacker retrieving the PIN already during the data collection. From the point of view of the neural network, a user-dependent attack would mean that the model only learns the specific typing behaviors of the users, and therefore is able to correctly classify the keypresses during validation and testing just because it knows how the specific

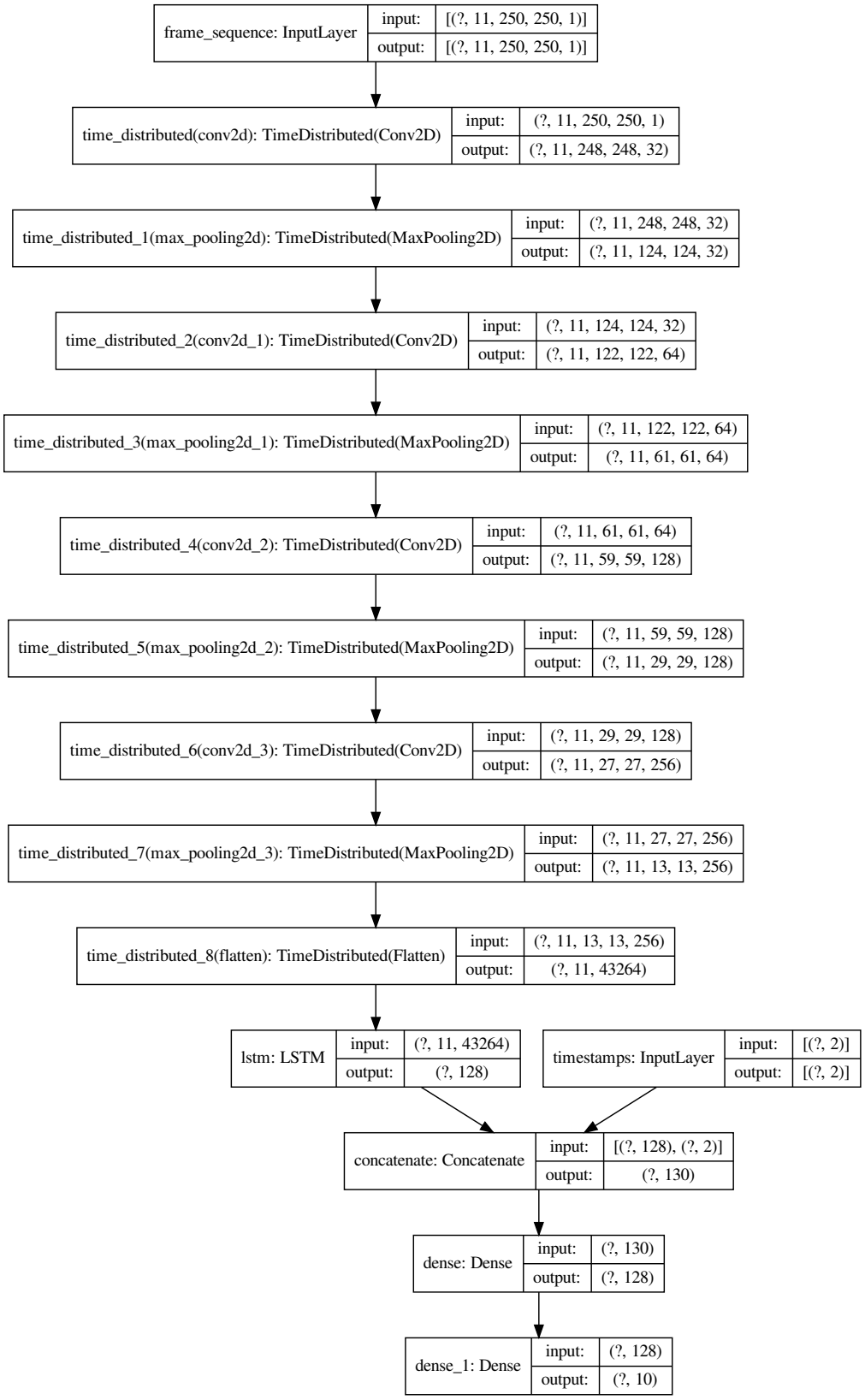


Figure 6.1: Graphical summary of the base model.

Table 6.1: Detailed amount of samples for each key with user-independent partitioning.

Key	0	1	2	3	4	5	6	7	8	9
Training	1377	1288	1402	1399	1426	1446	1438	1451	1455	1371
Validation	395	415	422	397	398	412	389	402	378	390
Testing	215	187	209	206	195	196	189	207	201	198

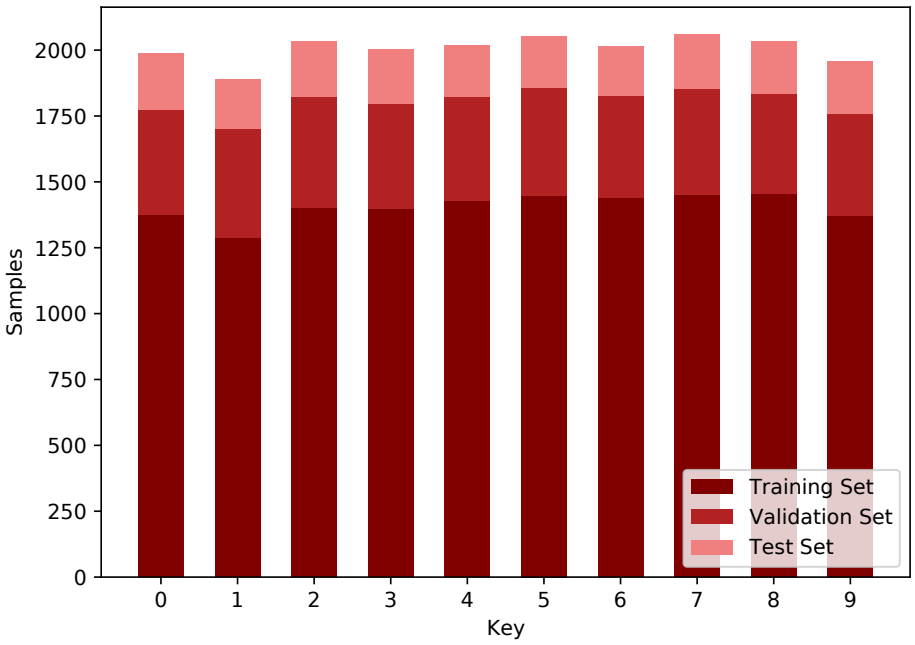


Figure 6.2: Amount of samples for each set and for each key with the user-independent partitioning.

users type. What the attacker wants is to train the model once, possibly by himself, and use it later on new victims, i.e., he wants the attack to be *user-independent*. For the attack to be user-independent, we partitioned the dataset in such a way to have samples from the same user only be assigned in one of the three sets; samples from 28 users ended up in the training set; samples from 8 users ended up in the validation set; the remaining 4 users ended up in the testing set. In this way, the model will be evaluated on users (and typing behaviors) it has never seen during training. Table 6.1 and Figure 6.2 show that the final distribution of the labels is still balanced throughout all the sets, even with the user-independent partitioning.

6.3 Preprocessing

Before providing the samples to the network, we apply a preprocessing pipeline; preprocessing is usually used to standardize the input samples. Some common preprocessing steps (in the computer vision context) are usually: cropping, rescaling, normalizing, changing color. The idea is that by making all the samples as structurally similar as possible, we reduce the amount of variation of the dataset; by doing this we may not need to employ complex models and solutions, which is desirable, as a simple and small solution usually generalizes better than a big and complex one. The preprocessing pipeline we implemented has the following steps:

1. converting the image to `grayscale`, as we were more interested in the geometry of the image rather than the colors;
2. normalizing the input so that all pixel values lie in $[0, 1]$;
3. cropping the image by cutting off the irrelevant part of the picture. This was easy to do as the camera had a fixed position and therefore never moved;
4. resizing the image; since the cropping step was already generating images with the same size, this step could have been avoided, however, we wanted the images to have squared shape, and therefore we resized them all to be 250×250 pixels.

6.4 Data Augmentation

To prevent the model from overfitting the data, we decided to implement a data augmentation pipeline. Data augmentation is the practice of applying small random graphical transformations to the input images of the training set, with the objective of improving the generalization of the model. Data augmentation also serves as way to virtually increase the number of available training samples; in fact by slightly changing the input, we are essentially creating a new sample. The data augmentation in this specific implementation is performed *on-the-fly*, i.e., no data is

Figure 6.3: Effects of data augmentation. The original image is the one in the top-left corner, all the others are augmented versions of it.

saved on disk, but transformations are applied before the sample is given as input to the network. Possibly, thanks to the randomness in the graphical transformations, the network might never see the same image twice throughout all the training epochs.

In my configuration, data augmentation was set with the following parameters:

- rotation: for a maximum of 10° both clockwise and counterclockwise
- horizontal shift: for a maximum of 10% of the width;
- vertical shift: for a maximum of 10% of the height;
- zoom: between 0.9 and 1.1.

In Figure 6.3 the image in the top-left corner is the original sample, while all the other images architecture augmented versions of it. Notice that the transformations had to be relatively small to prevent the hands and/or the keypad from getting out of bounds in the resulting image.

As we was dealing with sequences of images and not single frames, we had to implement the data augmentation pipeline in a different way than the usual one. When a sample is given to the pipeline, we first generate a seed number for the operations that require randomness, after that, for each frame of the sequence, since we fix the seed, we are able to apply the same exact augmentation that will be applied to the other frames of the same sequence. When a new sample follows, a new random seed is generated and the process repeats.

6.5 Frames per sample

Since the model requires all input samples to have the same length, the first task we had to do was to decide which frames and how many of them should be kept for each sample. The first attempt at setting the number of frames was done by following what was already available in the dataset (see §5.3.2), i.e., the frame numbers of the previous and following keypresses with respect to the target keypress. The initial approach was exactly this: given a target keypress, keep all the frames between the `keyup` event of the previous keypress, and the `keydown` event of the following keypress (we will call this set of frames the *full-neighborhood* of the target keypress). This approach, unfortunately, did not work well in practice, in fact, it turned out to be very computationally expensive, to the point where it was infeasible with the configuration reported in §6.7. The reason is that this way of equalizing the samples is heavily influenced by the outliers of the dataset, i.e., even though the average number of frames of the full-neighborhood per sample is 17, the sample with the longest full-neighborhood had 291 frames, which meant that all samples should have been extended to have exactly that number of frames.

To find a suitable number of frames, we had to perform a statistical analysis of the dataset. As stated before, the average number of frames in the full-neighborhood is 17, however, this value too is influenced by the outliers, therefore we decided to only keep those elements whose length of the full-neighborhood was within 3 standard deviations from the mean.[1] Figure 6.4 shows the distribution of the number of

[1] We only removed the outliers from the computation of the average number of frames; no data

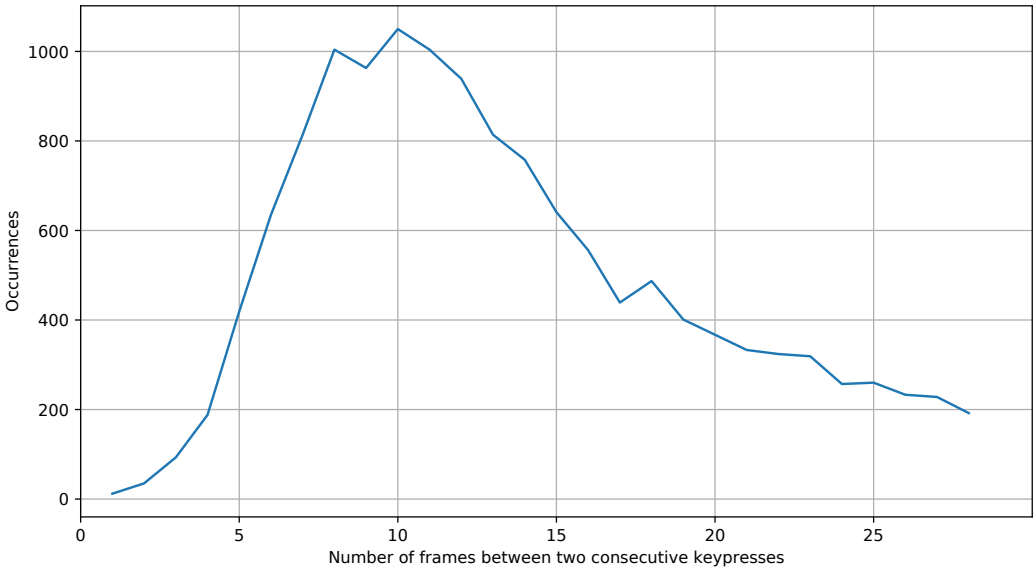


Figure 6.4: Distributions of the number of frames between the `keyup` event of the previous keypress and the `keydown` event of the following keypress with respect to the target keypress (after rejecting outliers). E.g., given the trigraph "345" where "4" is the target keypress, we count the number of frames between the `keyup` event of the key "3" and the `keydown` event of the key "5".

frames after rejecting the outliers; the average is now 12, while the maximum is 28.

We then decided to work with the new average value, which meant that, when loaded, the samples would all have at most 12 frames each. However, since we still wanted the target keypress to be in the center of the sequence, we decided to keep only the 5 frames preceding the target keypress and the 5 frames succeeding it, for a total of 11 frames per sample (including the target frame), which is close enough to the new average. For example, assume that the target keypress is located at frame number 10, assume now that the `keyup` of the previous keypress is located at frame number 3, and that the `keydown` of the following keypress is located at frame number 20, the just presented approach would only keep frames between frame numbers 5 and 15 (included), as shown in Figure 6.5. Iwewill call this the *neighborhood* (of size 5) of the target keypress.

was removed from the actual dataset.

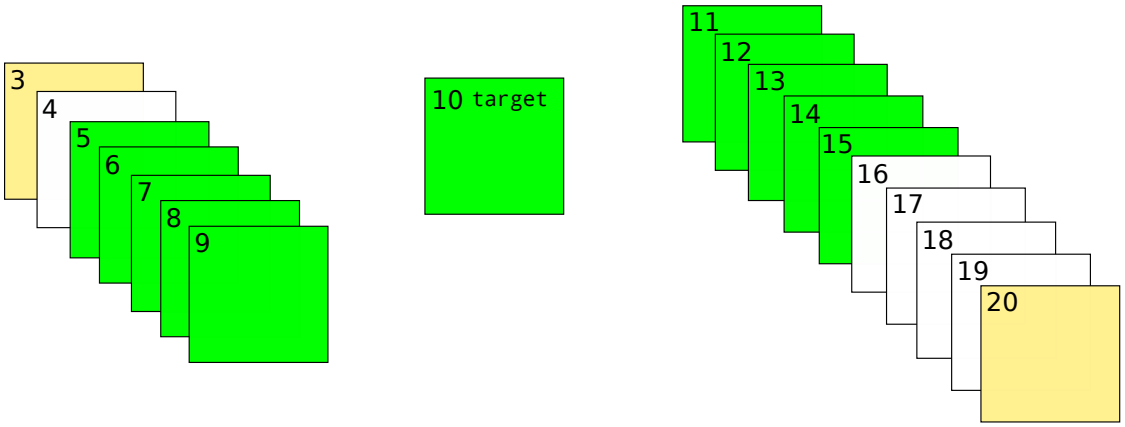


Figure 6.5: The neighborhood of size 5 of the target keypress is highlighted in green. White frames are too far away from the target frame and are therefore discarded. Yellow frames mark the `keyup` event of the previous key (frame number 3) and the `down` event of the following key (frame number 20).

After having decided the number of frames to keep, we thought about some alternative approaches on *which* frames to keep. One possible solution was to perform subsampling, i.e., taking equally spaced frames throughout the full-neighborhood (Figure 6.6). This does not change the final amount of frames, but the produced sequence provides a different kind of perspective. Unfortunately, this approach is also sensible to outliers, as samples with a long full-neighborhood would become sequences of seemingly unrelated frames. This also highlights one of the hidden advantages of keeping only frames that are near the target keypress: the distance in time between two keypresses is irrelevant if we only look at a small neighborhood of frames around the target.

Recall that, as seen in Listing 5.3, when the keypress refers to the first or the last key of the PIN, there are no previous and following keypresses respectively. This means that there might be samples where the target key is positioned at the beginning or at the end of the frame sequence, or even that there are samples with less than 11 frames. To prevent this from happening, we use black frames to extend short sequences, therefore, if the target keypress does not have, for example, a previous keypress, we add 5 black frames to fill the gap; if the previous keypress exists, but it happens to be too "early" in time (i.e., there are less than 5 frames between that keypress and the target keypress), then we add the needed amount of

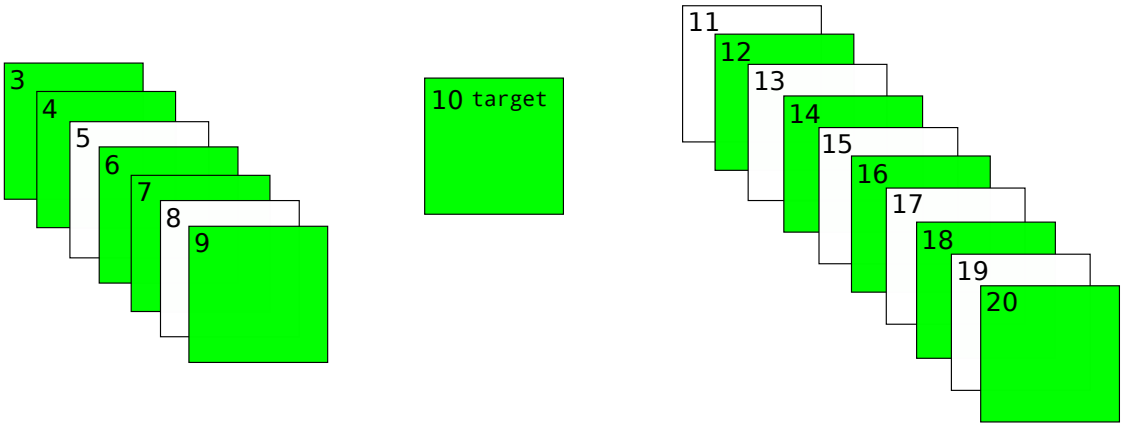


Figure 6.6: Keeping frames that are equally spaced throughout the sequence. Green frames are kept, white are discarded.

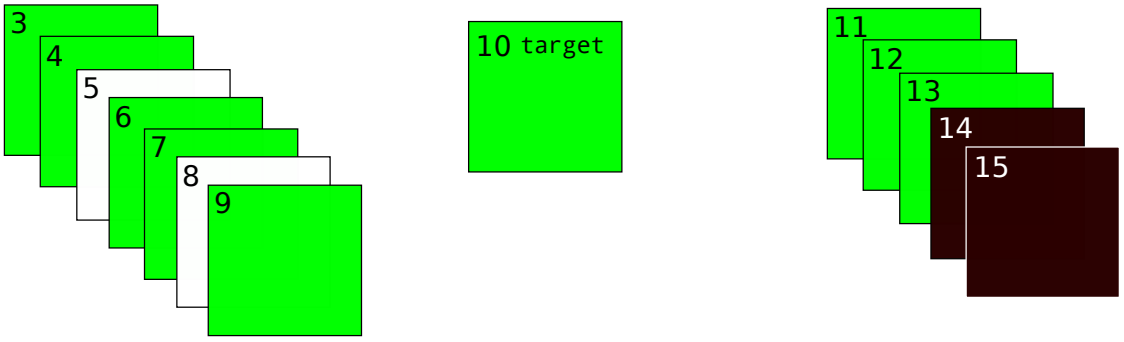


Figure 6.7: Selected frames (green) and padded frames (black) of the considered video sequence between subsequent keypresses.

black frames to fill the gap. This technique is shown in Figure 6.7.

Figure 6.8 shows how the subsampling gives no real advantage to the model, instead, as expected, it only makes the task is harder in the first epochs. For this reason, we decided to discard the subsampling option from the model.

6.6 Timing information

In §6.1, we introduced the concept of using the timing information as an added feature before performing the final classification. Figure 6.1 also displays this in one of the last layers where a `concatenate` layer is used to concatenate the 128 features of LSTM output with the 2 inputs coming from the timestamp before feeding it to the fully connected layer. To clarify, if the target PIN is “12345” and

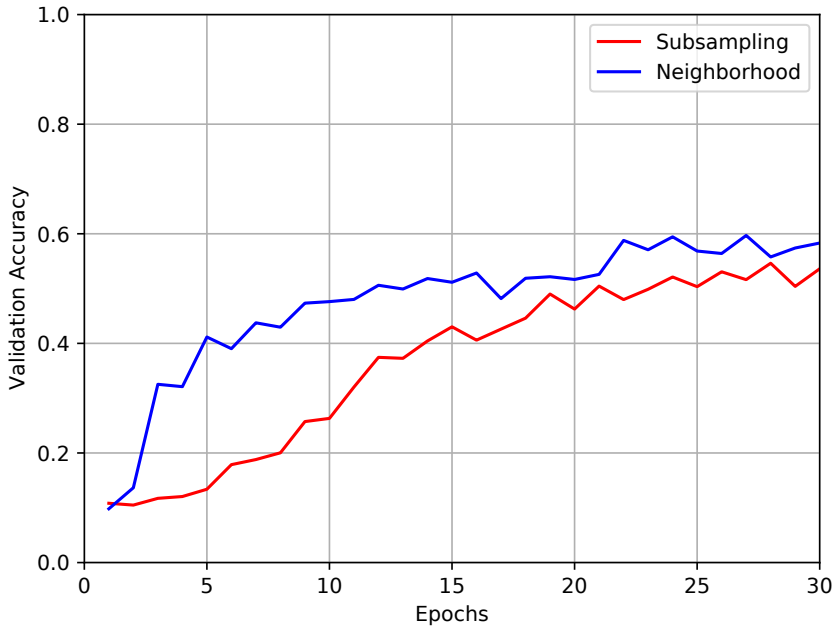


Figure 6.8: Difference in validation accuracy between the subsampling approach and the neighborhood approach (both with 11 frames in total).

the target key is "3", then the timing inputs would be:

- the timing between the `keyup` event of the previous key ("2") and the `keydown` event of the target key ("3"), i.e., the value of the field named `t1` in the dataset;
- the timing between the `keydown` event of the target key ("3") and the `keydown` event of the following key ("4"), i.e., the value of the field named `t2` in the dataset.

Notice that, as explained in §6.5, there are samples that do not have a previous or a following keypress. In these cases, the values of columns `t1` or `t2` are equal to 0, which we used as placeholder value to indicate this phenomenon. we also evaluated if and how much the inclusion of the timing info would affect the validation accuracy: it turns out, as shown in Figure 6.9, that adding the timing info does not really influence the accuracy of the model. However, because of compatibility reasons with the already existing data structures, we decided to keep timing inputs, but in

order to make it irrelevant for the predictions, we always use 0 as a value for both inputs for all samples.

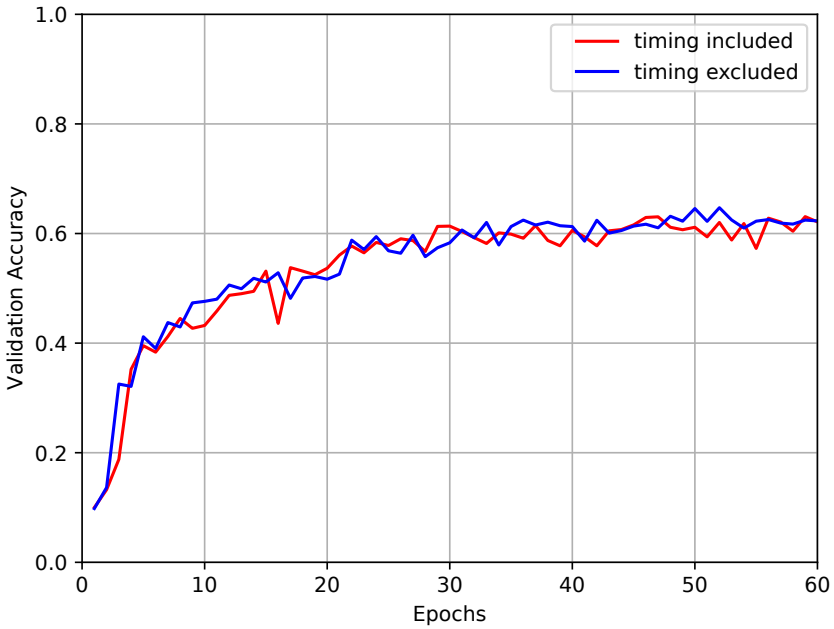


Figure 6.9: Difference in validation accuracy when including the timing information as input.

6.7 Configuration and Environment

The model is trained using Stochastic Gradient Descent (SGD) as the optimizer, while the chosen loss metric is the Categorical Cross Entropy, which works well with the SoftMax activation function of the output layer. The batch size is set to 16 samples, while the number of training epochs was generally set to different values based on what we were experimenting, in any case, for most of the models presented here the number of training epochs is 60, but the validation loss is used as a criterion for checkpointing.

The hardware on which we trained this model is the following:

- **CPU**: Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz, with 8 cores and a cache size of 20 MB;
- **GPU**: NVIDIA Tesla K20m, with 5 GB of DDR5 memory;

  - **NVIDIA driver**: 418.87.01
  - **CUDA driver**: 10.1

- **RAM**: 128 GiB
- **DISK**: 18 TiB

CHAPTER 7

Evaluation

In this chapter, we present the chosen model and we show its results during the testing phase. First, we show some of the most common classification metrics, then we also analyze some interesting patterns in the errors made by the model. In the first part of the chapter we only analyze the performance on single keys, while in the second part, we also evaluate the model on whole PINs.

7.1 Choosing the model

After trying different combination of approaches and different values for the hyper-parameters, this is the configuration that generally produced the best performing models:

- Data augmentation active;
- No timing included;
- No subsampling;
- SGD optimizer, and Categorical Cross Entropy as the loss metric;
- 60 epochs of training;
- User-independent dataset partitioning.

Figure 7.1 shows the performance of the model on the validation set; The best model is the one with the smallest validation loss, i.e., the one that can be seen at

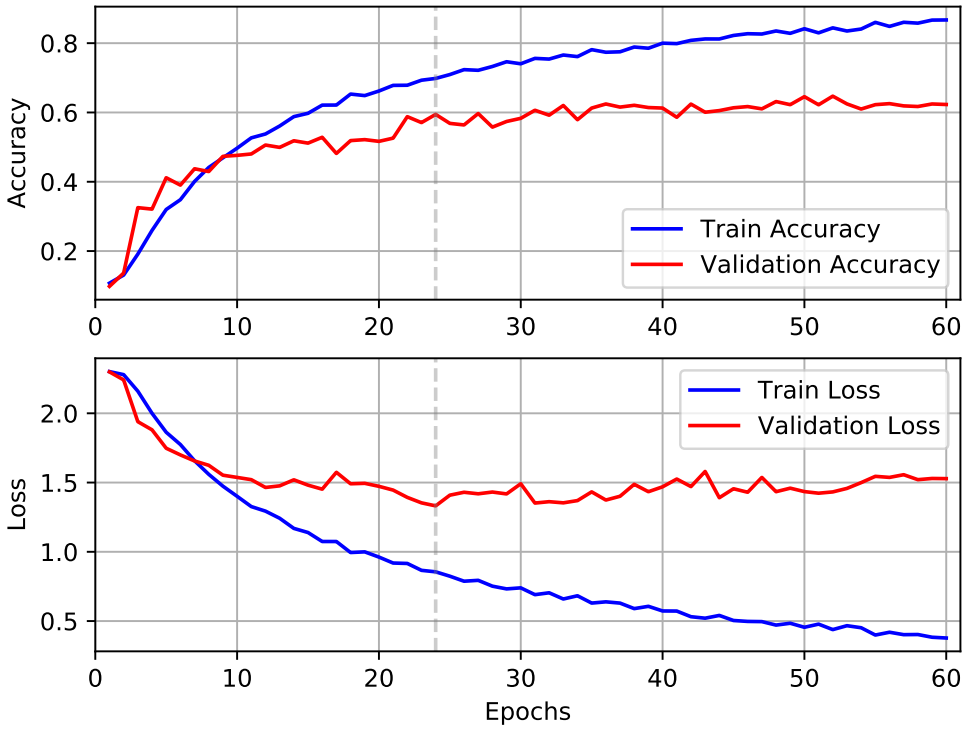


Figure 7.1: Validation and training performance comparison throughout the training epochs. The vertical dashed gray line shows the epoch on which the model performed best (i.e., lowest validation loss).

epoch 24 (highlighted with a dashed gray line). This is the model with which we carried out the testing phase.

7.2 Evaluation on single keys

Table 7.1 displays the performance of the model in some of the typical metrics used for classification tasks, while Figure 7.2 shows the normalized confusion matrix obtained by our model. The confusion matrix clearly highlights a pattern in the error distribution, in fact, one can notice that most of the prediction errors are not randomly distributed but are usually close (physically) to the target key. To better visualize this relationship, look at Figure 7.3, which displays in a different form the distribution of the predictions for key "1". Here we can immediately see how the model, even if it predicts the wrong key, it almost always manages to guess the

Table 7.1: Classification report produced by the Scikit-learn library [23]. These results refer to the test set and show different metrics for each class. The classes correspond to the pressed buttons.

class	precision	recall	f1-score	support
0	0.59	0.69	0.64	215
1	0.63	0.79	0.70	187
2	0.47	0.58	0.52	209
3	0.70	0.65	0.67	206
4	0.65	0.56	0.60	195
5	0.51	0.51	0.51	196
6	0.57	0.48	0.52	189
7	0.70	0.67	0.68	207
8	0.50	0.43	0.46	201
9	0.66	0.56	0.60	198
accuracy			0.59	2003
macro avg	0.60	0.59	0.59	2003
weighted avg	0.60	0.59	0.59	2003

right area in which the target key has been pressed.

7.3 Evaluation on PINs

The previous section analyzed the performance of the model on single keys, however, if we want to know how the model performs on PINs, some small changes need to be applied. The model only accepts single keys, so there is no automatic way to retrieve the PIN accuracy, therefore we need to develop the code for it. The first step of the process is to generate the PINs: luckily, we already have real PINs typed by the participants, so, to make this as realistic as possible, we can use the

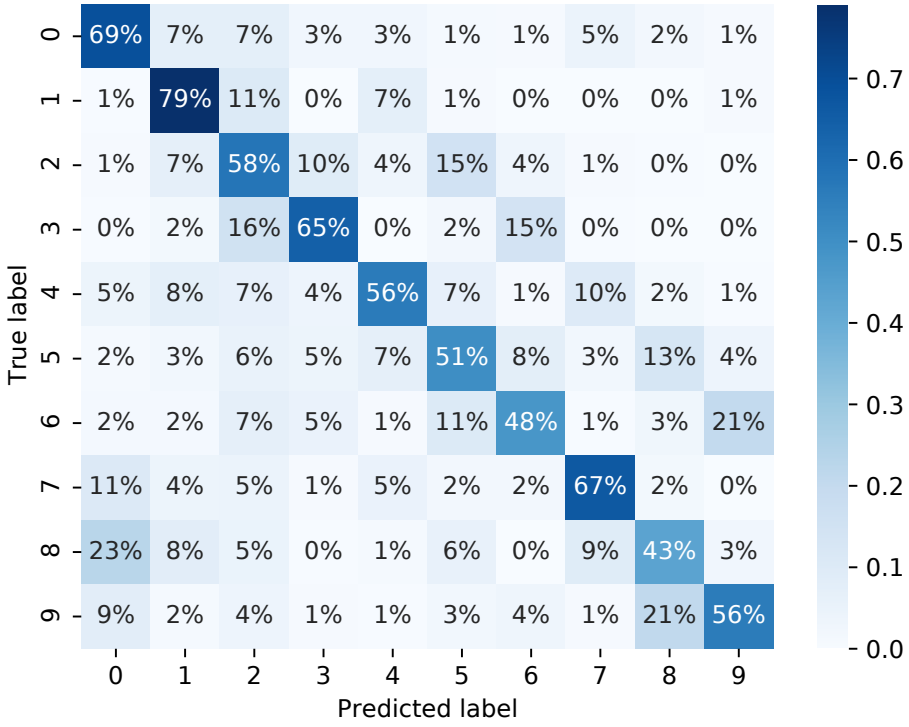


Figure 7.2: Normalized confusion matrix on the test set.

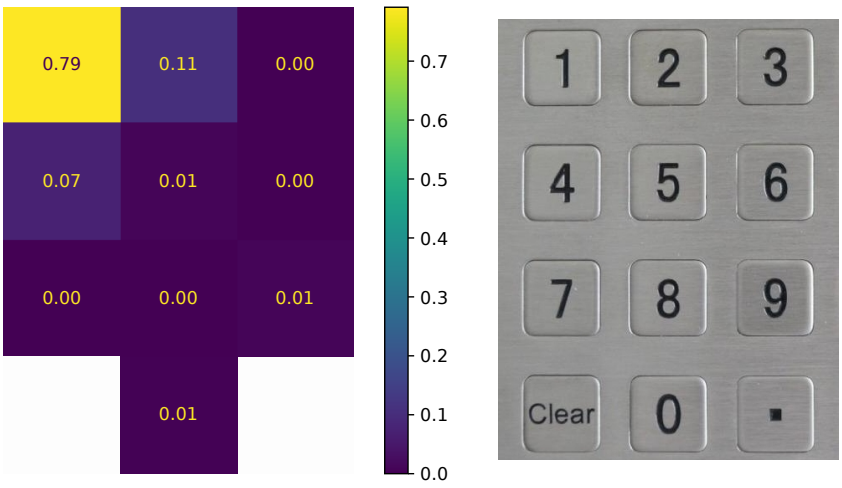


Figure 7.3: Distribution of the predictions for key "1"; most of the prediction errors are made on buttons that are physically close to the target button.

real sequences.[1] Now that we have a valid PIN, we feed the 5 keys to the network (separately); the network will output, for each given key, a probability distribution over the 10 labels, so we will now have, for each key, a list of 10 probabilities; we can thus compute all the possible combinations of keypresses and their probability as a whole for the given input. The result of this process is the full lists of all possible 5-digit PINs and their probability with respect to the input; by sorting this list by the probability value, we can lookup our original PIN and check at which position it is stored By repeating this process for all the PINs in the test set (400 PINs) we can evaluate how well the model performs. The main metric in this context is surely the TOP-3 accuracy, i.e., how many PINs end up in TOP-3 positions of the sorted list. Usually, ATMs give customers only 3 attempts for the PIN, therefore, metrics like TOP-5, or TOP-10 are not really relevant in this case.

The final result is as follows:

- **TOP-1**: 22%;
- **TOP-2**: 32%;
- **TOP-3**: 36%;

The results show that, on average, the model manages to put in the TOP-3 positions 1 PIN out of 3, which is a good result for a user-independent approach. The same result is shown graphically in the Cumulative Distribution Function graph (CDF) in Figure 7.4.

[1] The less realistic, but still valid, approach would be to build PINs by using random keys typed by different users.

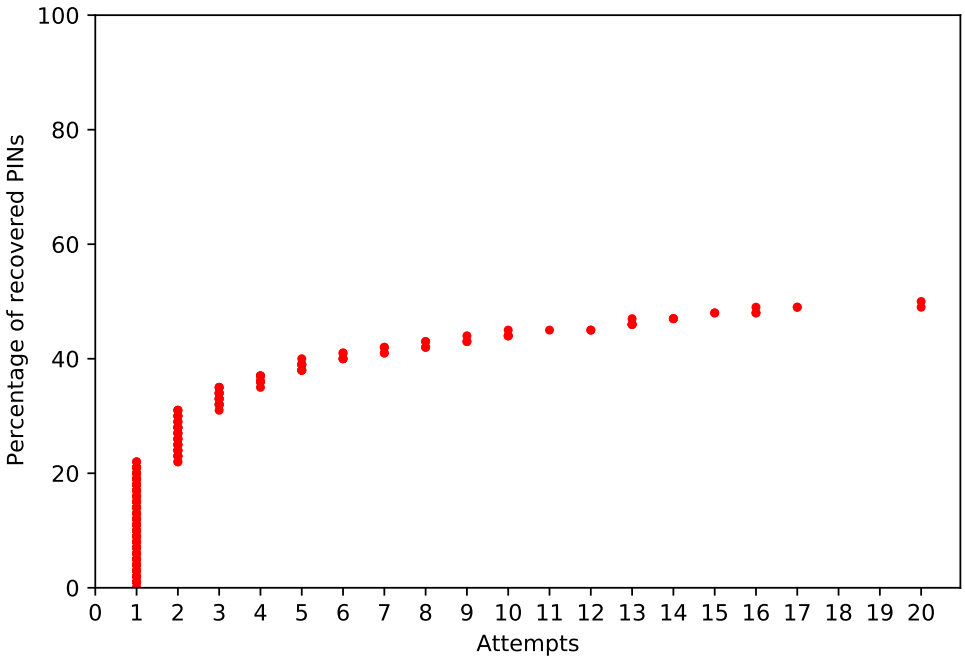


Figure 7.4: CDF showing the percentage of the PINs cracked during the testing phase; the model can to recover 36% of PINs in only 3 attempts.

CHAPTER 8

Conclusions and Future Work

This chapter presents the final remarks about the work done in my master thesis and provides some previews of possible extensions of the attack for the future.

8.1 Overall Summary

In this work, we developed and tested a deep learning model that evaluates the efficacy of a common and widespread countermeasure used against card-skimming attacks. The model is able to retrieve a victim's PIN just by looking at their covered hand when typing it on the keypad, and it does so without requiring any prior knowledge about the victim or their typing behavior. The trained model is in fact user-independent, which is a huge advantage to the attackers, since it allows them to collect the training data in a controlled environment (ATM replicas), and, after training the model, to use it in real attacks (real ATMs).

The results, unfortunately, show that the "covered hand technique" is not that effective when artificial intelligence is involved, in fact, the model manages to retrieve, on average, 1 PIN out of 5 in only 1 attempt, and 1 PIN out of 3 in at most 3 attempts. While users should still be covering their hand while typing, the results reported in this work highlight the inadequacy of current authentication methods regarding PIN and keypads. The more AI becomes widespread and easy to use, the more it will be exploited by malicious actors for day to day attacks; it is important, therefore, to be prepared when this will inevitably happen.

8.2 Future Work

This work provides many ways in which it can be extended. First of all, as mentioned in §5.1.2, the number of webcams set up in the ATM replica is three; the initial idea of this attack was in fact to use videos coming from three different points of view and to feed them to an ensemble of neural networks. In fact, for right-handed people, the right camera, might be able to reach a higher accuracy score than the central camera, and the inverse might be true for left-handed people, but, most important of all, using predictions from all three cameras might improve the overall accuracy of the model. Luckily, we do still have the videos collected by all three cameras, but the neural networks have not been developed nor tested yet.

In my opinion, the main flaw of this model is the lack of data. Unfortunately, the data collection was performed during the month of July 2020, during the COVID-19 pandemic, which meant that the number of people available for the experiments was extremely low for a deep learning model. While the model still behaved well, we have no way to tell if the model has reached its limit, or if this is just a baseline. If possible, in the near future, we will try to collect more data from more participants; hopefully, this will give us a way to test how powerful the model can be.

Moreover, while the model is user-independent, it is not *keypad-independent*, i.e., we do not know if and how well the attack would work when applied on a different keypad other than the one we used for our experiments (Figure 5.1b). It would be nice, as a future extension, to retrieve different keypads and test whether the model still reaches the same accuracy level.

Lastly, as mentioned in Chapter 1, some ATMs are employing plastic shields around the keypad to prevent hidden webcams and onlookers to learn the PINs. It would be interesting to test the effectiveness of the different types of shields and see how they compare to the model presented in this work.

Bibliography

[1] Michael Backes et al. “Acoustic Side-Channel Attacks on Printers.” In: *USENIX Security symposium*. 2010, pp. 307–322 (cit. on p. 7).

[2] Kiran Balagani et al. “PILOT: Password and PIN information leakage from obfuscated typing videos”. In: *Journal of Computer Security* 27.4 (2019), pp. 405–425 (cit. on pp. 1, 9).

[3] Kiran S Balagani et al. “Silk-tv: Secret information leakage from keystroke timing videos”. In: *European Symposium on Research in Computer Security*. Springer. 2018, pp. 263–280 (cit. on pp. 1, 18).

[4] Davide Balzarotti, Marco Cova, and Giovanni Vigna. “Clearshot: Eavesdropping on keyboard input from video”. In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE. 2008, pp. 170–183 (cit. on pp. 1, 8).

[5] *BESTÅ Frame, black-brown, 23 5/8x15 3/4x25 1/4" - IKEA*. URL: https://www.ikea.com/us/en/p/besta-frame-black-brown-20245964/ (visited on 08/19/2020) (cit. on p. 21).

[6] G. Bradski. “The OpenCV Library”. In: *Dr. Dobb’s Journal of Software Tools* (2000) (cit. on p. 29).

[7] Matteo Cardaioli et al. “Your PIN Sounds Good! On The Feasibility of PIN Inference Through Audio Leakage”. In: *arXiv preprint arXiv:1905.08742* (2019) (cit. on pp. 1, 9).

[8] *CNN Padding | Master Data Science*. 2018. URL: http://datahacker.rs/what-is-padding-cnn/ (visited on 09/04/2020) (cit. on pp. ix, 13).

[9] Alberto Compagno et al. “Don’t Skype & Type! Acoustic Eavesdropping in Voice-Over-IP”. In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 2017, pp. 703–715 (cit. on pp. 1, 8).

[10] Mauro Conti et al. “Can’t you hear me knocking: Identification of user actions on android apps via traffic analysis”. In: *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. 2015, pp. 297–304 (cit. on p. 8).

[11] Jeffrey Donahue et al. “Long-term recurrent convolutional networks for visual recognition and description”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 2625–2634 (cit. on p. 31).

[12] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. “Learning to forget: Continual prediction with LSTM”. In: (1999) (cit. on p. 31).

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cit. on pp. ix, x, 12, 14).

[14] Paul C Kocher. “Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems”. In: *Annual International Cryptology Conference*. Springer. 1996, pp. 104–113 (cit. on p. 7).

[15] Brian Krebs. *Why I Always Tug on the ATM*. Mar. 21, 2017. URL: https://www.pinguard.com/atm-wincor-nixdorf/ (visited on 09/04/2020) (cit. on pp. ix, 3, 4).

[16] Taekyoung Kwon and Jin Hong. “Analysis and improvement of a pin-entry method resilient to shoulder-surfing and recording attacks”. In: *Ieee transactions on information forensics and security* 10.2 (2014), pp. 278–292 (cit. on p. 1).

[17] Yann LeCun, Yoshua Bengio, et al. “Convolutional networks for images, speech, and time series”. In: *The handbook of brain theory and neural networks* 3361.10 (1995), p. 1995 (cit. on p. 31).

[18] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324 (cit. on pp. x, 16).

[19] Mun-Kyu Lee. “Security notions and advanced method for human shoulder-surfing resistant PIN-entry”. In: *IEEE Transactions on Information Forensics and Security* 9.4 (2014), pp. 695–708 (cit. on p. 1).

[20] *Logitech C920 HD Pro Webcam*. URL: https://www.logitech.com/it-it/product/hd-pro-webcam-c920?crid=34 (visited on 08/19/2020) (cit. on p. 22).

[21] *Matrix tastiera impermeabile industriale Custom tastierino numerico 4 x 4 tastiere: Amazon.it: Elettronica*. URL: https://www.amazon.it/tastiera-impermeabile-industriale-tastierino-tastiere/dp/B01KHQ416K (visited on 08/19/2020) (cit. on p. 22).

[22] Keaton Mowery, Sarah Meiklejohn, and Stefan Savage. “Heat of the moment: Characterizing the efficacy of thermal camera-based attacks”. In: *Proceedings of the 5th USENIX conference on Offensive technologies*. 2011, pp. 6–6 (cit. on pp. 8, 10).

[23] Fabian Pedregosa et al. “Scikit-learn: Machine learning in Python”. In: *Journal of machine learning research* 12.Oct (2011), pp. 2825–2830 (cit. on pp. xii, 47).

[24] *Pooling — Dive into Deep Learning 0.14.3 documentation*. URL: https://d2l.ai/chapter_convolutional-neural-networks/pooling.html (visited on 09/04/2020) (cit. on pp. x, 16).

[25] Diksha Shukla and Vir V Phoha. “Stealing passwords by observing hands movement”. In: *IEEE Transactions on Information Forensics and Security* 14.12 (2019), pp. 3086–3101 (cit. on p. 1).

[26] Diksha Shukla et al. “Beware, your hands reveal your secrets!” In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 2014, pp. 904–917 (cit. on p. 8).

[27] Dawn Xiaodong Song, David A Wagner, and Xuqing Tian. "Timing analysis of keystrokes and timing attacks on ssh." In: *USENIX Security Symposium*. Vol. 2001. 2001 (cit. on p. 7).

[28] Jost Tobias Springenberg et al. "Striving for simplicity: The all convolutional net". In: *arXiv preprint arXiv:1412.6806* (2014) (cit. on p. 16).

[29] Jingchao Sun et al. "VISIBLE: Video-Assisted Keystroke Inference from Tablet Backside Motion." In: *NDSS*. 2016 (cit. on p. 8).

[30] Furkan Tari, A Ant Ozok, and Stephen H Holden. "A comparison of perceived and real shoulder-surfing risks between alphanumeric and graphical passwords". In: *Proceedings of the second symposium on Usable privacy and security*. 2006, pp. 56–66 (cit. on p. 1).

[31] Suramya Tomar. "Converting video formats with FFmpeg". In: *Linux Journal* 2006.146 (2006), p. 10 (cit. on p. 29).

[32] *Wincor-Nixdorf ATM PINGuards | PINGuards for Wincor Nixdorf ATM machines | PINGuard*. URL: https://www.pinguard.com/atm-wincor-nixdorf/ (visited on 09/04/2020) (cit. on pp. ix, 2).