MOBIUS: MOBile keystroke Inferencing Using Sensors
Keystroke Inference from Gyroscope and Accelerometer Data on Android

Daniel Johnson
University of Kansas
danielrjohnson00@ku.edu

Dawson Rooney
University of Kansas
dawsonr@ku.edu

Abstract

With an increase in sensor quality in recent years in the area of mobile devices, it begs the question, can this increase in quality be used in an adversarial fashion? Our main contribution in this paper is showing that a side channel attack of siphoning gyroscope and accelerometer data is possible and largely significant to keystroke inference. The increase in precision and accuracy of the gyroscope and accelerometer sensors allows us to gather data that is precise enough such that we can determine when a user taps their phone and where on the screen they are tapping. This data led us to creating a LSTM Recurrent Neural Network Classifier that can accurately predict characters tapped on a screen at around 70 percent accuracy. One significant limitation of our methods is that it is currently undecidable whether the user tapped a normal character, special character, or the "shift-ed" version of those. However, this still correlates to a decrease in entropy which could be very significant to key logging or password cracking.

1 Introduction

In recent years, sensors in mobile devices have drastically improved. This, for the most part, is a boon. However, this quality can be used against users for adversarial purposes. These high quality sensors can easily leak data through side channels and as the sensors become more precise, so do the the things that the adversary can infer from them. In this paper we will explore the power of the Gyroscope and Accelerometer for keystroke inference without the dangerous “HIGH_FREQUENCY_SAMPLING” permission declared for it. A completely silent sensor setup that only requires user interaction to be a foreground process, not to declare its sensors.

1.1 Reasoning

Our reasoning basically comes down to the fact that an increase in sensor quality means an increase in possible quality of side channel data leakage. With great power comes great responsibility, if phones are going to have great sensors they need to be properly protected from adversarial attacks. Our paper attempts to show that the current protections are not enough and that an adversary could obtain sensitive information off a victim phone with nothing more than accelerometer and gyroscope data within the bounds of the android permissions model.

1.2 Contributions

The main contribution of this work is the creation of a LSTM Recurrent Neural Network (RNN) model that can be used for keystroke tap inference on Android. We show that with adequate data collection and processing, our model reaches 70% accuracy at predicting what key was pressed solely based on gyroscope and accelerometer data. This model could then be used for an array of adversarial attacks to gain passwords, personal information, etc. Along with this, we publicly release the entirety of our code at this GitHub Repository: `https://github.com/DanielRJohnson/GyroLogger`.

2 Background

2.1 Sensors on Mobile Devices

Most smartphones in production today hold a wide array of on-board physical sensors. Some of these include sensors that measure temperature, gravity, light, air pressure, humidity, linear acceleration, and rotation. These are used for everything from rotating orientation of the screen to automatically shutting off the device at exceedingly hot temperatures. With the increase of sensor accuracy, some of these sensors have been shown to provoke security vulnerabilities.

We will specifically focus on linear acceleration and rotation sensors. The gyroscope and accelerometer sensors provide three-dimensional readings at very high frequencies. Gyroscope readings represent roll (x), pitch (y), and Azimuth (z) to describe the current rotation of the device. Accelerometer readings represent three-dimensional vectors based on the linear acceleration of the device at a given time.

2.2 Android Permissions Model

The android permission's model in android 12 (API level 31) or higher places a limit on the refresh rate of specific sensors. The accelerometer and gyroscope are both affected by this and are therefore limited to a 200 Hz refresh rate before needing to be declared with the dangerous 'HIGH_SAMPLING_RATE_SENSORS' permission. Our data collection model collects at a rate of 100 Hz, so all that we are presenting is currently possible under the latest android API without declaring a dangerous permission.

However, the latest versions of android no longer allow background applications to act as listening points for sensors. So, an adversary would need to declare the application that collects the data as a foreground process visible to the user. This would require some generic social engineering from the adversary and is not outside of the realm of possibilities.

2.3 LSTM Recurrent Neural Networks

LSTM Recurrent Neural Networks (or LSTMs) are a type of neural network described as early as 1995 that are fundamentally designed to operate on constantly flowing sequence data. To accomplish this, the model must take into account more than the current input of the network. LSTMs operate on a combination of the current input vector, the output of the model for the previous pass, and some cell function meant to store information over every pass [4]. The cell function is defined through the use of a forget gate, input date, and output gate. These are defined as follows.

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \tag{1}$$

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \tag{2}$$

$$o_t = \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \tag{3}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot o_t \tag{4}$$

After the input, previous output, and the cell function are known, the model makes the prediction as such.

$$h_t = o_t \odot \sigma_h(c_t) \tag{5}$$

With these combinations of input, short term memory, and long term memory, LSTMs are able to do better sequence classification than any other recurrent models known to date.

3 Attack Model

Our attack model simply assumes that the adversary somehow gets an application onto the victim's phone that can export gyroscope and accelerometer data over the internet. This would require the user to accept the dangerous permission of internet access, and possibly require the adversary to do some generic social engineering to get the user to believe the app should be allowed as a foreground process. We believe that the average user may not fully understand that giving an application the foreground also allows it to continuously read data. We believe this attack model to be plausible, and that the entirety of this attack could be hidden behind a game, or simply a data logging app that shows the user what they're gyroscope/accelerometer has been up to. Our attack model also assumes the user types with the phone sitting on a sturdy flat surface. In reality this is the least plausible part of our attack model and we are currently examining it to see if we can manage without this guarantee.

4 Methodology & implementation

An overview of our implementation process can be found in figure 1. We split the process into modular pieces that can be ran in sequence to collect data, process data, and train models.

4.1 Data Collection

For our data collection device, we used a single Google Pixel 3 phone equipped with an app called sensor stream [1] as seen in figure 2. Sensor stream is a free to use data collection app for android that allowed us to set collection intervals for the gyroscope and accelerometer sensors and export the data easily to a computer to do analysis on.

The application requires the user to manually turn the sensors on and off when collecting data, which could result in a significant imbalance in the raw data files between the different sensors spending on their refresh rates. We modified the open source data sensor collection point on PC so that it only collected data if both the accelerometer and the gyroscope were active. This way we ensured that the data from both sensors easily matched up with each other.

The app and data center combination used an asynchronous function to collect the outputs from the sensor. Based on some careful examination, the app must create some sort of a queue of data reads for each sensor. It exports the data by reading from the queue for one sensor for a short period of time, and then switching to the other sensor and reading from its queue for a short period of time.

For our data collection process, we used a 100 Hz sample rate which is well within the 200 Hz High Sampling bottleneck imposed by the android permissions model. We used the vertical keyboard on the Google Pixel 3, and had the phone placed on a flat surface for the entirety of our testing. We gathered data initially by tapping words and phrases into the payload textbox for sensor stream, but that proved inefficient for getting a semi-equal distribution of taps on every key. So we switched to just tapping each key individually until they were all relatively similar in the number of taps each received.

We also attempted to grab the full area of each key by tapping not just the center, but around the edges to the point of overlap between the keyboard letters. We also collected data with a variety of different tap pressures, from light taps to forceful, however those are not categorized in any of our data models, it was more just to get a better overall model.

4.2 Data Processing

Having high quality data is key to effectively training our model. Once we have our raw sensor and payload data, we need to convert it into a more legible format for our analysis and to be easy for our model to understand.

To do this, we loop through every CSV file representing a recording session to find every time that the character payload changes. Once we have all of the payload changes, we store that corresponding sensor reading along with the ten readings before it and the ten readings after it. This series of sensor readings becomes our training and testing examples for our model, and the output would be the character of the linked payload change.

Once we have this baseline training data, it is important to transform it to get better results out of our model. We chose to use Z-score normalization for each feature. Z-score normalization is defined as follows.

$$x' = \frac{x - \hat{x}}{\sigma} \tag{6}$$

This gives our data a zero mean and a unit variance, which makes it much easier for our model to come to a meaningful hypothesis. Figures 3 and 4 show descriptive statistics of our features before normalization and figures 5 and 6 show descriptive statistics of our features after normalization.

Next, we use SMOTE oversampling [2] to alleviate any class imbalance problems. In our data collection, some characters were more common than others, so after ensuring that we have enough data from characters to be representative we use SMOTE oversampling to make every class exactly equal. SMOTE works by finding examples that are close to each other in feature space using the K-Nearest Neighbors algorithm, drawing a line in-between said examples, and making a new synthetic example at some point along the drawn line. SMOTE has been one of the most successful and reliable methods of oversampling in machine learning to date. This means that our model will not have any biases purely based on the amount of representation in training.

After capturing payload changes and their surrounding readings, scaling the input using Z-score normalization, and using SMOTE for appropriate oversampling,

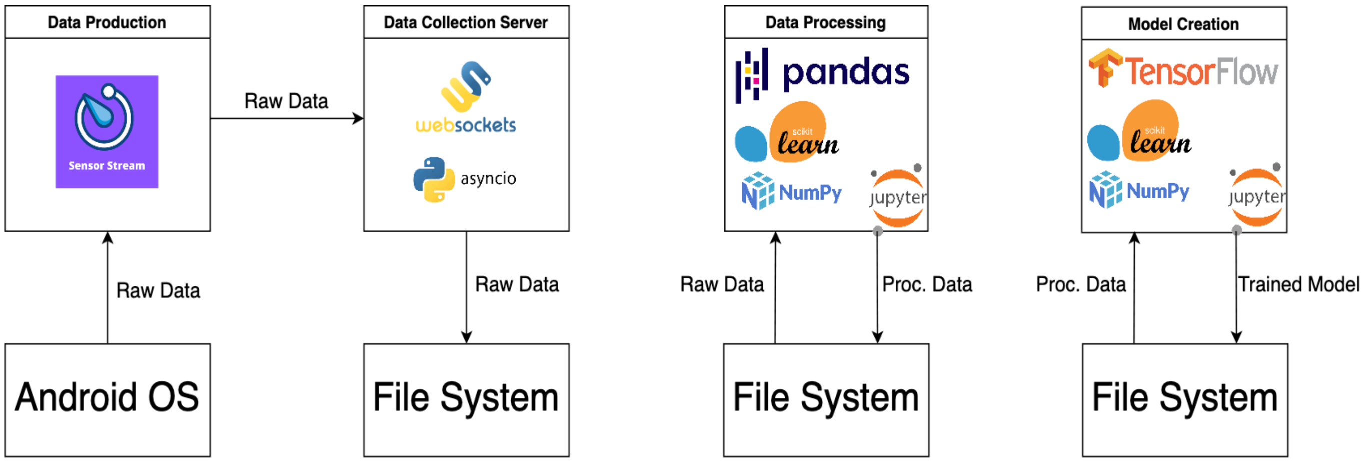

Figure 1: System Diagram

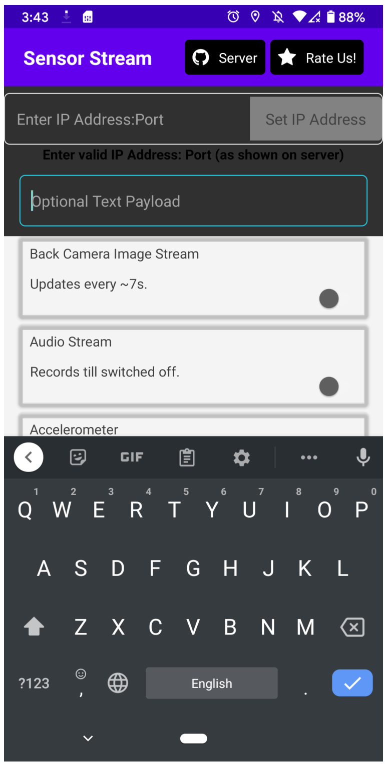

Figure 2: View of Data Collection Process

	acc_x	acc_y	acc_z
count	72870.000	72870.000	72870.000
mean	0.061	-0.148	9.822
std	0.378	0.174	0.131
min	-3.452	-2.639	4.411
25%	-0.036	-0.193	9.796
50%	0.072	-0.138	9.821
75%	0.150	-0.109	9.847
max	5.020	2.910	17.643

Figure 3: Statistics of accelerometer data before normalization

	gyro_x	gyro_y	gyro_z
count	7.287e+04	7.287e+04	7.287e+04
mean	-1.955e-04	1.127e-04	-1.138e-04
std	2.003e-03	2.240e-03	1.733e-03
min	-6.658e-03	-4.927e-03	-2.796e-03
25%	-1.198e-03	-1.332e-03	-1.465e-03
75%	1.065e-03	1.864e-03	6.658e-04
max	3.728e-03	7.590e-03	5.593e-03

Figure 4: Statistics of gyroscope data before normalization

	acc_x	acc_y	acc_z
count	7.287e+04	7.287e+04	7.287e+04
mean	-1.872e-17	4.493e-16	2.171e-15
std	1.000e+00	1.000e+00	1.000e+00
min	-2.272e+01	-1.634e+01	-2.991e+01
25%	-4.489e-01	-3.553e-01	-3.773e-01
50%	3.857e-02	8.178e-02	-3.416e-03
75%	3.985e-01	4.083e-01	3.749e-01
max	2.837e+01	2.217e+01	4.934e+01

Figure 5: Statistics of accelerometer data after normalization

	gyro_x	gyro_y	gyro_z
count	7.287e+04	7.287e+04	7.287e+04
mean	-5.492e-16	-9.610e-16	6.241e-17
std	1.000e+00	1.000e+00	1.000e+00
min	-3.226e+00	-2.250e+00	-1.548e+00
25%	-5.009e-01	-6.447e-01	-7.796e-01
75%	6.292e-01	7.814e-01	4.500e-01
max	1.959e+00	3.338e+00	3.293e+00

Figure 6: Statistics of gyroscope data after normalization

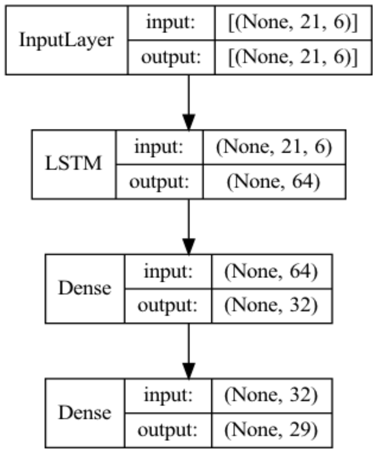

Figure 7: Visualization of LSTM Model

we believe that we have then created a representative dataset fit for analysis and model understanding.

4.3 Network Architecture

For our character prediction model's architecture, we decided to choose a simple, small LSTM (Long Short Term Memory) architecture. We chose this recurrent architecture to better predict time series data of multiple sensor readings, this way we try to capture the whole press effectively. A relatively small model was chosen to show that accurate predictions are possible without exorbitant complexity.

Figure 7 shows our model with each layer's type and its shape of input and output. The (21,6) input shape resembles the three dimensions of the gyroscope and accelerometer over ten sensor readings before and after the character payload change. After passing through an LSTM layer and a fully-connected layer, the feed gets to the output layer. The (29) output shape resembles the 29 non-special characters that we tracked. After a softmax activation, the character prediction is made from the probability distribution of the output.

With this capable and lightweight model, we were able to come to a strong model hypothesis while minimizing training time and compute power.

4.4 Training Process

To train our model, we used K-Fold cross validation with a K of five. Since we are using a standard Tensorflow model, we are able to use the built-in fit method accelerated by our GPU. Each fold is trained for 20 epochs with a batch size of 64. Once each fold has trained a model, we save the model with the highest testing accuracy.

4.5 Evaluation & Results

Each of the five folds that we ran had a training and testing accuracy of between 60 and 70 percent. The highest of which had a 69% testing accuracy and a 70% accuracy on our entire data set. Figure 8 shows the confusion matrix of this best model over our dataset. A confusion matrix shows what the model predicted and what the example was for each example in our data. From this, we can see that this model has good success at generalizing to recognize all symbols, but still has some problems that could be addressed in the future.

The prediction accuracies of each character are shown in figure 9. From this, we can see that there are some characters being predicted with very high accuracies and some characters being predicted with very low accuracies. This means that further improving our model will lead to better results because this variance likely points to a sub-optimal minimum being found by the neural network training. If each accuracy was very similar around 70%, that would point to us hitting a theoretical maximum of how accurate we could ever get. While some characters are not being predicted well, this still leads to much hope on how well our methods could work in the future.

5 Related Work

5.1 Tap Inference from Physical Sensors

Previous attempts at something like this were done in 2012 on much older versions of android with much lower quality sensors. The papers that explored this area did area classifications on taps [7] in attempts to find regions of the screen that the victim/user was tapping in.

Another paper attempted to classify key inputs [6] like our paper using only accelerometer data, and the horizontal keyboard, which does not make for a very realistic model in the modern age of smart devices. The results from this paper showed that, while some information could be gathered from the accelerometer, it was only enough to determine key taps at around 18% accuracy even on the horizontal keyboard. Our model is able to determine key taps at around 70% on the vertical keyboard, which is a significant improvement

5.2 Tap Inference from Audio

There have been previous attempts of keystroke inference based on the sound of a tap. Specifically, Kim et. al. [5] reached reasonable success of 75% accuracy on QWERTY characters in an ideal situation. The main disadvantage of this approach is that different types of noise can change performance drastically. With a robust enough model, combining this method with physical sensors could be useful work to be done.

5.3 Tap Inference from Video

There have been previous attempts of keystroke inference based on streaming video from the camera of a phone. Chen et. al. [3] explored using human eye gaze tracking for keystroke inference. They have good success at cracking a user's pin and picking out select words. The main disadvantage of this approach is that it assumes you are able to see a user's eye very well to get great results. This method also relies on the user actually looking at the keys that they press, but not every user looks at the keys that they are typing. Along with this, the camera is a dangerous permission in Android, and users should be very skeptical to give it up.

6 Future Work

6.1 Deeper Training of Models

We argued that for the sake of time and resources we would use a small model. One avenue of future work that we plan to explore is creating much more capable models by adding layers and increasing sizes. This would take more compute time and power, but it would allow us to model much more complicated relationships and get better results.

6.2 GyroLogger & GyroCrAccer

Our goals for future work are to extend our current model to work in tandem with another model which checks over the same set of data but while examining the special characters keyboard. This would allow us to create a inferencing tool that could look at character

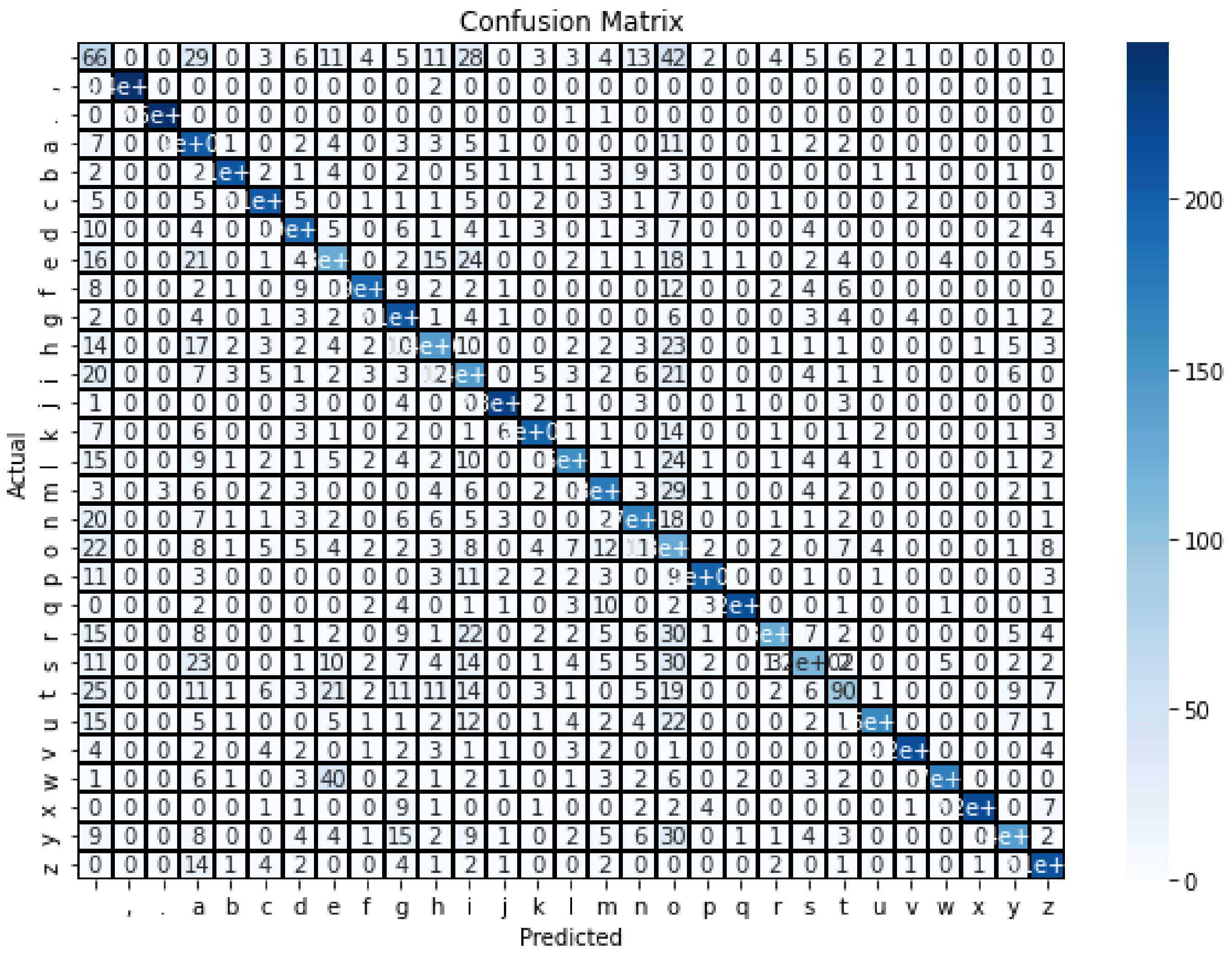

Figure 8: Confusion Matrix of Best Model

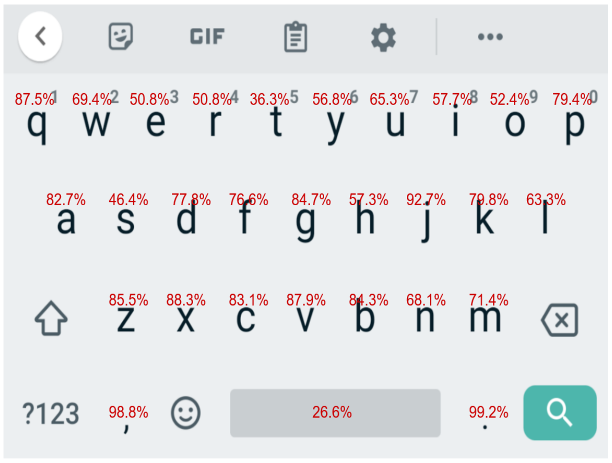

Figure 9: Per character prediction accuracies

frequency data and attempt to determine if the current input is text, or if it might be a special character in a password or other sensitive information. This would be the culmination of our research because the tool would both be able to assist in password cracking, as well as log possible sensitive data such as text messages, web searches, maps addresses, and any other sensitive information that uses the keyboard on a mobile device.

Another goal for our future work, and possibly even a more important goal than the combination model, is shoring up the main weakness of our attack model. We hope to generalize the model to handheld orientations, and hopefully any orientation in the future, and we believe this to be possible.

7 Conclusion

This work performs analysis on side channel data leakage through the gyroscope and accelerometer of a Google Pixel 3 phone. The data is used in the creation of a ML model that attempts to infer keyboard letter taps based on the changes of the aforementioned sensors. This work is a proof of concept that these sensors can be used in an adversarial fashion and that they can be used to garner sensitive information in the long term. With around 70% average accuracy to infer any individual keyboard input, a piece of software in this same vein could significantly lower the entropy on password cracking, or gain significant information towards inferring text messages, emails or other private information. Lastly, the sensors were used within the bounds of non-dangerous permissions and thus did not need user interaction to become active, which is a point of concern moving forward in android permissions and phone hardware.

References

[1] Sensor stream. https://play.google.com/store/apps/details?id=com.sensorsensei&hl=en_US&gl=US. Accessed: 9/2021.

[2] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority oversampling technique. In *Journal Of Artificial Intelligence Research*, volume 16, pages 321–357, 2002.

[3] Yimin Chen, Tao Li, Rui Zhang, Yanchao Zhang, and Terri Hedgpeth. Eyetell: Video-assisted touchscreen keystroke inference from eye movements. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 144–160, 2018.

[4] Klaus Greff, Rupesh K. Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2222–2232, 2017.

[5] Hyosu Kim, Byunggill Joe, and Yunxin Liu. Tapsnoop: Leveraging tap sounds to infer tapstrokes on touchscreen devices. *IEEE Access*, 8:14737–14748, 2020.

[6] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. Accessory: Password inference using accelerometers on smartphones. In *HotMobile '12: Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, pages 1–6, 2012.

[7] Zhi Xu, Kun Bai, and Sencun Zhu. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *WISEC '12: Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 113–124, 2012.