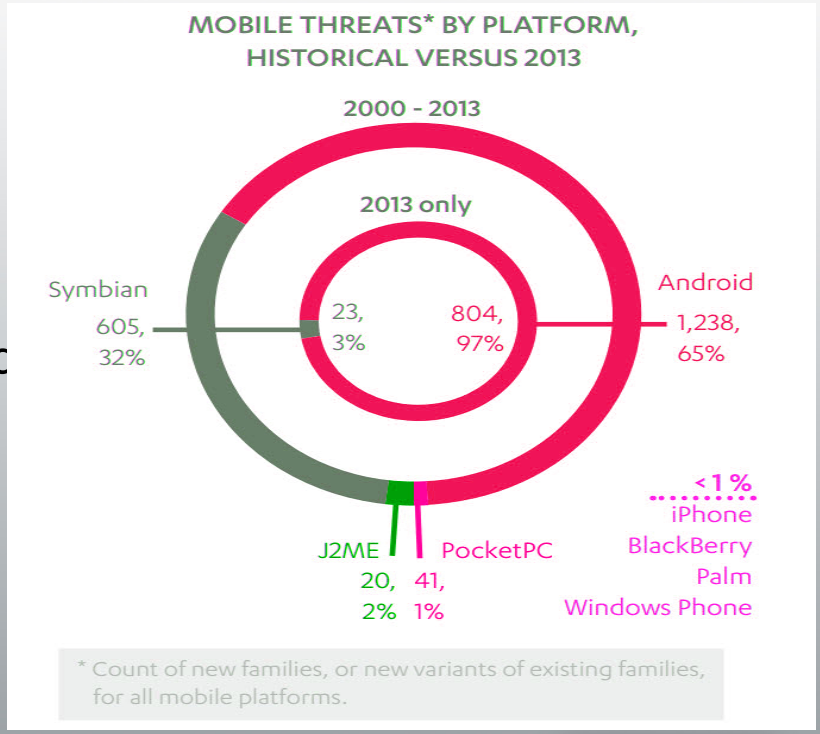
Examining Features For Android Malware Detection

Matthew Leeds, Miclain Keffeler, Travis Atkison

THE UNIVERSITY OF
ALABAMA®

Why is Android Security Important?

- Mobile devices are increasingly becoming people's primary means of computing.
- Compromised mobile devices can lead to identity theft, robbery, etc.
- Far more malware plagues Android than any other platform. [1]
- Android is the most popular mobile devic OS globally. [1]
- Android poses different challenges for malware protection than desktops.



What Does Android Malware Look Like?

- Often it's installed from outside the Play Store (either a 3rd party store such as F-Droid or directly from an apk file).
- Most commonly a legitimate application has been modified to have a malicious component.
- The application may even download the malicious component at runtime to avoid detection at install time.
- One common action malware takes is to send texts to paid numbers to steal money.

What Do Existing Anti-Malware Strategies Look Like?

- Preventive measures are the most effective (use the manufacturer OS, keep it updated, don't allow installation from unknown sources, etc.).
- Reactive measures can help as well (AV apps exist but only the OS can scan the contents of apps).
- Many anti-malware apps are proprietary, so it's unclear how they work.

Previous Academic Research

- Dimjasevic et al. use “maline” to orchestrate running applications in virtual devices, sending random events to them, and recording the system calls they make. [1]
- Li et al. used an SVM approach by looking at dangerous permissions that are likely used by Android malware. [2]
- A neuro-fuzzy based clustering method is presented by Altaher et al. [3]
- Mobile botnet families are clustered by Aresu et al. by analyzing the generated HTTP traffic. With the algorithm they used, a small number of signatures can be extracted from the clusters, allowing it to achieve a good tradeoff between the detection rate and the false positive rate. [4]
- Bengio et al. found that gradient descent may be inadequate to train for tasks involving long-term dependencies, such as consistently dangerous permissions. [5]
- Alam et al. were able to use a random forest of decision trees in detecting malware in Android devices. [6]
- We used a single layer neural network in detecting malicious android applications. [7]

Extending Our Research

- The work presented here is an extension of previous efforts in [7].
- Feature selection was based on permissions and system calls.
  - Features used in a ML Model
  - Specifically, it was a single layer neural network using a Gradient Descent Optimizer and softmax regression, implemented in TensorFlow.
- This method produced classification accuracy results between 80-85% using permissions, and a considerably lower accuracy using system calls.

Previous Classification Accuracy vs False +- Rates

- Using Permissions Data
  - False Positive ≅ Negative with enough training
- 20% Reduction in Accuracy with System Call Data

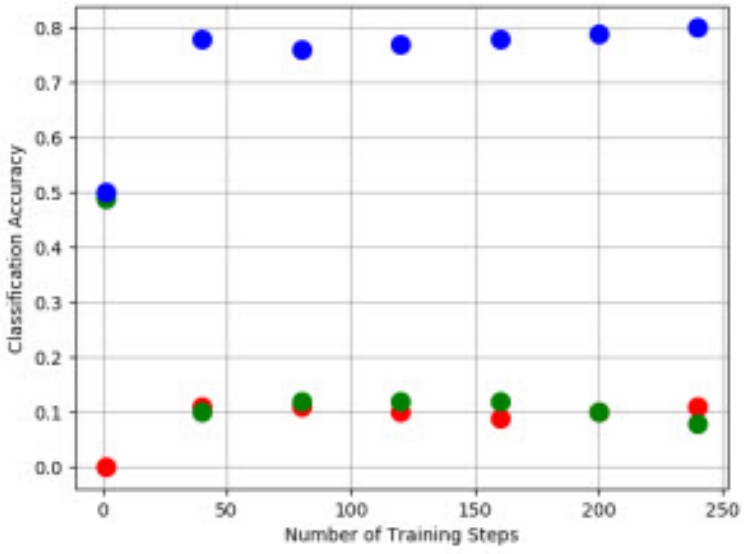


Fig. 1

NUMBER OF TRAINING STEPS VS CLASSIFICATION ACCURACY AND FALSE NEGATIVE/POSITIVE RATES.

= Classification Accuracy

=False Positive Rate

=False Negative Rates

Examining Malicious Feature Weights

- Previously, examination of the model itself was unavailable
  - READ_PHONE_STATE is second most used in malicious apps
    - gives applications the ability to access information such as the phone number of the device and current cellular network information
- Malicous:Benign Ratio was much higher

Table 1

PERMISSIONS INDICATIVE OF MALICIOUSNESS

Rank	Permission	Weight
1	READ_PHONE_STATE	1.179
2	WRITE_APN_SETTINGS	0.938
3	INSTALL_PACKAGES	0.748
4	READ_SMS	0.611
5	GET_TASKS	0.579
6	RECEIVE_BOOT_COMPLETED	0.558
7	INTERNET	0.522
8	WRITE_SMS	0.497
9	MOUNT_UNMOUNT_FILESYSTEMS	0.429
10	BIND_ACCESSIBILITY_SERVICE	0.417

https://developer.android.com/reference/android/Manifest.permission.html

Examining Benign Feature Weights

- USE_CREDENTIALS allows authentication with Google accounts.
  - Requires user interaction
- Limited applications for malware in Bluetooth

Table 2

PERMISSIONS INDICATIVE OF BENIGNITY

Rank	Permission	Weight
1	WAKE_LOCK	0.869
2	USE_CREDENTIALS	0.662
3	BLUETOOTH_ADMIN	0.571
4	ACCESS_NETWORK_STATE	0.558
5	BLUETOOTH	0.538
6	CALL_PHONE	0.479
7	PACKAGE_USAGE_STATS	0.470
8	DOWNLOAD_WITHOUT_NOTIFICATION	0.421
9	INTERACT_ACROSS_USERS_FULL	0.347
10	WRITE_MEDIA_STORAGE	0.320

https://developer.android.com/reference/android/Manifest.permission.html

Malicious System Call Feature Weights

- fchmodat, found in row 9, is most logical
  - apps may want to set the executable bit on a script they're injecting
  - make a file writable that isn't by default.

Table 3

SYSTEM CALLS INDICATIVE OF MALICIOUSNESS

Rank	System Call	Weight
1	mkdirat	0.322
2	nanosleep	0.272
3	getdents64	0.149
4	fstatat64	0.138
5	clock_gettime	0.137
6	exit	0.130
7	pread64	0.124
8	getppid	0.112
9	fchmodat	0.108
10	futex	0.104

http://man7.org/linux/man-pages/man2/fchmodat.2.html

Benign System Call Feature Weights

- Readlinkat only allows the app to read the contents of a symbolic link
  - It is unclear how, if at all, this system call could be utilized in a malicious manner

Table 4

SYSTEM CALLS INDICATIVE OF BENIGNITY

Rank	System Call	Weight
1	readlinkat	0.621
2	renameat	0.333
3	faccessat	0.277
4	pwrite64	0.247
5	fsync	0.214
6	sendmsg	0.196
7	getsockname	0.161
8	gettimeofday	0.135
9	epoll_create1	0.130
10	pipe2	0.129

http://man7.org/linux/man-pages/man2/readlinkat.2.html

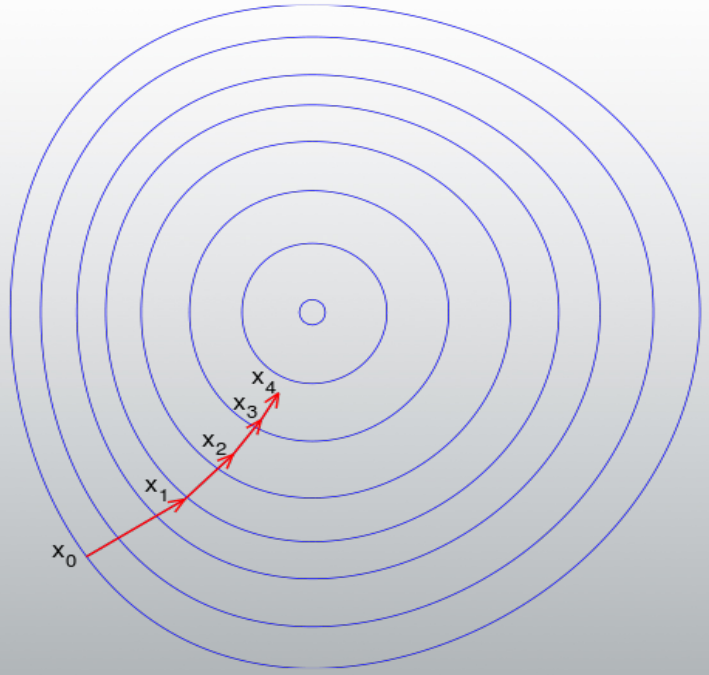
Our Methodology

1. Gather malware samples using the andrototal.org API
2. Classify each as benign or malicious, again using andrototal.org
3. Static analysis: Parse the *AndroidManifest.xml* in each to determine requested permissions.
4. Using TensorFlow, train separate neural networks on the various features (supervised learning).
5. Have each model classify a different subset of samples, and compare its predictions to the "truth" to determine accuracy.

Our Machine Learning Model

- We used the Gradient Descent Optimizer provided by TensorFlow with a learning rate of 0.01.
- The model seeks to minimize cross-entropy, which is essentially the difference between the correct and prediction vectors.
- For example, the prediction vector [0.8,0.2] would mean the model assigns an 80% chance to the app being malicious.

Gradient Descent Visualization

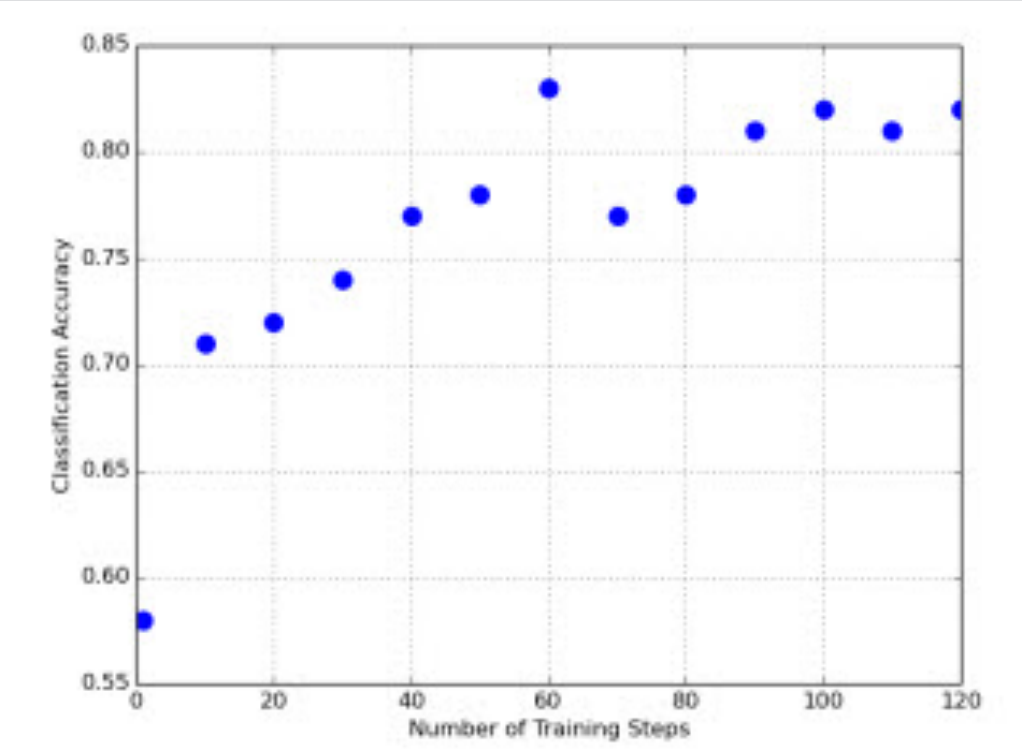


Features Explored

- **Java API Call Data**
- Decaying Learning Rate
- Hyperparameter Optimization

Java API Call Data

- API calls give more specific information about what actions the apps are trying to perform
- Expanded into 2000 App dataset
  - 82% Classification Accuracy with larger dataset (5-7% increase)



Classes Indicative of Maliciousness

- Improved model is very notable
  - Java.crypto.SecretKeyFactory was previously 5th most indicative in 200 app dataset
    - Now 66th in 2000 app dataset

Table 5

CLASSES INDICATIVE OF MALICIOUSNESS

Rank	Class	Weight
1	java.net.SocketException	0.309
2	java.lang.StringBuffer	0.303
3	java.lang.Character	0.294
4	javax.crypto.Cipher	0.282
5	java.io.ByteArrayInputStream	0.193
6	java.lang.UnsatisfiedLinkError	0.165
7	java.net.Proxy	0.161
8	java.util.Hashtable	0.161
9	java.util.zip.InflaterInputStream	0.157
10	java.util.GregorianCalendar	0.153

Classes Indicative of Benignity

- data structure classes are prevalent
  - abstract list,currency
  - ia reflection of the more careful coding practices used for benign applications.

Table 6

CLASSES INDICATIVE OF BENIGNITY

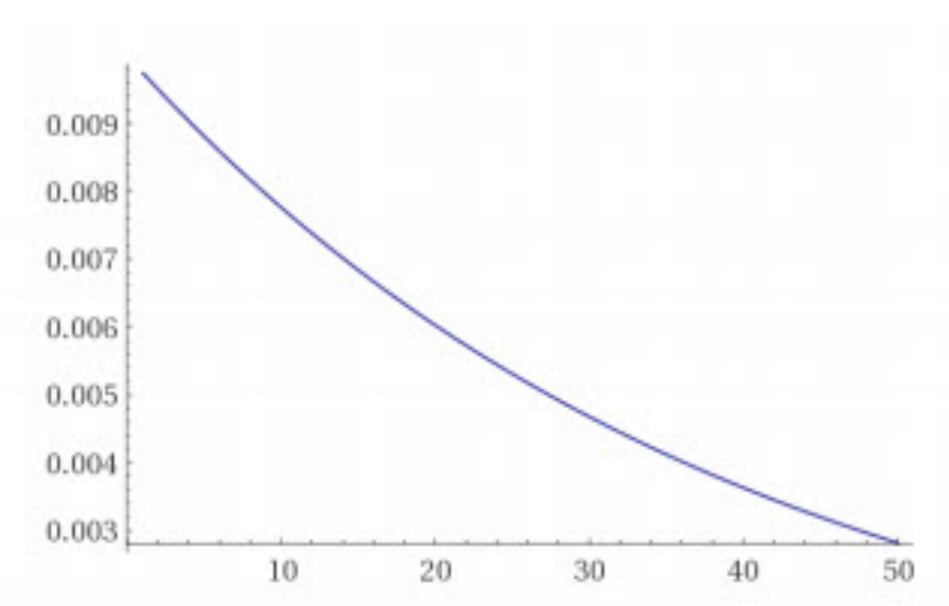
Rank	Class	Weight
1	java.util.AbstractList	0.266
2	java.util.concurrent.atomic.AtomicLong	0.229
3	java.lang.reflect.Constructor	0.218
4	java.security.Signature	0.209
5	java.io.FilterOutputStream	0.199
6	java.util.Currency	0.194
7	java.security.SecureRandom	0.188
8	java.lang.Throwable	0.177
9	java.nio.charset.Charset	0.174
10	java.lang.Runtime	0.173

Features Explored

- Java API Call Data
- **Decaying Learning Rate**
- Hyperparameter Optimization

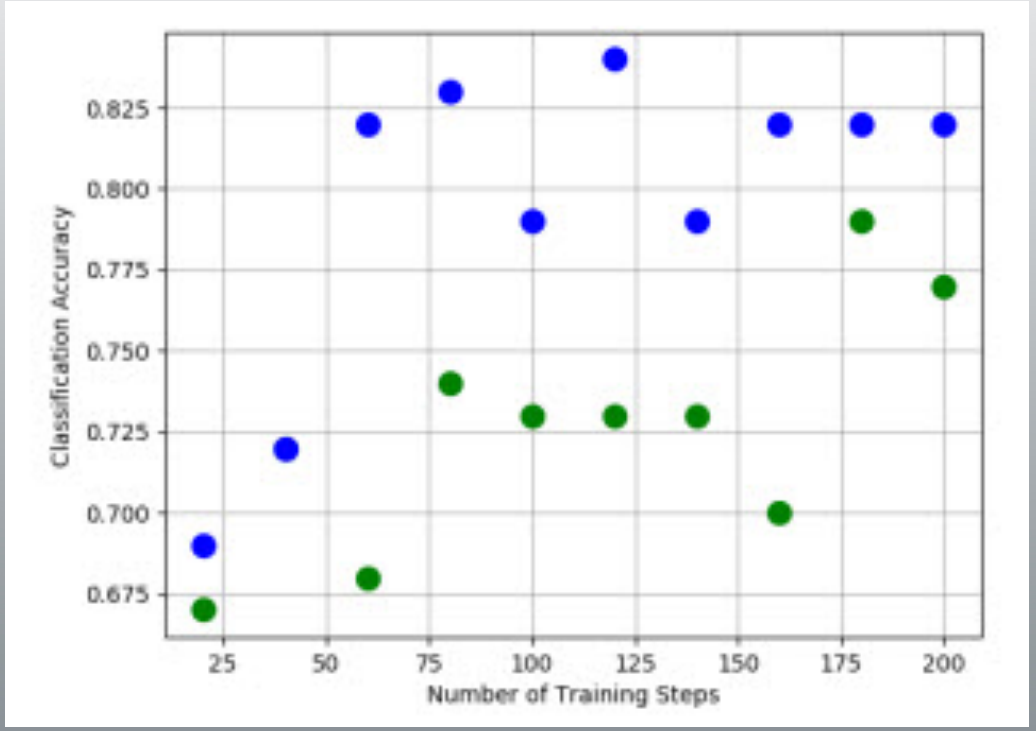
Learning Rate Decay

- Constant learning rate in previous work
  - Exponentially decaying rate reduces overfitting
    - $y = (0.01)(0.975)^x$



Learning Rate Decay Results

- API Calls as a feature
  - 2000 apps, learning rate = .1 (decayed starts at .1)

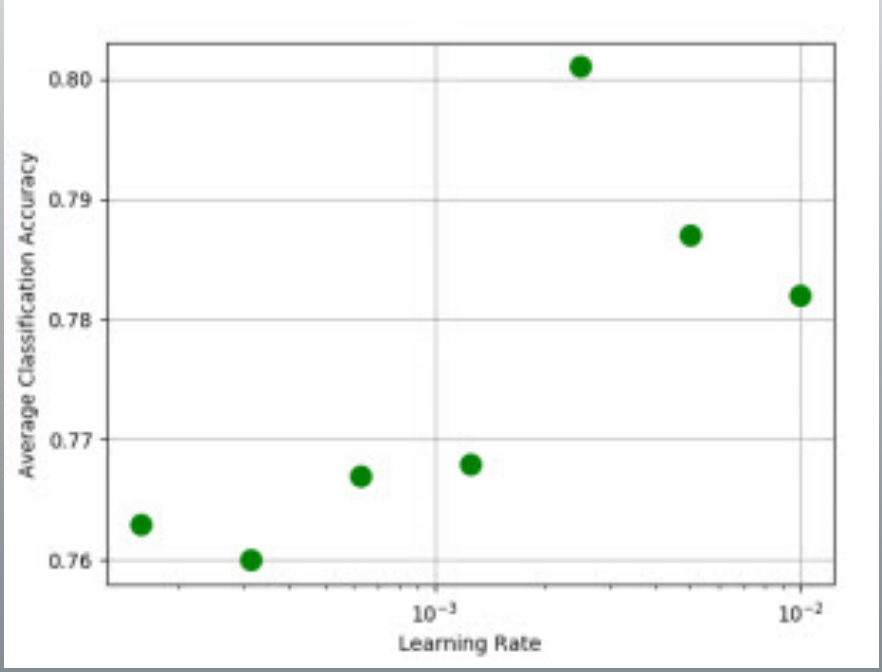


Features Explored

- Java API Call Data
- Decaying Learning Rate
- **Hyperparameter Optimization**

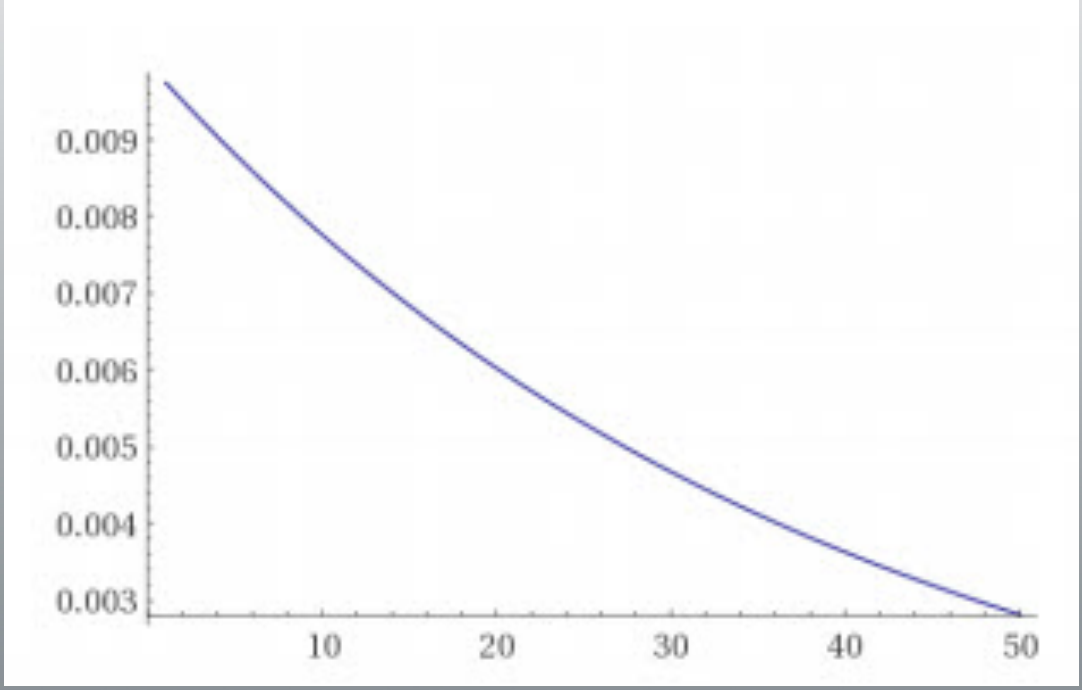
Hyperparameter Optimization

- The higher-level properties of the model that are chosen by the user
  - Learning rate, chunk size, and training steps
  - .0025=Optimal learning rate



Making Experimentation Easier

- The ability to swap features and models around easily is imperative for future testing
  - Open Source: https://github.com/mwleeds/android-malware-analysis



Future Work

- Comparison of ML Models
  - SVM's, Decision Trees, etc.
- More Features through Static and Dynamic Analysis
- Using Fuzzyhashes to compare applications
- Conformal Predictions
  - Classify apps as benign or malicious with a specific degree of confidence

This Presentation has been made available online:

Thank You

Questions?

Works Cited

- 1. G. Kelly. (2014) Report: 97% of mobile malware is on android. this is the easy way you stay safe. [Online]. Available: http://www.forbes.com/sites/gordonkelly/2014/03/24/report-97-of-mobile-malware-is-on-android-this-is-the-easy-way-you-stay-safe/#4ddd7dbe7d53
- 2. J. Callaham. (2015) Google says there are now 1.4 billion active android devices worldwide. [Online]. Available: http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide
- 3. J. Sahs and L. Khan, "A machine learning approach to android malware detection," in Intelligence and Security Informatics Conference (EISIC), 2012 European. IEEE, 2012, pp. 141–147
- 4. W. Li, J. Ge, and G. Dai, "Detecting malware for android platform: An svm-based approach," in Cyber Security and Cloud Computing (CSCloud), 2015 IEEE 2nd International Conference on. IEEE, 2015, pp. 464–469.
- 5. A. Altaher and O. BaRukab, "Android malware classification based on anfis with fuzzy c-means clustering using significant application permissions."
- 6. A. Altaher, "Classification of android malware applications using feature selection and classification algorithms," VAWKUM Transactions on Computer Sciences, vol. 10, no. 1, pp. 1–5, 2016.
- 7. M. S. Alam and S. T. Vuong, "Random forest classification for detecting android malware," in Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing. IEEE, 2013, pp. 663–669.
- 8. "Gradient Descent" https://en.wikipedia.org/wiki/Gradient_descent#/media/File:Gradient_descent.svg
- 9. "MNIST For ML Beginners" https://www.tensorflow.org/versions/r0.12/tutorials/mnist/beginners/index.html