Hybrid Analysis of Android Apps for Security Vetting using Deep Learning

Dewan Chaulagain, Prabesh Poudel, Prabesh Pathak, Sankardas Roy
Department of Computer Science
Bowling Green State University
Bowling Green, Ohio, USA
dewanc, pprabes, ppathak, sanroy@bgsu.edu

Doina Caragea
Department of Computer Science
Kansas State University
Manhattan,
Kansas, USA
dcaragea@ksu.edu

Guojun Liu, Xinming Ou
Computer Science and Engineering
University of South Florida
Tampa, Florida, USA
guojunl@mail.usf.edu,
xou@usf.edu

***Abstract*—The phenomenal growth in use of android devices in the recent years has also been accompanied by the rise of android malware. This reality warrants development of tools and techniques to analyze android apps in large scale for security vetting. Most of the state-of-the-art vetting tools are either based on static analysis or on dynamic analysis. Static analysis has limited success if the malware app utilizes sophisticated evading tricks. Dynamic analysis on the other hand may not find all the code execution paths, which let some malware apps remain undetected. Moreover, the existing static and dynamic analysis vetting techniques require extensive human interaction. To address the above issues, we design a *deep learning* based hybrid analysis technique, which combines the complementary strengths of each analysis paradigm to attain better accuracy. Moreover, automated feature engineering capability of the deep learning framework addresses the human interaction issue. In particular, using lightweight static and dynamic analysis procedure, we obtain multiple artifacts, and with these artifacts we train the deep learner to create independent models, and then combine them to build a hybrid classifier to obtain the final vetting decision (malicious apps vs. benign apps). The experiments show that our best deep learning model with hybrid analysis achieves an area under the precision-recall curve (AUC) of 0.9998. In this paper, we also present a comparative study of performance measures of the various variants of the deep learning framework. Additional experiments indicate that our vetting system is fairly robust against imbalanced data and is scalable.**

***Index Terms*—Android App, Security Vetting, Deep Learning, LSTM, Classifier, Static Analysis, Dynamic Analysis**

I. Introduction

Android ecosystem has been growing at a tremendous rate, which currently occupies about 74% of worldwide mobile OS market share [1]. This growth in use of android devices and android apps not only attracted regular users but also lured attackers to develop malicious apps. Discovery of malware apps being present in Google Play and other app stores make news headlines regularly. The growth in android malware seen over the years warrants the need of a vetting system that can classify apps as malicious or benign. Such a vetting system can be deployed in an app store, or can be used as a stand-alone system on a smart phone.

Static analysis aims to capture the behavior of the android app under scrutiny without executing the app. State-of-the-art static analysis tools like Flowdroid [2], Amandroid [3], *etc*. search for a (control/data flow) pattern (representing information leakage, *etc*.) in the app code. A static analysis tool typically builds Control Flow Graph (CFG), Data Flow Graph (DFG), and more, which could be computationally intensive tasks [4]. Furthermore, static analysis typically suffers from high rate of false alarms, and it has limited success if the app utilizes sophisticated evading techniques such as code obfuscation and dynamic loading. On the other hand, in dynamic analysis, an emulator is used to execute the android app and record the runtime behaviors, which are used to detect possible malciousness. Dynamic analysis overcomes many of the problems associated with static analysis, but it may fail to explore all the code execution paths, which leads to missed alarms. Moreover, a malicious app may recognize that it is being executed in an emulator (*i.e.*, not on a real phone) and could hide the malicious part of the app [4], and thus may escape vetting.

Hybrid analysis (which involves both static and dynamic analysis) is more effective as it can leverage the complementary strengths of both type of analyses. For instance, a malicious app which turns off its malicious feature might evade dynamic analysis but static analysis can still extract the static features (*e.g.*, API-calls as seen in the app code) of the app and detect its maliciousness. On the other hand, apps with obfuscated code or dynamic code loading behavior (which might evade static analysis) could be tracked by dynamic analysis. However, the capability of hybrid analysis remains limited if its component static analysis or dynamic analysis has following limitations: (a) it could be computationally intensive, and (b) it relies on human expertise to design the control/data flow-based signature [3] of a malware app.

To address the above limitations, the research community [5] [6] has recently shown immense interest in leveraging machine learning to design a vetting system. The main idea is to gather static and dynamic features of an app to feed to a machine learning (ML) unit for the app classification. Unfortunately, as the feature set which is used by the ML unit (for app classification) are manually engineered, the reliance of the vetting system on human expertise remains. Furthermore, with rapidly changing android ecosystem, it becomes virtually impossible for a human specialist to timely obtain new ma-

licious signatures/patterns. Literature reflects that the vetting tools that rely on traditional machine learning implementation lose their efficacy over time [7]. To keep up with the rapidly changing android ecosystem, we need a system that can learn features by itself instead of depending on a human specialist.

Recent advancement in deep learning (DL) offers automated *feature engineering* [8], which shows promise of identifying the features from the raw dataset without applying the domain knowledge. Furthermore, unlike ML, deep learning models' performance does not plateau with growth in data size [8] [9]. With increased computational power and algorithmic improvements, use of deep learning might lead to multiple benefits such as higher classification accuracy (precision/recall), coping up with regular changes in android ecosystem, and more [10]. Furthermore, traditional ML models are not best suited for learning from sequencial artifacts (*e.g.*, a time series like the executed app code sequence). In contrast, DL technology Recurrent Neural Networks (RNN) can accept sequential artifacts, and attempts to maintain (*i.e.*, remember) states for an arbitrarily long context window.

However, learning long-term dependencies in a sequence (*e.g.*, time series) is not possible with traditional RNN that uses gradient descent based learning [11] [12]. The problems of failing to track long-term dependencies and vanishing/exploding gradient might be resolved (to some extent) by using Long Short Term Memory (LSTM) [12]. LSTM [13] is a RNN architecture that comprises of multiple recurrently connected memory blocks. In the current work, we use the LSTM technology and its variants (*e.g.*, Bi-directional LSTM and Attention-based Bi-directional LSTM) to design the classifier.

In particular, we design a new *hybrid analysis* based security vetting system for android apps, which leverages (lightweight) static and dynamic analysis. Figure 1 presents the high level view of our system design. On the static analysis side, we obtain API-calls present in the app code as artifacts and use them to train a LSTM classifier (or its variant). On the dynamic analysis side, we collect (via an emulator) system calls invoked by the app during runtime as the artifacts and feed them to a similar LSTM classifier. Note that API-calls (and system calls) are quasi-sequential (and sequential, respectively) artifacts. As the input dataset is sequential in nature, we use LSTM and its variants to learn from such sequences [13].We merge the vetting decision of static analysis side and that of the dynamic analysis side via a simple (probability) aggregation scheme as shown in Figure 1. Our experimental results show that this hybrid approach makes the classification accuracy even better. Moreover, we experiment with variants of LSTM models such as bidirectional LSTM (Bi-LSTM), attention-based bidirectional LSTM (Attn-BiLSTM) and compare the performance results. Furthermore, we compare the performance of these DL models with prior models based on traditional ML algorithms.

The main contributions of this paper are as follows: **(i)** We design and implement a DL-based hybrid classifier for security vetting of Android apps, which leverages automated feature engineering as well as the complimentary strengths of static and dynamic analysis. **(ii)** Via extensive experiments,

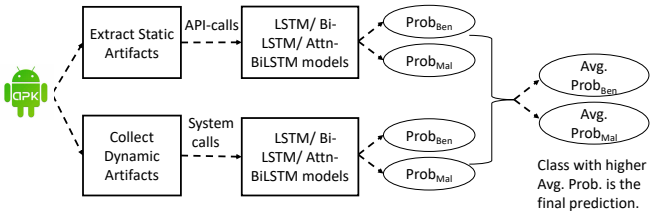

Fig. 1. System design of the hybrid classifier for Android app vetting: Prediction probabilities from both the static and dynamic artifacts-based deep learning (DL) models are used to derive the final prediction.

we evaluate the performance of the vetting system. The results show that the DL models based on static artifacts only (or on dynamic artifacts only) yield high precision and recall. The classification accuracy is further enhanced with the hybrid classifier, which yields near-to-perfect accuracy with area under the precision-recall curve (AUC) being 0.9998. **(iii)** The experimental results show that among the variants of the LSTM technology used Attn-BiLSTM performs best. **(iv)** Additional experiments indicate that our vetting system is fairly robust against imbalanced data and it scales with dataset size.

II. Background

As in the literature, by *system calls* we refer to low-level functions implemented in the kernel. In particular, OS-level services such as hardware related services (*e.g.*, accessing a memory location, a file, *etc*.), execution and creation of new process, *etc*. are implemented in the kernel, and an app can avail such a service by invoking a system call. Obtaining such system calls invoked by an app can might provide real (regardless of code obfuscation, if any) picture of what the app does.

Machine learning frameworks build mathematical models using training data to perform classification tasks. Words/strings input need to be represented numerically, as vectors, before feeding them to a ML/DL model. The representation of a ML/DL's input feature as a vector comprising of real numbers is called *feature vector*. The process of mapping a feature to a vector of a certain dimension is called *word embedding*. Frequency count of words, one hot encoding, TF-IDF vectorization, *etc*. are methods to do this. Mikolov *et al.* [14] compared aforementioned methods with embedding words to vector space using Continuous Bag of Words (CBOW) and Skipgram algorithms. CBOW or Skipgram achieve state-of-the-art performance, preserving the semantic similarity among the words/strings. In the current work, we use the gensim [15] library's *word2vec* API to perform word embedding.

An LSTM network [13] is a recurrent neural network (RNN), whose architecture consists of a recurrent cell that enables information to be passed from one time step to the next one. Similar to an RNN, an LSTM unfolds into a sequence of recurrently connected memory blocks. The main difference is that the LSTM cell contains a cell state (long term memory), in addition to the hidden state (short term memory), and

also three gates (an input gate, a forget gate and an output gate). Each gate is associated with a sigmoid activation and a point-wise multiplication operation[16]. The gates control which information from the previous cell state is to be retained or forgotten, which information is to be updated, and which information is to be output as the current cell state.

Bi-LSTM extends the single forward direction layer of basic LSTM by using another LSTM layer where hidden states flow in reverse temporal order. With Bi-LSTM, the model can learn from all available past and future information [17]. Using depth concatenated hidden states of both forward LSTM and backward LSTM, it is possible to learn from both the past and the future sequences [17].

A Bi-LSTM's output states can be fed to a fully connected layer to learn the attention weights, which helps the model focus on the most important components of the input with respect to the output. Use of such attention weights with the output states of Bi-LSTM yields Attn-BiLSTM. The learned attention weights of Attn-BiLSTM can also be potentially used for interpretation of the important input components. In this way, using Attn-BiLSTM, we can focus on the important features obtained from the Bi-LSTM instead of focusing on all outputs. We experiment with some of these variations of LSTM.

III. Related Work

A. Static Analysis and Dynamic Analysis

FlowDroid [2], Amandroid [3], DroidSafe [18], *etc*. are examples of the state-of-art static analysis tools for security vetting of Android apps. These tools typically track control and data flow in the app code and match the findings with a known set of malicious flow patterns. However, manual modelling of malicious signatures relies on human expertise whereas tracking control and data flows are computationally expensive operations both in terms of time and memory. On the other hand, TaintDroid [19] is a dynamic analysis tool.

Literature [20] [7] suggests that use of static or dynamic artifacts to train machine learning models for classifying malicious apps and benign apps can yield better results. Moreover, with feature engineering capability of deep neural networks, human hours devoted to manual modelling of malicious signatures can be avoided. Furthermore, android security experts cannot keep up with the rapidly changing android ecosystem. To keep up with changing android ecosystem and to save human hours, machine learning approaches were introduced.

B. Machine Learning and Deep Learning

MAMADROID [7] uses a machine learning classifier using the features obtained by abstracting the API-calls collected from the call graph of the android app. The authors claim it to be better performing than DroidAPIMiner [21] which uses frequency of API-calls used by the app. However, when a one-time trained MAMADROID model is used to perform classification over the years, its efficacy decreases [7].

Drebin [6] uses any feature that could be obtained from an app's source code and Manifest file as artifact, which results in over $500,000$ static features. Standard machine learning algorithms are trained based on these features to perform android apps classification. Roy *et al.* [5] demonstrated that by selecting only 471 features out of the $500,000$ static features used by Drebin, similar results can be obtained.

Success with use of APICalls and machine learning models also motivated researchers to use other features such as simplified bytecode. Xu *et al.* [22] used both traditional machine learning models and deep learning models to perform android classification. Authors claim that the use of stacked LSTM on bytecode level helps learn bytecode semantics at app level.

Karbab *et al.* [20] use API-calls in their deep learning (convolutional neural network) based android malware detection framework MalDozer. Authors claim that MalDozer is resilient to the ordering of API calls of an app.

Furthermore, Hou *et al.* [23] built a vetting system known as Deep4MalDroid, which uses system calls. The system calls were used to construct a graph with weighted nodes and edges. The constructed graph based features serve as inputs for classification using Stacked AutoEncoders (SAEs). Authors claim that their deep learning model with three layers of SAEs outperforms other existing shallow learning models.

LSTM network can only learn from past data/sequences and not from future sequences [24]. Schuster *et al.* [24] proposed Bi-LSTM to learn from all available past and future information. Further, Zhou *et al.* [17] demonstrated the effectiveness of Attn-BiLSTM on the SemEval-2010 relation classification task.

Let us summarize the similarity and difference of our work with/from others. Similar to [20] [7], we use API-calls as the static analysis artifacts. However, unlike [20] [7], we use LSTM and variants of LSTM to build our models. Similar to Deep4MalDroid [23], our work utilizes system calls as dynamic artifacts. However, we use the original sequence of system calls whereas Deep4MalDroid uses a weighted system call graph. Another big difference of our approach from these prior works is that prior works used either static analysis or dynamic analysis but not a hybrid one.

IV. Approach

In this section, we first present a motivating example to illustrate the usage of API-calls and system calls as artifacts. We then describe in high level our overall approach to doing hybrid analysis.

A. Motivating Example

Let us consider an example app that steals a SMS message from the local storage (*Content Provider* associated with SMS inbox) and leaks it to a remote server via the internet. Listing 1 illustrates a (Java) code fragment of the app, showing a SMS message getting leaked from a source point to a sink point (remote server).

We consider the API-calls in the app code as static artifacts. We consider an invocation of procedure p (say at code line number x) as an API-call if the definition of p is not available

in the app code (*e.g.*, if p is an Android library procedure). Table I lists the API-calls which are invoked in *onStartCommand* procedure. Note that the APIs are presented in JAWA [3] code (decompiled byte-code similar to Smali code). For the sake of clarity, in Table I, we do not show the arguments and return types associated with the API-calls. Note that invocation of *uploadSms* procedure in L9 (in Listing 1) is not an API-call because the definition of *uploadSms* procedure is present in the app's code. So, *uploadSms* invocation does not show up in Table I.

TABLE I
API-CALLS INVOKED WITHIN ONSTARTCOMMAND PROCEDURE

Code Block #	JAWA Line #	API-calls
1	c9f8	Landroid/net/Uri;.parse
1	ca12	Landroid/content/ContentResolver;.query
1	ca1a	Landroid/database/Cursor;.moveToFirst
2	ca2a	Landroid/.../Cursor;.getColumnIndexOrThrow
2	ca32	Landroid/database/Cursor;.getString
3	ca42	Ljava/lang/String;.replace
3	ca50	Landroid/app/Service;.onStartCommand

```
public int onStartCommand(Intent i, int flags,int startId){
   L2 : String str = "";
   L3 : Uri inboxURI = Uri.parse("content://sms/inbox");
   L4 : Cursor cur = getContentResolver.query(inboxURI,null,...)
   L5 : if (cur.moveToFirst()){
   L6 :   str = cur.getString(cur.getColumnIndexOrThrow("body"));
   L7 : }
   L8 : str=str.replace(" ","_");
   L9 : uploadSms(str);
   L10: return super.onStartCommand(intent,flags,startId);
}
```

Listing 1. A code fragment from the SMS-stealer app.

The motivation of using API-calls as artifacts is that the list of API-calls may constitute a signature of what the app does. In Table I, we see that *ContentResolver* is queried and a cursor is obtained to extract string data from the SMS inbox, which is then modified using the *replace* API-call. Similarly, we list API-calls (not shown here due to brevity) corresponding to *uploadSms* method, which suggests that a *HttpURLConnection* is set up and the SMS string is added to the URL header. Individually, any of these API-calls may not bear much information, but the whole sequence of API-calls can represent a signature of the app (*e.g.*, SMS leakage).

Absolutely speaking, Table I does not represent a flat sequence of API calls in *onStartCommand* procedure. In reality, these API-calls are in multiple *code blocks* (as defined in the literature [20]). So, real execution might result in a different order of the code blocks (though inside a code block the order is fixed). To track the true sequence of code-blocks, we need to build the (inter-procedural) control flow graph (CFG) of the whole app, which is computationally expensive. In this paper, we limit ourselves to lightweight static artifacts, and we consider the list of API-calls as they appear in the code of a procedure. This results in us collecting a quasi-sequence of API-calls, where the true sequence is maintained only within a code block.

In nutshell, the static artifacts of an apk is represented as a text file that contains one paragraph of text (where an API-call is considered as a word) for each (Java) class of the app. Furthermore, if a class c has a procedure p, then the API-calls of p makes one sentence in the corresponding paragraph of c. Standard delimiter are used to separate neighboring words, sentences, or paragraphs.

On the dynamic analysis side, we consider system calls as the artifacts. To demonstrate the usage of system calls as artifacts, let us consider another example app named ReadFromWeb, which attempts to read a webpage at a given URL and writes it to a local file. The app uses Java library's URL.openStream(), BufferedReader and InputStreamReader functionalities to access the webpage as a string, and then uses PrintWriter.println to write it to a file. Table II presents the system calls as recorded by Linux utility tool *Strace*. For the sake of clarity, in Table II, we use '. . .' to replace lengthy arguments.

TABLE II
SYSTEM CALLS INVOKED ON EXECUTION OF READFROMWEB APP

execve("/usr/bin/java", ["java", "ReadFromWeb"], . . .) = 0
brk(NULL) = 0x17e0000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT
readlink("/proc/self/exe", "jvm/java-8-oracle/jre/b". . ., 4096) = 39
... (more)
stat("/usr/lib/jvm/java-8-oracle/.../libjvm.so", {st_mode= . . .}) = 0
futex(0x7f137d1080c8, FUTEX_WAKE_PRIVATE, 2147483647) = 0
openat(AT_FDCWD, . . .) = 3
read(3, "\177ELF\0\0\0\0\0\0", . . ., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=17022752, . . .}) = 0
mmap(NULL, . . ., 3) = 0x7f137bb30000
mprotect(0x7f137c812000, 2097152, PROT_NONE) = 0
mmap(0x7f137ca12000, . . ., 0xce2000) = 0x7f137ca12000
mmap(0x7f137cad9000, . . ., -1) = 0x7f137cad9000
close(3)
... (more)
futex(0x7f137d7449d0, FUTEX_WAIT, 9588, NULL) = 0
exit_group(0) = ?

From Table II we observe that the Dalvik bytecode is passed to the *execve* function for execution. Then heap is extended by invoking *brk* function and relevant library files are accessed if possible. The control waits for the import task to complete and then protects a region of memory to be mapped with the new file. Once the file operation is performed, the program exits successfully. In the current work, we filter the *Strace* output (as obtained above) whereas we only keep the system calls invoked by the app (*i.e.*, we drop the associated arguments and return values). We stress that the collected sequence of system calls does represent a true time series.

B. Extracting Artifacts from Apps for Deep Learning

As illustrated in Figure 1, we independently extract static artifacts (API-calls) as well as dynamic artifacts (system calls)

from an android app. To collect API-calls, we adopt an open-source tool named Amandroid [3]. To collect dynamic analysis artifacts, each app from the dataset is executed in an emulator where we use Linux command line tool *Strace* to record the the sequence of system calls invoked by the app during execution.

C. Deep Learning Process

As illustrated in Figure 1, we process static artifacts and dynamic artifacts independently. On the static side, we represent the artifact of apps by a set of lists where each list represents API-calls associated with one app. We use this data to perform word embedding for each API-call and then use the word embedded vectors as input to the LSTM model. In our experiment, we feed the apps in random order to the LSTM model. We build, train, perform hyper-parameter tuning, and test the models on AWS using Keras, a deep learning library, which runs on top of TensorFlow 1.14. We follow a similar approach for dynamic analysis. We do similar preprocessing, but we trim the system calls such that all the return types and arguments are ignored. We follow a similar approach for word embedding, model generation, training, and testing.

D. Hybrid Analysis

Independently using the static analysis or dynamic analysis based models, we obtain the probability of prediction of an app x for the two classes (malicious and benign). As illustrated in Figure 1, we compute the hybrid probability of prediction of the app x being malicious (or benign) as the average of probability of prediction for x being malicious (or benign) from the two models. Finally, the hybrid classifier predicts the output class of app x as the one with higher probability. For example, if static analysis model predicts the probability of app x to be malicious as 0.6 and benign as 0.4, and dynamic model assigns 0.7 for malicious and 0.3 for benign, we take $(0.6 + 0.7)/2 = 0.65$ as probability for x to be malicious. Similarly, for app x to be benign we get probability of $(0.4 + 0.3)/2 = 0.35$. Since the probability of x to be malicious is greater than that to be benign, the hybrid analysis predicts x to be malicious. Finally, the hybrid classifier predicts the output class as the one with higher probability.

V. Implementation

Here we discuss the experimentation environment, dataset preparation, and implementation of the deep learning based vetting system.

A. Experimentation Environment

To obtain static artifacts from apps, we run the extractor program on a *i2.8Xlarge* instance (on AWS) with 32 cores in parallel. To collect dynamic artifacts (*i.e.*, system calls), we use Genymotion Android Emulator (which is available as Product as a Service on AWS) to execute the android apps: We used android 7.0 (Nougat) to execute the apps and record the system calls. To build the deep learning system, we use EC2's Ubuntu 18.04 Deep Learning AMI (V25.3) loaded in a *g3.8Xlarge* instance. This type of instance uses 32 vCPUs, 244 GiB memory, and 150 GiB SSD. We configure the instance with port-forwarding so that we can connect to Jupyter notebook from the local machine. We store the set of files containing static and dynamic artifacts in an AWS S3 bucket from where we load them to EC2 instance when necessary. We used the same EC2 instance for all experiments.

B. Dataset Preparation

Below we discuss how we prepare the app dataset, and how we extract artifacts to be used in the deep learning phase.

1) Preparing Apk Dataset

Android Malware Dataset (AMD) [25] is comprised of about $25K$ apks dating from 2010 to 2016. AMD is the source of the malicious apps in our dataset. Furthermore, AndroZoo [26] is a continually growing dataset, currently having more than 10 million apps. AndroZoo is the source of the benign apps in our dataset. To establish ground truth we double check the status of an app x with VirusTotal [27] before including x in our malicious or benign dataset. In particular, each app x is fed to VirusTotal, which generates a report (a json file) comprising of individual reports from about 50 anti-malware software (as used by VirusTotal). We use the reports corresponding to app x to determine the status of x: We consider x to be benign only if no anti-malware in VirusTotal flags x as malicious; we consider x to be high qualility malicious app if majority (*i.e.*, more than half) of antimalware products flag x as malicious; otherwise, we consider x as a low quality malicious app.

For experimentation we build multiple datasets whereas the difference in them is the quality of ground truth of malware apps. In particular, *Dataset1* consists of high quality malware apps along with benign apps whereas *Dataset2* consists of low quality malware apps along with benign apps. *Dataset3* contains larger number of apps, and it has both high and low quality malware apps along with benign apps.

Out of all apps from AMD dataset, $12,063$ apps were deemed as high quality malicious by the aforementioned majority voting scheme. We randomly selected only $5,617$ apps from these $12,063$ apps to include in *Dataset1*. Similarly, out of $56,300$ (randomly selected) AndroZoo apps, we find $36,114$ coming out as clear benign through VirusTotal. We randomly selected only $12,610$ apps from these $36,114$ to include in *Dataset1*. We had to limit ourselves to this smaller dataset (malicious apps and benign apps) for experimentation due to monetary constraint as the process of extracting dynamic artifacts is expensive (as explained later). However, we used a bigger dataset (*i.e.*, *Dataset3*) for the static analysis experiments when necessary (as mentioned later).

Out of all apps from AMD dataset, $12,436$ apps were deemed as low quality malicious by the aforementioned majority voting scheme. We randomly selected only $5,617$ apps from these $12,436$ apps to include in *Dataset2*. Regarding the benign apps, *Dataset2* has the same benign apps (*i.e.*, $12,610$ in count) as in *Dataset1*. On the other hand, we include all AMD apps $(24,406)$ and all the aforementioned $36,114$ benign apps to build *Dataset3*. For the sake of brevity, in the

rest of the paper, unless explicitly mentioned, all experiments are run on *Dataset1*, and by dataset we refer to *Dataset1*.

2) Extraction of Static Artifacts

We introduce a new mode in the open source tool Amandroid to obtain the API-calls without performing more expensive operations, such as building CFG or DFG. This new mode decompiles the apk file, and collects API-calls. In particular, we track each class and the procedures within each class. We obtain the API-calls from a class as a paragraph where API-calls from each function in the class is a sentence and each API-call is a word in the sentence.

3) Extraction of Dynamic Artifacts

We execute each app on android 7.0 (Nougat) instance of Genymotion on AWS to obtain the sequence of system calls. Manually triggering random events would not be feasible. So, we automate the process as follows: We use Android Asset Packaging tool (AAPT) to obtain package information, then install the app using Android debug bridge (ADB), then fetch the process id associated with the app, and then use Strace to track the system calls while executing the app by providing random events through ADT Monkey tool. ADT Monkey causes random events such as swipe, clicks, keystrokes, *etc*. to expand the code coverage.

C. Deep Learning

As shown in Figure 1, we do deep learning on static and dynamic artifacts where the learning process is essentially the same. For the sake of brevity, let us discuss the deep learning process in general terms, which is agnostic to the type of artifact. Since the artifacts are sequential in nature, we opt to use the LSTM network, which is well known to capture sequential behavior of a dataset. Figure 2 presents the deep learning network with LSTM unrolled in time.

As the artifacts are in string format, we cannot directly provide them as an input to LSTM because LSTM can only work with numbers. So, we convert each word (*i.e.*, API call or system call) into a vector of real numbers. In particular, we use the *word2vec* algorithm to get the vector representation of a word, capturing the semantic meaning of that word. For example, system call (words) such as *socket* and *connect* are closely related, which might be reflected on their vector representation *i.e.*, vectors representing these two words could be highly similar.

The output of *word2vec* algorithm is an embedding matrix whose dimension is $[\#words, dim]$, where $\#words$ refers to the total number of distinct words (API-calls or system calls), and dim refers to the dimension of the embedding vector. This means, each row of this matrix represents the vector representation of a word. In our experiments, we use the CBOW model of *word2vec* algorithm, and we set dim to 100.

We implement the deep learning system using Keras in Jupyter notebook on AWS. We first set up an embedding layer. Instead of learning weights for embedding layer, we provide the embedding matrix (output of *word2vec* algorithm) to the embedding layer which is used to map each word in the artifact to its vector representation. This vector representation is passed as input for the LSTM network. From that point on, the flow of network is shown in Figure 2. We consider that input X has 4000 timesteps, and accordingly we reshape the input. The i-th timestep data ($X_{(i)}$) is fed to a LSTM cell and, so on. Note that the LSTM cell's each inner FC layer has 128 hidden units. Further, note that we studied the distribution of number of API-calls and system calls (of an app) in our artifacts dataset before we chose to set 4000 timesteps in the LSTM model.

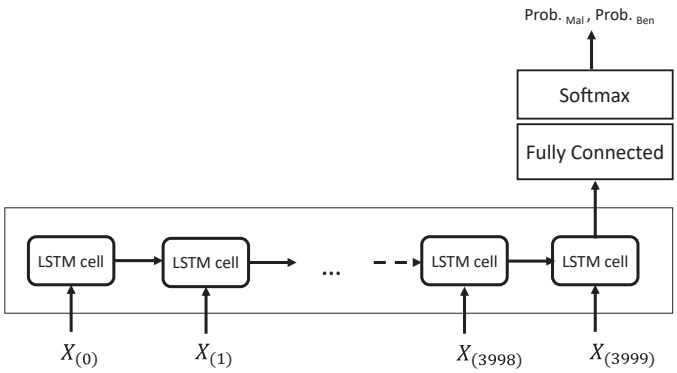

Fig. 2. Unrolled LSTM model with dimensions

Regarding hyperparameter tuning, we performed multiple experiments with several learning rates to know which learning rate gives the best result. We first divide the dataset into the train and test data, and then we further divide the train data into training and validation set. For each learning rate, we run training on the training set and validate using the validation set. To this aim, we use *scikit-learn*'s StratifiedShuffleSplit to generate random training and validation set for each learning rate. We maintain a batch size of 75. This way we found out that the learning rate of 0.01 gives the best result. We then use this learning rate to train the final LSTM model whose training data consist of combined training and validation set that we used in the above-mentioned experiment. Finally, we test the model with the test data that was left untouched in the above experiment.

Furthermore, we also apply other variants of LSTM, such as Bi-LSTM and Attn-BiLSTM. The implementation of Bi-LSTM and Attn-BiLSTM are straightforward extensions of LSTM: As in the case of Bi-LSTM we add an extra layer of backward cells to the LSTM network. We achieve this by adding the Bidirectional layer wrapper provided by Keras. For Attn-BiLSTM, we use attention weights and biases over Bi-LSTM. We do this by adding the SeqWeightedAttention layer (from the keras-self-attention package) on top of the bi-directional LSTM layer.

VI. Evaluation

In this section, we present the performance results of our vetting system, and when applicable, we compare the results with the prior work.

A. Spliting the dataset in train set and test set

We split the app dataset in a train set and a test set: The train set contains $4,617$ malicious and $10,610$ benign apps whereas the test set contains the remaining apps ($1,000$ malicious and $2,000$ benign apps). Table III shows the train-test split used. We remind the reader that we use the same train-test split for static and dynamic analysis.

TABLE III
TRAIN-TEST SPLIT

	Malicious	Benign
Train Set	4,617	10,610
Test Set	1,000	2,000
Whole *Dataset1*	5,617	12,610

B. Evaluation of Deep Learning Models

As discussed in Section V-C, during hyperparameter tuning, we observed that Adam optimizer with a learning rate of 0.01 yields best results. Note that only hyperparameter that we tune is the learning rate. We set the same hyperparameters for all the models (as discussed in the rest of the paper) to train and test. As shown in literature [5], area under precision-recall curve (AUC) is a good metric to evaluate the performance of a classifier. So, we choose to use mainly this metric in this paper. Higher the AUC, better is the model. Below we present the results obtained from static analysis, dynamic analysis, and hybrid analysis.

1) Evaluation of Static Analysis

All three models *i.e.*, LSTM, Bi-LSTM and Attn-BiLSTM based on static artifacts are trained and tested on the same dataset.

Comparison of Static Analysis Models. We summarize the results obtained from the models trained and tested on static artifacts in Table IV. AUC as shown in Table IV suggests that the DL models are able to learn better when sequence-orders are considered from both forward and backward direction in Bi-LSTM. Use of attention weights along with Bi-LSTM yields the best result out of the three models.

TABLE IV
COMPARISON OF STATIC ANALYSIS RESULTS

DL Model	Precision	Recall	F1	AUC
LSTM	0.9470	0.9130	0.9297	0.9779
Bi-LSTM	0.9454	0.9190	0.9320	0.9823
Attn-BiLSTM	0.9897	0.9690	0.9792	0.9968

Comparing with the Prior Work (ML Algorithm). Roy *et al.* used 471 static analysis features with traditional ML models to classify android apps. We use Roy *et al.*'s feature extraction engine to extract those 471 features from our apk dataset, and use the same train-test split. We present the results obtained in Table V. We stress that even if the ML algorithm's result looks competitive with Table IV, ML algorithm relies on human expertise to *manually* select the features. In contrast, DL algorithms learn the features themselves, which aids to automation and scalability.

TABLE V
STANDARD ML ALGORITHMS WITH 471 FEATURES USED BY ROY *et al.*

ML Model	Precision	Recall	F1	AUC
Bernoulli Naive Bayes	0.8430	0.6912	0.7596	0.8558
K-Nearest Neighbor (K=5)	0.9025	0.9837	0.9414	0.9574
Support Vector Machine	0.9734	0.9252	0.9487	0.9819

2) Evaluation of Dynamic Analysis

Similar to evaluation of the static artifacts based models, we compare the results obtained from training LSTM, Bi-LSTM and Attn-BiLSTM using dynamic artifacts.

Comparison of Dynamic Analysis Models. We summarize the results obtained from the deep learning models trained and tested on dynamic artifacts in Table VI. We observe that the results are similar to those obtained from static artifacts based models. Attn-BiLSTM does the best job with only 1 false alarm and 16 missed alarms. However, Bi-LSTM gives the best area under the curve, the difference being 0.0002 $(0.9973 - 0.9971)$ as compared with Attn-BiLSTM.

TABLE VI
COMPARISON OF DYNAMIC ANALYSIS RESULTS

DL Model	Precision	Recall	F1	AUC
LSTM	0.9979	0.9730	0.9853	0.9932
Bi-LSTM	0.9969	0.9820	0.9894	0.9973
Attn-BiLSTM	0.9989	0.9840	0.9914	0.9971

3) Evaluation of Hybrid Analysis

Comparison of Hybrid Analysis Results. We present the results obtained using the hybrid approach in Table VII. We observe that hybrid approach to classification outperforms static analysis and dynamic analysis.

TABLE VII
COMPARISON OF HYBRID ANALYSIS RESULTS

DL Model	Precision	Recall	F1	AUC
LSTM	0.9969	0.9760	0.9863	0.9978
Bi-LSTM	1.0	0.9820	0.9909	0.9993
Attn-BiLSTM	0.9979	0.9930	0.9954	0.9998

We observe that on the AUC metric, Bi-LSTM outperforms LSTM, and Attn-BiLSTM does better than both LSTM and Bi-LSTM.

C. Experimenting with Imbalanced Data

So far, the number of malicious apps and benign apps in the test dataset in our experiments was fixed as shown in Table III. To test out the performance of our vetting system on imbalanced data, we perform the following experiment. We create multiple test dataset by varying the number of malicious apps as 200, 400, 600, 800, 1000 while keeping the number of benign apps at 2000. This allows us to vary the imbalance (malicious to benign) ratio from 1:2 to 1:10. Figure 3 shows the performance results of our static artifacts based LSTM

model on the above varying datasets. The more imbalanced the test data more the challenge the classifier faces in keeping a good AUC score. However, we observe that our static artifacts based LSTM model is fairly robust against imbalanced data.

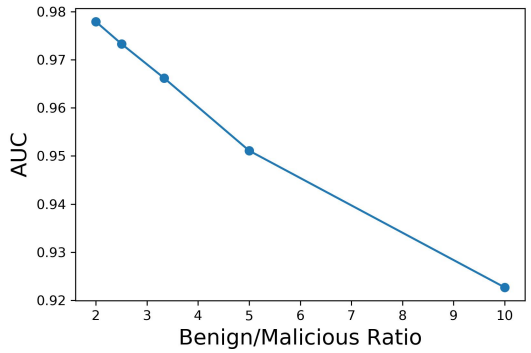

Fig. 3. Vetting accuracy of static artifacts based LSTM model on imbalanced data (varying benign and malicious app ratio in the test data.)

D. Impact of malware quality on vetting accuracy

We performed an experiment with our static artifacts based LSTM model on *Dataset2*, which consists of benign apps and low quality malicious apps. The results are shown in Table VIII. We observe that our model performs fairly well in identifying low quality malware apps. However, the accuracy is significantly lower than what we got before. The results also show that accuracy of a classifier depends not only on the model design but also on the quality of data. This point the community should keep in mind when comparing two published papers' (reporting android app vetting performance) models.

TABLE VIII
OUR STATIC MODEL'S ACCURACY ON LOW QUALITY DATASET

DL Model	Precision	Recall	F1	AUC
LSTM	0.9333	0.868	0.899	0.9588

E. Scalability of our vetting system

Regarding scalability, only possible concern is the training phase as the testing phase takes no significant time in comparison. Recall that the training set of *Dataset1* consists of $15,227$ apps. It took 2.21 hours for the training of static artifacts based LSTM model to complete. To investigate scalability, we then trained the LSTM model on *Dataset3*, which is a much larger dataset. The training set of *Dataset3* consists of $52,520$ apps ($21,406$ malicious + $31,114$ benign), and it took 5.93 hours for the training to complete. The above indicates that the training time has increased linearly only. Also, note that the static artifacts based LSTM model gave us 0.97 AUC for *Dataset3*.

F. Experiments with Malware Apps from Prior Work

We have received the Maldozer (malware) dataset from the MalDozer research group [20]. We have built an experimental dataset by combining ($5,617$ random samples) from the above apps with our same $12,610$ benign apps (as in $Dataset1$). We extracted static features and ran our deep learning vetting system on this experimental dataset. The results are shown in Table IX. Maldozer group reported that their DL model [20] achieves an F1-Score of 0.96 with false positive of 0.06.

TABLE IX
OUR STATIC MODEL'S ACCURACY ON MALDOZER DATASET

DL Model	Precision	Recall	F1	AUC
LSTM	0.9463	0.9340	0.9401	0.9821
Bi-LSTM	0.9761	0.9840	0.9800	0.9983
Attn-BiLSTM	0.9900	0.9960	0.9930	0.9998

VII. CONCLUSION AND FUTURE WORK

In this work, we used API-calls and system calls to train deep learning models for security vetting of Android apps. In particular, as the deep learning technology we experimented with LSTM and its variants (Bi-LSTM and Attn-BiLSTM). Individual models trained on static artifacts and dynamic artifacts showed that Attn-BiLSTM models yield better results than Bi-LSTM and LSTM. We designed a hybrid approach to combine the vetting decision of a static artifacts based model and a dynamic artifacts based model. Hybrid Attn-BiLSTM was the best model that we built, which yielded a near-perfect classification accuracy.

Limitations of the current work include: (a) The static artifacts (API-calls) do not form a true time-series, which means some information is getting lost before reaching the LSTM model. (b) We did not yet study the semantics of the *attention* in the attention-based models.

As part of future work, we will investigate some of the weights produced by the *attention* layer and will identify to what *words* they correspond. Thus, we can add some semantic to the predictions. As other future work, we look forward to incorporating noise in the training data and also detecting zero day malware. Also, we want to experiment with other artifacts, such as using the entire app bytecode/resource files. Furthermore, in the future, we would like to experiment with other approaches to hybrid classification such as feeding both static and dynamic artifacts together (with back-propagation) to train the model (as opposed to current hybrid classifier, which merely combines end results from static and dynamic artifacts based models) and more.

ACKNOWLEDGMENT

This work has been partially supported by the U.S. National Science Foundation (NSF) under grant no. 1718214, 1717871, and 1717862. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Statista, "Mobile OS Market Share," 2020. [Online]. Available: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," *SIGPLAN*, pp. 259–269, 2014.

[3] F. Wei, S. Roy, X. Ou, and R. Robby, "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps," *ACM Transactions on Privacy and Security*, pp. 1–32, 2018.

[4] S. Roy, D. Chaulagain, and S. Bhusal, "Book Chapter: Static Analysis for Security Vetting of Android Apps," in *From Database to Cyber Security: Essays Dedicated to Sushil Jajodia on the Occasion of His 70th Birthday*. Springer International Publishing, 2018, pp. 375–404.

[5] S. Roy, J. DeLoach, Y. Li, N. Herndon, D. Caragea, X. Ou, V. P. Ranganath, H. Li, and N. Guevara, "Experimental study with real-world data for android app security analysis using machine learning," in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, 2015, pp. 81–90.

[6] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," in *Symposium on Network and Distributed System Security (NDSS)*, 2014, pp. 23–26.

[7] L. Onwuzurike, E. Mariconti, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models (extended version)," *ACM Transactions on Privacy and Security*, pp. 1–31, 2019.

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. The MIT Press, 2016.

[9] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, pp. 436–444, 2015.

[10] R. Vinayakumar, S. Kp, P. Poornachandran, and S. Kumar S, "Detecting Android Malware using Long Short-term Memory," *Journal of intelligent and fuzzy systems*, pp. 1277–1288, 2018.

[11] Y. Bengio, P. Simard, and P. Frasconi, "Learning Long-term Dependencies with Gradient Descent is Difficult," *IEEE Transactions on Neural Networks*, pp. 157–166, 1994.

[12] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: continual prediction with lstm," in *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, 1999, pp. 850–855.

[13] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, pp. 1735–1780, 1997.

[14] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," in *ICLR Workshop*, 2013.

[15] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, 2010, pp. 45–50.

[16] C. Olah, "Understanding LSTM," 2015. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[17] P. Zhou, W. Shi, J. Tian, Z. Qi, B. Li, H. Hao, and B. Xu, "Attention-Based Bidirectional Long Short-Term Memory Networks for Relation Classification," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 2016, pp. 207–212.

[18] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information Flow Analysis of Android Applications in DroidSafe," in *Symposium on Network and Distributed System Security (NDSS)*, 2015, pp. 110–126.

[19] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010, pp. 393–407.

[20] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "MalDozer: Automatic Framework for Android Malware Detection using Deep Learning," *Digital Investigation*, pp. S48–S59, 2018.

[21] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android." in *International conference on security and privacy in communication systems*, 2013, pp. 86–103.

[22] X. Ke, Y. Li, R. H. Deng, and K. Chen, "DeepRefiner: Multi-layer Android Malware Detection System Applying Deep Neural Networks," in *2018 IEEE European Symposium on Security and Privacy*, 2018, pp. 473–487.

[23] S. Hou, A. Saas, L. Chen, and Y. Ye, "Deep4MalDroid: A Deep Learning Framework for Android Malware Detection Based on Linux Kernel System Call Graphs," in *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*, 2016, pp. 104–111.

[24] M. Schuster and K. Paliwal, "Bidirectional Recurrent Neural Networks," *IEEE Transactions on Signal Processing*, pp. 2673–2681, 1997.

[25] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep Ground Truth Analysis of Current Android Malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*, 2017, pp. 252–276.

[26] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: Collecting Millions of Android Apps for the Research Community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 468–471.

[27] "VirusTotal: Analyze Suspicious Files and Urls," 2020. [Online]. Available: https://www.virustotal.com/