Darwinian Malware Detectors: A Comparison of Evolutionary Solutions to Android Malware

Zachary Wilkins
Dalhousie University
Computer Science
Halifax, Nova Scotia
zachary.wilkins@dal.ca

Nur Zincir-Heywood
Dalhousie University
Computer Science
Halifax, Nova Scotia
zincir@cs.dal.ca

ABSTRACT

Google's Android platform dominates the smartphone world, but this tremendous popularity has led to an appealing target for malicious actors. This is especially troublesome because the diverse nature of Android devices, and the modifications that are made by each manufacturer, make consistency in system software difficult across all devices. As new attack and evasion behaviours emerge, security researchers work to create more sophisticated detection systems. Many of these systems are based on machine learning approaches. An especially promising avenue that is being actively researched is the use of evolutionary systems, where detectors are bred, rather than built. In this paper, we examine two such evolutionary systems, training them on established datasets before comparing their performance on large, unknown datasets. Our experiments demonstrate that these systems are effective in predicting contemporaneous and evolved malicious applications, and meet or exceed the results of a state-of-the-art, rule-based system.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; • **Computing methodologies** → *Genetic algorithms*;

KEYWORDS

Cyber security, Machine learning, Malware, Mobile security, Smartphone, Android, Cyber attack, Evolution

ACM Reference Format:
Zachary Wilkins and Nur Zincir-Heywood. 2019. Darwinian Malware Detectors: A Comparison of Evolutionary Solutions to Android Malware . In *Genetic and Evolutionary Computation Conference Companion (GECCO '19 Companion), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3319619.3326818

1 INTRODUCTION

In the world of smartphones, two operating systems dominate: Google's Android platform and Apple's iOS [5]. While iOS is restricted to Apple devices[1]and tightly controlled by the company, Android is available on devices from many manufacturers, and frequently modified to suit their purposes. This flexibility and resulting variability, combined with the large install base, creates many opportunities for malicious actors to exploit the platform [2]. The number of malicious applications for Android are steadily increasing and evasion techniques are becoming evermore sophisticated [19].

Accordingly, the security community has responded by employing a variety of techniques to detect these malicious applications before they can infect the devices of users. Using machine learning approaches has been successful [3] [12] [8] [17], as detectors need to be able to make reasonable predictions about current and future samples.

One area in particular that has shown promise is solutions that evolve, rather than build, detection systems. By employing evolutionary techniques such as genetic algorithms, detectors can be sufficiently nuanced to accurately discern malicious and benign applications [11] [10].

Taking this idea a step further, research has been conducted to create an "arms race" by evolving Android malware alongside malware detectors [2] [15], which can make even more accurate predictions about potential future malware.

In this paper, we evaluate an evolutionary and a co-evolutionary system against a state-of-the-art, rule-based system, namely Assemblyline. The evolutionary systems are trained on well-established malicious Android datasets, as well as benign applications, before being tasked with identifying large, unknown datasets. The results indicate that co-evolutionary solutions are indeed more capable than similar evolutionary solutions, and that both evolutionary and co-evolutionary systems are competitive with rule-based static analysis on the datasets employed.

The rest of this paper is organized as follows. We begin by surveying related research in Section 2. Following this, we discuss the detection systems and the datasets used in Section 3. We then present and comment on the results of the experiments in Section 4. Section 5 concludes our observations and suggests directions for future work.

2 RELATED WORK

Machine learning approaches are being actively employed to improve the performance of malware detectors, both on desktop computers and mobile devices. We begin by reviewing more general machine learning based approaches before focusing on evolutionary mobile malware detection.

[1]https://www.apple.com/ios/ios-12/

GECCO '19 Companion, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6748-6/19/07…$15.00
https://doi.org/10.1145/3319619.3326818

David et al. introduce a malware detection system, DeepSign, that is idempotent to small scale changes in malware samples [3]. DeepSign employs deep neural networks, inspired by work in computer vision to recognize the same object from many angles and in different environments. In a sandbox, the behaviour of the application is recorded and represented as a vector which indicates the presence of the top unigrams. This vector is then fed to the deep belief network (DBN), where it is mapped through many layers to produce the malware signature. The trained DBN model could classify 98.6% of the testing data correctly. The authors also employ Support Vector Machine (SVM) and k-Nearest Neighbour (k-NN) classifiers as a comparison, which perform less successfully at 96.4% and 95.3%, respectively.

Milosevic et al. employ many different machine learning algorithms to identify malicious Android applications [12]. They use Android permissions and source code as features for evaluating the performance of classification and clustering. The classification algorithms applied include SVM, C4.5 Decision Trees, Random Forests (RF), Bayesian Networks, JRip, and Logistic and Linear Regression. Clustering is done through a bag-of-words model, where unigrams are extracted from the merged source code and simple k-means, Farthest-First, and Expectation-Maximization algorithms are applied. They also apply ensemble learning by grouping various algorithms and determining the result through majority voting, though the performance gains here are statistically insignificant. The classification approaches are very successful, with the highest F-scores for permissions and source code being SVM at 87.9% and 95.1%, respectively. The clustering approaches are not as effective, failing to achieve F-scores greater than 82.3%.

Liu et al. build a system for classifying and clustering malware using ensemble voting [8]. They begin by creating an image representation of each malware binary, as well as extracting trigram opcodes and imports. After reducing the dimensionality of features using information gain, the samples are passed through a variety of classifiers, such as RF, k-NN, Gradient-Boosting (GB), Naive Bayes, Logistic Regression, SVM, and Decision Trees. A label is assigned through a weighted combination of these results. The samples can then be clustered into malware families using the shared nearest neighbour algorithm. Their experiments show that this is a highly effective approach, with a best accuracy of 98.9% in classification achieved with RF, while clustering new malware attains an accuracy of 86.7%.

Ucci et al. undertake a comprehensive review of machine learning techniques for mobile malware analysis [17]. Specifically, they identify trends in the objectives, features, and algorithms of malware analysis. Especially of interest to us is their summation of the current state for predictions of future variants of malware. The authors specifically note that this is an area that lacks investigation and more research may yield useful information. Evolutionary solutions are well suited to this, as they have the ability to evolve new solutions.

Meng et al. perform an extensive study into the successes of commercial, academic, and online app store anti-malware tools (AMTs) for Android by creating a system called Mystique [11]. The approach focuses on codifying attack and evasion features into basic reusable components. By choosing these features through an evolutionary algorithm, effective malware can be generated. Their effectiveness is judged by their maximizing aggressiveness while minimizing evasiveness and detectability. These evolved malware are significantly harder to detect, with their tested detection rate dropping from an average of 73.3% to 11.96% when comparing the unevolved and evolved malware on a variety of detectors. They then include dynamic loading as an evasion technique in their follow up work [19].

Martín et al. develop an evolutionary malware detection system, MOCDroid, that is designed to be hardened against malware obfuscation [10]. It does so by analyzing third party import terms in Android applications, as these are difficult to obfuscate without breaking compatibility. After extracting import terms using a decompilation tool, the import statements are clustered in a similar manner to text mining, using the k-means algorithm. A classifier is trained on these clusters to maximize accuracy and minimize false-positives. This results in malicious and benign models, to which fresh samples can be compared. The resulting classifier is very successful, with accuracy rates of up to 95.2% and false-positives rates as low as 1.7%. Ten malware detection engines are used as a baseline, which never exceed 83% accuracy.

Bronfman-Nadas et al. strive to create a simulated competition, An Artificial Arms Race, between malware and detectors [2]. Hereafter, we refer to this system as ArmsRace. The goal of each component of ArmsRace is in opposition to the other, where the malware attempt to be undetectable and the detectors attempt to identify these undetectable applications. The malware are built in a similar fashion to Mystique, as an evolutionary computation technique is employed to maximize and minimize particular malicious aspects of the application by selecting appropriate features. The detectors are created using linear genetic programming, and represented as a sequence of instructions. By having these components work against each other in a feedback loop, an adversarial relationship is formed where they cannot be optimal simultaneously. The detectors created in this process are as accurate as comparable detectors without co-evolution (92%), but are much less complex, employing fewer features and instructions [2].

Sen et al. craft a fully automated co-evolution system, where malware are generated from existing samples and a detection system is implemented with genetic programming, to develop robust evasion and detection capabilities [15]. After converting Android applications to smali, the assembly language for the Android Runtime, genetic programming is employed to evolve the applications. These are then repackaged and deployed in an emulator for fitness evaluation. The detection system takes into account 146 features, such as API calls, permissions, and other static attributes. The detection systems are then evolved with genetic programming, with an eye on minimizing false-positive rates. These co-evolved anti-malware tools are highly effective, detecting 100% of the testing samples and performing similarly well on unseen data (91.3% – 99.4%), with an acceptable false-positive rate (7.4%).

Given the successes of the evolutionary approaches in this research, we focus on the ArmsRace and MOCDroid systems in our pursuit to extend the malware families they are capable of detecting. These solutions perform similarly well, and both have publicly available source codes [2] [9]. In addition, choosing these solutions provides us the opportunity to compare evolutionary and co-evolutionary detection systems.

3 METHODOLOGY

In this section, we describe the detection systems and datasets employed in these experiments.

3.1 Detection Systems

3.1.1 MOCDroid. As previously mentioned, MOCDroid is an evolutionary malware detection system that is designed to be hardened against malware obfuscation [10]. This is a significant concern in static analysis, as obfuscation is an effective tool to avoid detection.

By analyzing third party import terms in Android applications, MOCDroid can group commonly used import terms into specific behaviours. These third party import terms are essential to counteracting obfuscation, because they are difficult to mask without breaking compatibility with any external libraries.

The process begins with decompilation of the application using Jadx[2]. Jadx reproduces the source code to the application, and from these files the import terms can be extracted.

MOCDroid views every application as a document, and each import as a term in that document. In doing so, the generated term-document matrices can be used as input to the Text Mining Package in R, where the output is a number of clusters.

This clustering is done with the *k*-means algorithm. The authors employed Model-Based and Partition Around Metoids PAM clustering, but found the results inferior to *k*-means. They also employ a sparse parameter to remove isolated terms that contribute only noise to the clustering process. This parameter and the cluster size are varied in order to determine the best combination.

The resulting malware and benignware term sets are then used as inputs to a genetic algorithm. This algorithm aims to create malicious and benign models. MOCDroid employs these models as comparisons to make predictions about new data samples. A new sample is judged to be a member of whichever model it shares the most clusters with.

The encoding of the individuals in the genetic algorithm is a binary vector, divided into malicious and benign segments, where a set bit corresponds with the presence of a cluster of import terms. The classifier is trained by modifying the presence of clusters in the model. The genetic algorithm has multiple objectives: maximize accuracy, and minimize false-positives. The authors elected to prioritize accuracy in case of conflict.

3.1.2 ArmsRace. ArmsRace hypothesizes that a simulated competition between malware and detectors is an effective way to evolve sophisticated malware detection systems [2]. In addition, the process yields a shareable data set for other researchers in the field.

The detectors are built in an evolutionary computation framework employing linear genetic programming, a form of supervised learning. The detection programs are represented as a linear sequence of instructions for a virtual programmable machine. These instructions allow the programs to do such basic actions as read from input, read / write to memory, and perform simple mathematical operations. Since there are many individuals in any given generation, a gradient of feedback is achieved. In this way, more nuanced feedback is available to guide selection, as opposed to a simple binary result for a single detector.

To benchmark the detectors outside of the co-evolution system, linear genetic programming and a C5.0 decision tree were also implemented. These models are then used for feature selection. They began by examining all possible Android permissions, before faceting to the most relevant 15. In addition, 8 code features are also considered by the detectors. These features are representative of the structure and use of code in the malicious applications.

For the malware generator, a framework inspired by Mystique was implemented [11]. In this generator, a malware template is built by selecting aggressive or evasive features to achieve multiple objectives. Aggressiveness is to be maximized, while evasion and detectability is minimized. These objectives are satisfied through an evolutionary process facilitated by the Indicator-Based Evolutionary Algorithm (IBEA). The researchers designed ArmsRace to focus on privacy leakage malware.

After constructing both components, the authors combined them into a co-evolution system, where a feedback loop is formed between the detectors and malware. As these components are direct adversaries, they cannot be optimal simultaneously. This dynamic is similar to the conditions that exist in the real-world between malware and anti-malware authors.

3.1.3 Assemblyline. Assemblyline is an open-source tool developed by the Canadian Centre for Cyber Security to detect and analyze malware [13]. It is designed for flexibility, being able to operate as a single node application or scale to a large cluster. Assemblyline is also highly extendable, through its customizable services and API, which can be accessed through the provided Python library or any HTTP-capable client. As Assemblyline is developed by the "Government of Canada's centre of excellence in cyber security" [13], and used by the cyber security community in Canada, we considered it to be a state-of-the-art system.

Akin to its name, Assemblyline operates as a modular framework for analysis, where many services can operate on files they are specifically designed to analyze. The system ships with over 30 services, and users are encouraged to build and deploy their own services. Assemblyline raises alerts if the ingested file is determined to be malicious. Files are ranked with an integer, according to the severity of the malicious features. Files scored less than -1000 are certain to be benign, and files scored greater than 1000 are certain to be malicious, with lesser degrees of certainty in between. This ranking system allows for a triaged approach to malware analysis, drawing attention to more dangerous files. The threshold for determining a file to be likely malicious and raising an alert is set to 500 by default.

The system also provides a web interface that displays pertinent analysis information to the user, as well as aggregable tags. To avoid unnecessary work, Assemblyline fingerprints each file it analyzes and skips duplicate files by default.

For Android applications, Assemblyline provides APKaye. This service is capable of disassembling APK files and checking a number of rules to determine maliciousness. Some of the service actions include checks for the presence of scripts and binaries, irregularities in the signing certificate, dangerous permissions in the Android manifest, suspicious SDK targets, and string analysis.

[2] https://github.com/skylot/jadx

For further scrutiny, APKaye can optionally convert the Android `.dex` files into `.jar` files, which are then passed to the Espresso service for Java archive analysis. We elected to focus on APKaye in this research, so this functionality was not employed.

3.2 Datasets

3.2.1 Android Malware Dataset (AMD). AMD is an extensive, labelled dataset produced by the Argus Lab of the University of South Florida [18]. The set consists of 24,553 malicious samples from 2010 to 2016, divided into 135 varieties from 71 families.

Each variety and family is labelled according to the composition of the malware, installation and activation methods, the type of information to be stolen, privileges, command and control (C&C) specifics, evasion techniques, persistence, and monetization approaches.

The authors also document [18] the evolution of Android malware through the time period of the dataset, and indicate the dominant behaviours observed.

3.2.2 CICAndMal2017 (UNB). This dataset is a collection of 426 malicious and 1,700 benign applications collected from 2015 to 2017 by researchers at the University of New Brunswick (UNB) [7]. The malicious samples are split into four categories (Adware, Ransomware, Scareware, SMS Malware) and 42 families.

In addition to providing the APK files, the authors also ran each malicious sample on real Android smartphones and captured network traffic during installation, before restart, and after restart. From this, more than 80 features are available in the form of CSV files.

3.2.3 Drebin. The Drebin dataset contains 5,560 malicious APKs representing 179 malware families [1] [16]. These files were gathered from 2010 to 2012 by researchers at the Technische Universität Braunschweig.

The malware family labels are available, in addition to features extracted from the Android manifest and disassembled code. Such features include permissions, hardware / app component requests, intents, restricted / suspicious API calls, and network addresses.

3.2.4 F-Droid. F-Droid is an online app store for Android which focuses on the distribution of free and open-source software [4].

For the evaluation of the ArmsRace system [2], the researchers collected over 600 applications, with multiple versions per application in some cases, for a total benignware count of 1,339. We employed this dataset in our research.

3.2.5 Genome. The Android Malware Genome Project was the first large dataset of malicious Android samples released to the research community [20]. Forty-nine different malware families are represented in 1,260 samples, which were collected from 2010 to 2011 by researchers at North Carolina State University.

In their paper, the researchers identify many qualities of the malware, such as the installation and activation methods, and their malicious payloads. This dataset was utilized by ArmsRace in their evaluations [2], so we employed it in our research.

3.2.6 Google Play. The Google Play Store is the official app store for the Android platform, with a variety of free and paid applications available for download [6].

System	Training Datasets	# of Malware Apps	# of Benign Apps
MOCDroid	Genome / F-Droid	300	300
MOCDroid	Drebin / F-Droid	300	300
MOCDroid	Drebin / Google Play	300	300
MOCDroid	Genome / Google Play	300	300
ArmsRace	Genome / F-Droid	420	420
ArmsRace	Drebin / F-Droid	420	420
ArmsRace	Drebin / Google Play	420	420
ArmsRace	Genome / Google Play	420	420

Table 1: Evolutionary Detectors Training Datasets

The ArmsRace system collected a dataset for evaluations from the Google Play Store [2], so we also employed a sample of 1,085 apps in our research.

3.2.7 VirusShare. VirusShare is an online repository of malicious files crowdsourced by members of the security community [14]. As of writing this paper, more than 33,000,000 samples for a variety of platforms are available for download.

Given that the project is not commercially or academically supported, the collection is distributed in large chunks via torrents. For the purposes of our experiments, we acquired 35,397 APKs from two torrents, dated 2013 and 2014.

In the following evaluations, we use Drebin, F-Droid, Genome, and Google Play as datasets for training and testing the evolutionary detectors. AMD, UNB, and VirusShare are then engaged as large, unknown datasets for evaluating the performance of the detectors. Since these unknown datasets were compiled and released after the training datasets, we use them to determine how well the models generalize to 'future' adversaries. For the purposes of comparison, we also evaluate every dataset with APKaye in Assemblyline.

4 EVALUATIONS

To effectively evaluate the chosen detectors, they were each trained and tested on a variety of malicious and benign samples. By employing datasets which differ from the original evaluations of the aforementioned systems, we can evaluate how their performance is affected by the chosen dataset.

After training, the best results were selected from each malicious and benign combination for MOCDroid and ArmsRace. Then, each detector was evaluated against the AMD, UNB, and VirusShare datasets. These datasets are newer, and much larger than the training and testing sets, and should therefore be illustrative of the broader potential of each detector. Every dataset is also assessed with Assemblyline. These results are presented in Subsection 4.4, and the process is graphically depicted in Figure 1. Training dataset combinations and the number of malware / benign apps are given in Table 1.

It is important to note that since each detector uses a different decompiler, each fails on some samples in each dataset. This is why the sets are different sizes for each detector, and differ from the total samples acquired. The exception here is VirusShare, which had a large number of corrupted files (approximately 10,000), but still contains over 20,000 valid samples. These corrupted files are unable to be successfully decompiled by any of the detection systems.

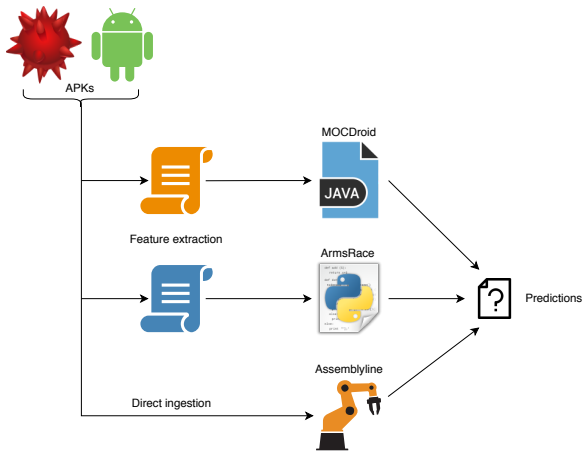

Figure 1: The experiment process.

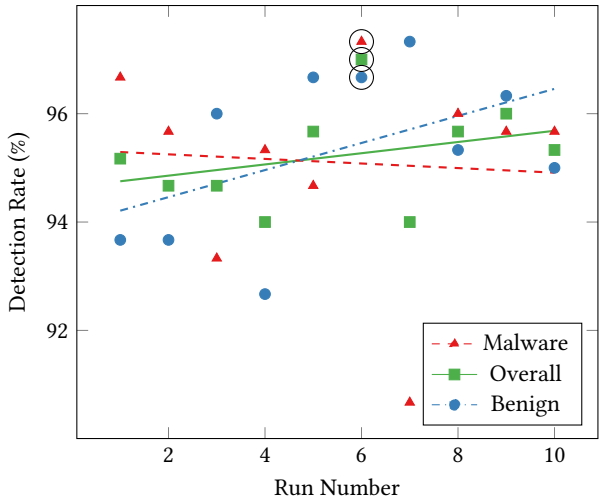

Figure 2: MOCDroid Detection Rates over 10 runs.

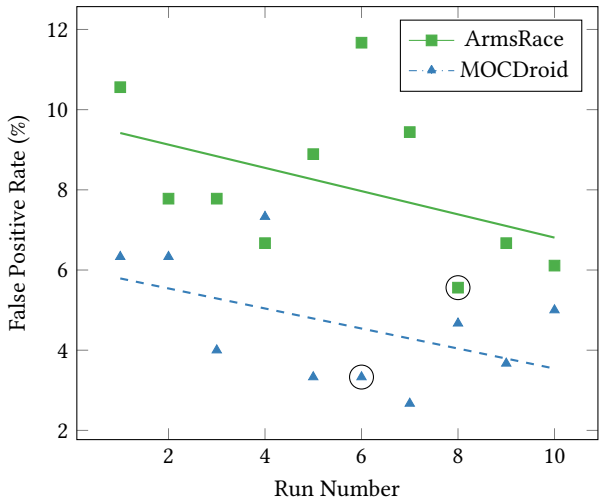

Figure 3: ArmsRace / MOCDroid False Positive Rates over 10 runs.

These failures, on the whole, are relatively infrequent and do not compromise the results.

4.1 MOCDroid

These evaluations of MOCDroid endeavoured to be as close to research in [10] as possible. To achieve this, we used the same parameters / parameter combinations as given in [10]. In [10], MOCDroid is trained and tested on samples from the Aptoide App Store and VirusShare, whereas we trained and tested on Drebin, F-Droid, Genome, and Google Play.

Table 2 shows the results on unseen test data of the different dataset combinations where each test dataset has 300 malware and 300 benign apps. The best results are bolded in this table, and others. These results are consistent with what was reported in [10]. Accuracy sits at approximately 95% and the false positive ratio is less than 5% across many configurations.

Note though, that the false-positives are higher for Google Play / Genome (averaging 7.42%) and much higher for Google Play / Drebin (averaging 9.92%). In addition, clustering Google Play apps failed in R[3] when the sparse parameter was set to 0.99, because of a lack of memory. These results seem to suggest that the Google Play apps are more complicated and use riskier features than F-Droid apps.

There is not a clear winner for sparse / cluster size combination, but this is actually a positive outcome. That is, all pairs are reasonably effective, and any would be a decent choice for future assessments.

To account for the stochastic nature of genetic algorithms, we chose the most successful overall configuration (Genome / F-Droid (0.95, 180)), and ran it an additional 10 times. Figures 2 and 3 depict the detection rates and false positive rates, respectively. The best results are circled in these figures, and the differences between runs are not statistically significant. For clarity, linear regression models for each detection and false positive rate are also presented in these Figures as dashed and solid lines.

[3] https://www.r-project.org/

4.2 ArmsRace

The result for ArmsRace in Table 3 follows a similar narrative to MOCDroid. In [2], ArmsRace is primarily trained and tested on samples from Drebin and F-Droid, whereas we trained and tested on Drebin, F-Droid, Genome, and Google Play. Overall accuracy is slightly up, relative to the original research [2], averaging 93.82% over all configurations. The notable exception is the model trained on Google Play / Drebin, which had a more difficult time identifying benign applications (83.89%, compared to greater than 90% for all other configurations).

As with MOCDroid, false-positives rates are higher for the models trained on Google Play applications. The converse of that is a nearly perfect (98.89%) malware detection rate when using Google Play for benign applications. This reinforces the assumption that

Malware	Benign	Sparse/Cluster	Malware Detection	Benign Detection	Total Accuracy	FP Rate
Genome	**F-Droid**	**0.95/180**	**289/300 (96.33)**	**289/300 (96.33)**	**578/600 (96.33)**	**3.67**
Genome	F-Droid	0.96/120	275/300 (91.67)	287/300 (95.67)	562/600 (93.67)	4.33
Genome	F-Droid	0.97/140	284/300 (94.67)	287/300 (95.67)	571/600 (95.17)	4.33
Genome	F-Droid	0.98/180	278/300 (92.67)	294/300 (98.00)	572/600 (95.33)	2
Genome	F-Droid	0.99/120	279/300 (93.00)	289/300 (96.33)	568/600 (94.67)	3.67
Drebin	F-Droid	0.95/180	281/300 (93.67)	290/300 (96.67)	571/600 (95.17)	3.33
Drebin	**F-Droid**	**0.96/120**	**288/300 (96.00)**	**282/300 (94.00)**	**570/600 (95.00)**	**6**
Drebin	F-Droid	0.97/140	279/300 (93.00)	289/300 (96.33)	568/600 (94.67)	3.67
Drebin	F-Droid	0.98/180	286/300 (95.33)	285/300 (95.00)	571/600 (95.17)	5
Drebin	F-Droid	0.99/120	283/300 (94.33)	290/300 (96.67)	573/600 (95.50)	3.33
Drebin	Google Play	0.95/180	287/300 (95.67)	260/300 (86.67)	547/600 (91.17)	13.33
Drebin	**Google Play**	**0.96/120**	**279/300 (93.00)**	**281/300 (93.67)**	**560/600 (93.33)**	**6.33**
Drebin	Google Play	0.97/140	274/300 (91.33)	270/300 (90.00)	544/600 (90.67)	10
Drebin	Google Play	0.98/180	278/300 (92.67)	270/300 (90.00)	548/600 (91.33)	10
Genome	Google Play	0.95/180	290/300 (96.67)	274/300 (91.33)	564/600 (94.00)	8.67
Genome	Google Play	0.96/120	281/300 (93.67)	273/300 (91.00)	554/600 (92.33)	9
Genome	**Google Play**	**0.97/140**	**289/300 (96.33)**	**284/300 (94.67)**	**573/600 (95.50)**	**5.33**
Genome	Google Play	0.98/180	290/300 (96.67)	280/300 (93.33)	570/600 (95.00)	6.67

Table 2: MOCDroid Testing Results

Malware	Benign	Malware Detection	Benign Detection	Total Accuracy	FP Rate
Drebin	F-Droid	168/180 (93.33)	173/180 (96.11)	341/360 (94.72)	3.89
Genome	F-Droid	167/180 (92.78)	172/180 (95.56)	339/360 (94.17)	4.44
Drebin	Google Play	178/180 (98.89)	151/180 (83.89)	329/360 (91.39)	16.11
Genome	**Google Play**	**178/180 (98.89)**	**164/180 (91.11)**	**342/360 (95.00)**	**8.89**

Table 3: ArmsRace Testing Results

Google Play apps are probably using riskier features than F-Droid apps, hence the higher detection rates.

Finally, the most successful overall configuration (Genome / Google Play) ran an additional 10 times, with Figures 4 and 3 showing the detection rates and false positive rates, respectively. As above, the best results are circled in these figures, linear regression models are calculated and shown as dashed / solid lines, and the differences between runs are not statistically significant.

4.3 Assemblyline

As shown in Table 4, Assemblyline does a good job on the older, well established Genome and Drebin datasets, averaging 88.03%. The benignware detection results are not very effective, averaging 51.63% across the F-Droid and Google Play datasets. The nature of the system, however, is to analyze files already suspected to be malicious, so this less permissive approach is likely to be an intended side-effect. That is, recognizing malicious files is much more important than correctly identifying benign files.

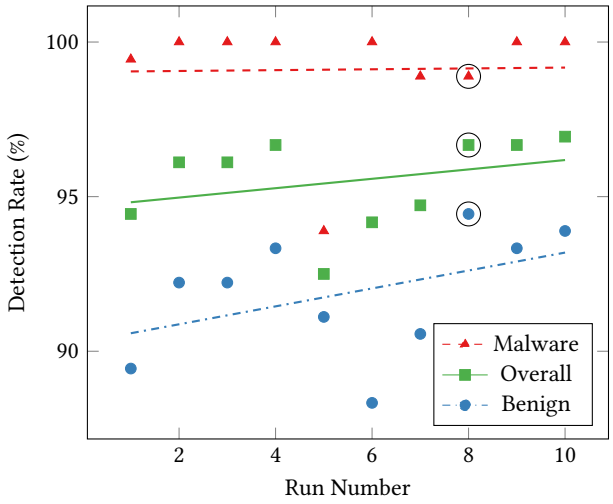

Figure 4: ArmsRace Detection Rates over 10 runs.

Dataset	Is Malicious?	Detection Rate
AMD	T	16850/23618 (71.34)
Drebin	T	4649/5514 (84.31)
Genome	**T**	**1146/1249 (91.75)**
UNB-Malware	T	315/424 (74.29)
VirusShare	T	16095/20884 (77.07)
F-Droid	F	767/1337 (57.37)
Google Play	F	490/1068 (45.88)
UNB-Benign	F	1154/1688 (68.36)

Table 4: Assemblyline Results on all datasets employed

Testing Dataset	Training Datasets	Detection Rate
AMD	Genome / F-Droid	15104/24553 (61.52)
AMD	**Drebin / F-Droid**	**21518/24553 (87.64)**
AMD	Drebin / Google Play	21295/24553 (86.73)
AMD	Genome / Google Play	16187/24553 (65.93)
UNB Ben	**Genome / F-Droid**	**1675/1700 (98.53)**
UNB Ben	Drebin / F-Droid	1608/1700 (94.59)
UNB Ben	Drebin / Google Play	1584/1700 (93.18)
UNB Ben	Genome / Google Play	1653/1700 (97.24)
UNB Mal	Genome / F-Droid	153/426 (35.92)
UNB Mal	Drebin / F-Droid	275/426 (64.55)
UNB Mal	**Drebin / Google Play**	**294/426 (69.01)**
UNB Mal	Genome / Google Play	221/426 (51.88)
VirusShare	Genome / F-Droid	15193/20984 (72.40)
VirusShare	**Drebin / F-Droid**	**19775/20984 (94.24)**
VirusShare	Drebin / Google Play	19122/20984 (91.17)
VirusShare	Genome / Google Play	15147/20984 (72.18)

Table 5: MOCDroid Results on Unknown datasets

4.4 Unknown Datasets

As discussed earlier, for these evaluations, we employed the AMD, UNB and VirusShare datasets. Given that no parts of these datasets were used in training, we called them unknown datasets (from the perspective of the training model).

4.4.1 MOCDroid. The best detectors based on different training datasets for each malware / benignware combination were selected as follows: Genome / F-Droid (0.95, 180), Drebin / F-Droid (0.96, 120), Drebin / Google Play (0.96, 120), and Genome / Google Play (0.97, 140), where the first number in parentheses represents the sparse parameter and the second number represents the cluster size.

MOCDroid performs well on the AMD and VirusShare sets in certain configurations, as can be seen in Table 5. In particular, the Drebin / Google Play model detects 87.64% and 94.24% of the the malicious samples, for AMD and VirusShare respectively. For both datasets, using Drebin as the malicious training samples lead to an average detection rate increase of almost 22%.

In addition, the UNB benignware are very accurately detected, ranging from 93.18% with the Drebin / Google Play model to 98.53% with the Genome / F-Droid model, averaging at nearly 96%.

Every model breaks down on the malicious UNB samples, however. The best model, Drebin / Google Play, correctly flags 69.01% of the samples, while the worst, Genome / F-Droid, only achieves a 35.92% detection rate.

Testing Dataset	Training Datasets	Detection Rate
AMD	Genome / F-Droid	19279/24449 (78.85)
AMD	Drebin / F-Droid	20371/24450 (83.32)
AMD	**Drebin / Google Play**	**24254/24449 (99.20)**
AMD	Genome / Google Play	19020/24449 (77.79)
UNB Ben	Genome / F-Droid	1511/1700 (88.88)
UNB Ben	**Drebin / F-Droid**	**1642/1700 (96.59)**
UNB Ben	Drebin / Google Play	1249/1700 (73.47)
UNB Ben	Genome / Google Play	1528/1700 (89.88)
UNB Mal	Genome / F-Droid	269/425 (63.29)
UNB Mal	Drebin / F-Droid	296/425 (69.65)
UNB Mal	**Drebin / Google Play**	**381/425 (89.65)**
UNB Mal	Genome / Google Play	324/425 (76.24)
VirusShare	Genome / F-Droid	17779/20972 (84.77)
VirusShare	Drebin / F-Droid	18931/20972 (90.27)
VirusShare	**Drebin / Google Play**	**20757/20972 (98.97)**
VirusShare	Genome / Google Play	17680/20972 (84.30)

Table 6: ArmsRace Results on Unknown datasets

4.4.2 ArmsRace. As depicted in Table 6, this system performs excellently on the AMD and VirusShare datasets. Similar to MOCDroid, the Drebin / Google Play model is the most successful one, achieving detection rates of 99.20% and 98.97%, respectively. Even the worst configurations correctly detect 77.79% and 84.30% of the samples, respectively. Using Drebin for training corresponds to an average increase of almost 12% in detection rate. This increase is not as dramatic as it is for MOCDroid, but is still noticeable.

ArmsRace scores slightly lower in accuracy on the UNB benignware, relative to MOCDroid. The average score is 87.21%, with three of the four models scoring below 90%.

The UNB malware, on the other hand, are detected much more accurately than with MOCDroid. Three of the four ArmsRace models score above the best MOCDroid model, with the model trained on Drebin / Google Play achieving the best score at nearly 90%.

4.4.3 Assemblyline. Referring again to Table 4, we can see that Assemblyline lags behind the best models for the evolutionary systems on most of the unknown datasets.

In particular, Assemblyline only manages a better accuracy than the best MOCDroid model when classifying the UNB Malicious dataset. Overall, the unknown datasets lead to an average of almost 15% lower detection rate in Assemblyline than MOCDroid.

This disparity is even more pronounced for ArmsRace. As Assemblyline scores lower on every unknown dataset, the overall average is 23% lower.

These scores are especially affected by Assemblyline's aforementioned poor performance on benignware, but even when looking at only the malicious unknown datasets, there is still an average score drop of 9% and 22% compared to MOCDroid and ArmsRace, respectively.

5 CONCLUSION AND FUTURE WORK

After reviewing these results, we can draw conclusions in regard to the effectiveness of evolutionary, co-evolutionary, and rule-based detection systems on recent Android malware, as well as the datasets that should be used for training future detection systems.

MOCDroid, an evolutionary system, is very effective on older malicious datasets such as Genome and Drebin, and benign datasets sourced from F-Droid and Google Play. In training, these detectors routinely score above 90%. These results are also reflected, to a lesser extent, in the AMD and VirusShare datasets. When more recent malicious samples are introduced, however, MOCDroid begins to break down, failing to achieve a score greater than 70%.

ArmsRace, a co-evolutionary system, is similarly effective on the aforementioned malicious and benign sets, but also manages a respectable 89% when evaluating the newer UNB Malicious samples. This is possibly because of the focus ArmsRace places on privacy leakage malware, which is not present in the UNB samples.

To summarize, these results seem to support the hypothesis in Ucci et al.'s recent survey [17] that evolutionary / co-evolved solutions may be well suited for detecting new variants of malware, because they are more able to adapt to new malware. Furthermore, our results show that both the ArmsRace and MOCDroid systems are competitive with the state-of-the-art, rule-based system, Assemblyline.

Looking at the influence of the training and testing datasets, Genome is beginning to show its age. The MOCDroid and ArmsRace detectors trained on Drebin score consistently higher than their Genome counterparts. This is not entirely unexpected, as malware evasion practices have evolved significantly since Genome was released [19].

Prior to these evaluations, VirusShare was considered to be a more ambiguous set, as the anonymous crowd-sourcing nature of the repository makes sample attribution impossible. AMD, on the other hand, is a relatively well known academic data set. The assumption was that this would be reflected in the detection rates of malware and benign apps, where VirusShare would be harder to identify. Tables 5 and 6 suggest otherwise. VirusShare seems to be a more predictable set than AMD. This might suggest overlap occurring between the training / testing datasets and that malware and benign apps behaviours in the training datasets seems to be similar to the ones in the VirusShare dataset. Further analysis is necessary to understand these behaviours.

Moreover, we would like to dig deeper into other malware families and evaluate rule-based, non-evolutionary learning, evolutionary and co-evolutionary detector models on such datasets for future work. Increasing the variety of training datasets may lead to increased stability in the machine learning models. Also, we would like to study the detection approaches used in this paper against deep learning based detection systems and explore hybrid approaches for adapting to new variants of malware.

6 ACKNOWLEDGEMENT

This research is supported by the Natural Science and Engineering Research Council of Canada (NSERC). This research is conducted as part of the Dalhousie NIMS Lab at: https://projects.cs.dal.ca/projectx/.

REFERENCES

[1] Daniel Arp, Michael Spreitzenbarth, Malte Huebner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *21st Annual Network and Distributed System Security Symposium (NDSS)*.

[2] R. Bronfman-Nadas, N. Zincir-Heywood, and J. T. Jacobs. 2018. An Artificial Arms Race: Could it Improve Mobile Malware Detectors?. In *Network Traffic Measurement and Analysis (TMA) Conference*.

[3] E. David and N. Netanyahu. 2015. DeepSign: Deep Learning for Automatic Malware Signature Generation and Classification. In *International Joint Conference on Neural Networks (IJCNN)*. Killarney, Ireland, 1–8.

[4] F-Droid Limited. [n. d.]. F-Droid. https://f-droid.org/

[5] Amy Ann Forni and Rob van der Meulen. 2017. Gartner Says Worldwide Sales of Smartphones Grew 9 Percent in First Quarter of 2017. https://www.gartner.com/en/newsroom/press-releases/2017-05-23-gartner-says-worldwide-sales-of-smartphones-grew-9-percent-in-first-quarter-of-2017

[6] Google LLC. [n. d.]. Google Play. https://play.google.com/store

[7] Arash Habibi Lashkari, Andi Fitriah A.Kadir, Laya Taheri, and Ali A. Ghorbani. 2018. Toward Developing a Systematic Approach to Generate Benchmark Android Malware Datasets and Classification. In *52nd IEEE International Carnahan Conference on Security Technology (ICCST)*.

[8] Liu Liu, Bao-sheng Wang, Bo Yu, and Qiu-xi Zhong. 2017. Automatic malware classification and new malware detection using machine learning. *Frontiers of Information Technology & Electronic Engineering* 18, 9 (01 Sep 2017), 1336–1347. https://doi.org/10.1631/FITEE.1601325

[9] A. Martín, H. Menéndez, and D. Camacho. 2016. MOCDroid GitHub repository. https://github.com/alexMyG/MOCDroid

[10] A. Martín, H. Menéndez, and D. Camacho. 2017. MOCDroid: Multi-objective evolutionary classifier for Android malware detection. *Soft Computing* 21, 24 (2017), 7405–7415.

[11] G. Meng, Y. Xue, C. Mahinthan, A. Narayanan, Y. Liu, J. Zhang, and T. Chen. 2016. Mystique: Evolving Android Malware for Auditing Anti-Malware Tools. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16)*. ACM, New York, NY, USA, 365–376. https://doi.org/10.1145/2897845.2897856

[12] N. Milosevic, A. Dehghantanha, and K.-K.R. Choo. 2017. Machine learning aided Android malware classification. *Computers & Electrical Engineering* 61 (July 2017), 266–274. http://eprints.whiterose.ac.uk/128366/ © 2017 Elsevier Ltd. This is an author produced version of a paper subsequently published in Computers & Electrical Engineering. Uploaded in accordance with the publisher's self-archiving policy. Article available under the terms of the CC-BY-NC-ND licence (https://creativecommons.org/licenses/by-nc-nd/4.0/).

[13] Canadian Centre for Cyber Security. [n. d.]. Assemblyline. https://cyber.gc.ca/en/assemblyline

[14] J-Michael Roberts. [n. d.]. VirusShare. https://virusshare.com

[15] S. Sen, E. Aydoğan, and A. I. Aysan. 2018. Coevolution of Mobile Malware and Anti-Malware. *IEEE Transactions on Information Forensics and Security* 13, 10 (Oct. 2018), 2563–2574. https://doi.org/10.1109/TIFS.2018.2824250

[16] Michael Spreitzenbarth, Florian Echtler, Thomas Schreck, Felix C. Freling, and Johannes Hoffmann. 2013. MobileSandbox: Looking Deeper into Android Applications. In *28th International ACM Symposium on Applied Computing (SAC)*.

[17] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. 2019. Survey of machine learning techniques for malware analysis. *Computers & Security* 81 (2019), 123–147. https://doi.org/10.1016/j.cose.2018.11.001

[18] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep Ground Truth Analysis of Current Android Malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*. Springer, Bonn, Germany, 252–276.

[19] Y. Xue, G. Meng, Y. Liu, T. H. Tan, H. Chen, J. Sun, and J. Zhang. 2017. Auditing Anti-Malware Tools by Evolving Android Malware and Dynamic Loading Technique. *IEEE Transactions on Information Forensics and Security* 12, 7 (July 2017), 1529–1544.

[20] Y. Zhou and X. Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *2012 IEEE Symposium on Security and Privacy*. 95–109. https://doi.org/10.1109/SP.2012.16