An Artificial Arms Race: Could it Improve Mobile Malware Detectors?

Rapahel Bronfman-Nadas
Computer Science
Dalhousie University
Halifax, Canada
raphael@dal.ca

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

John T. Jacobs
Raytheon Space and Airborne Systems
California, USA
john_t_jacobs@raytheon.com

Abstract—**On the Internet today, mobile malware is one of the most common attack methods. These attacks are usually established via malicious mobile apps. To combat this threat, one technique used is the deployment of mobile malware detectors. As the mobile threats evolve, designing and developing mobile malware detectors remains a challenging task. In this paper, we aim to explore whether creating an artificial arms race between mobile malware and detectors could improve the ability of the detector to adapt to the evolving threats. To better model this interaction, we present a co-evolution of both sides of the arms race using genetic algorithms. The experimental evaluations on publicly available malicious and non-malicious mobile apps and their variants generated by the artificial arms race show that this approach improves the detectors understanding of the problem.**

Index Terms—**Malware detection, Android, Artificial Arms race, Co-evolution, Genetic Algorithms**

I. Introduction

Malware detection remains a challenging task as malware is created specifically to avoid alerting malware detectors. Likewise, detectors are designed to overcome this malicious opponent and retaliate by improving the methods with which they can detect malware. From this vantage point, there exists an arms race between those attempting to manufacture malware and those attempting to detect it. As one's performance improves, the other's is reduced. Malware strength is partly based on how detectable it is. Detectors performance is entirely based on how well it correctly classifies malware and benign files.

Thus, in this research, our primary goal is to explore whether creating a simulated competition between mobile malware and detectors could improve the ability of the detector to adapt to the evolving threats / malware. Furthermore, such a competitive training process also provides us with datasets that could be shared and used among the researchers working in this area. To achieve this, we aim to create the artificial arms race framework by using a bio-inspired malware generator and a bio-inspired malware detector under a co-evolutionary paradigm. In this framework, the malware detection module could function in conjunction with the malware generation module, or it could be an independent module. To better simulate this direct adversarial intent of malware, this paper presents a method of training a malware detector using a malware generator as a co-evolved genetic algorithm. The methodology used in this work defines two populations with competitive goals. Each is individually striving towards its own goal and will be awarded based on how well it performs. Given that the parts of the populations that survive are those that outperformed or were closest to outperforming the other population, this grows the knowledge of the population beyond the static sets used in other algorithms. The first population consists of a genetic program deployed to detect malware using feature-based analysis. This serves as the detector which can be evolved to classify as either malware or benign. The second being a population of malware which attempts to become as strong a sample as possible, given that it must remain simultaneously undetectable. This is the population representing the changing malware arms race. The malware begins simple and is rewarded for both achieving greater levels of control of the target system, and by being misclassified by the detector population. The premise is based on the logic that the generated malware is used to show what the detectors are lacking and inform what future changes are required. Simultaneously, the malware will learn from the results of the the detectors what kind of malware is classified incorrectly, providing a dataset comprised of new malware that is more difficult to classify.

With this approach, not only we aim to improve the detectors understanding of the problem, but also provide meaningful datasets to represent real life situations for further analysis and measurements. Our results show that the signatures / solutions obtained for malware detectors are simpler, and require less data from each malware sample to be detected / classified accurately. The rest of the paper is organized as follows: Literature review and background work are summarized in section II. The methodology used to generate malware and the detector side of the artificial arms race as well as co-evolve them are introduced in section III. While experimental set up is given in section IV, the results are presented in section V. Finally, conclusions are drawn and the future work is discussed in section VI.

II. Literature review

In this section, we summarize the background works in malware detection on mobile systems by considering (A) the detector, (B) the artificial arms race between the detector and malware, and (C) the Android mobile system as the platform.

A. Detector

The mobile malware detector needs to be able to classify malware and benign software in some meaningful way. The state of mobile malware, such as malware on the Android platform, is changing as new malware is developed creating a moving target for malware detectors. The performance of feature based detectors is better when new malware is able to be processed. By building a more scalable deeper understanding of the apps process, the application itself is better able to predict future trends [1]. Previous work on using machine learning has produced results showing that, indeed, malware may be classified through machine learning approaches. For a Bayesian classifier, it was found that 15 to 20 features were optimal for increased performance [2]. The mobile malware detection space has expanded research on using external app resources to help detect malware. Some detectors have been built to use Battery Life and App Permission for heuristically predicting what may be malware. Skovoroda et. al. discussed that external values such as these could be combined with both static and dynamic analysis, and machine learning to create more robust detectors [3]. Permission based detection with Android malware has proven to be very successful [4]. A large amount of malware can be classified using only permissions. This method functions because the permissions decide what actions the app is able to perform. Permission evaluation also becomes an issue as apps may request more permissions then they actually need [5]. Even if an app is not malicious, requesting additional permissions is considered a security risk.

B. Malware

Malware, in the mobile space, is focused on apps. These apps are often from different malware families with their own goals and malicious intents. Due to the structure of Android systems, most malware are modified versions of existing apps that contain malicious functions. Many apps of the same family have similar feature combinations as some are required for the exploits or actions they intend to use [6]. Earlier research in this field include vulnerability analysis tools which aim to evolve new variants of known malicious behaviour such as buffer flow attacks [7], [8], where Kayacik et. al. evolved using genetic programming against open source anomaly detectors (Stide, Ph) [9] and intrusion detection systems (Snort) [10]. Fraser et al. employed a similar approach to demonstrate a proof of concept for evolving ROP-chain payloads [11]. Noreen et al. and Meng et al. researched similar approaches for evolving mobile malware in [12] and [13], respectively. However, all the aforementioned works depend on stand alone malware detectors, where these detectors do not change over the course of the evolutionary system. The detectors can take many forms. For example, some pre-trained machine learning methods were used in Mystique [13] or non-machine learning systems such as [9]. In summary, the above previous works demonstrate that by using existing malware, a form of abstract representations and an evolutionary algorithm, new malware can be evolved beyond the capacity of the static detectors. Evolved malware is given the opportunity to find the flaws in the detectors. In doing so, the researchers aim to improve the detectors before such malware is introduced into the intended marketplace, such as Android.

C. Android

Android has a key place in the mobile market that made it ideal for this experiment. Firstly, Android is the most widely used mobile operating system since 2011 [3]. Secondly, Android's app system requires most external actions to be noted statically in a manifest file, categorized into small groups of actions called permissions. These permissions alone are an effective way to determine what exploits and potential harm an app could accomplish [14]. Finally, there has been direct research in the automatic evolution of Android malware [13]. Thus, in our research we also make use of the Android platform and available apps to be able to evaluate our proposed system and present it in conjunction to the previous work in this area.

III. Methodology

As discussed earlier, the primary goal of this research is to explore whether creating a simulated competition between mobile malware and detectors could improve the ability of the detector to adapt to evolving threats / malware. The following details the components of the proposed framework to achieve this goal.

A. Malware Detector

The detector of the proposed artificial arms race framework is implemented using an evolutionary computation technique with the goal of evolving to correctly classify an Android app as malware or benign. This implementation takes the form of a Linear Genetic Programming (GP). This is a supervised learning algorithm, which is a variant of GP where programs in a population are represented in linear form, as a sequence of instructions from an imperative programming language [15].

This approach was chosen because we aim to co-evolve the detectors along with the malware using an approach similar to that of Mystique [13]. The co-evolution will require a population of varying individuals that can provide a gradient of feedback, instead of a simple binary response. Moreover, the evolved solution must be able to classify and build behaviors based on the given inputs. Given these requirements, GP becomes the natural fit as the learning technique since it has the ability to evolve and generate code automatically. These are the main reasons why GP is chosen to test this methodology.

In linear GP, each instruction executes an operation over the operands, which can be registers, constants, or input value. Then, the result of each instruction is stored in a register. The final result of the program is taken as the values of the registers, which are designated as the output registers at the end of the program. There are two properties that differentiate Linear GP from other representations of GP. First, the imperative representation allows the data to be processed as in a directed graph, thus facilitating reuse of register content by multiple instructions. This in turn allows the reuse of subprograms for evolving compact solutions. Second, structurally noneffective code (introns) - instructions that have no impact on the output registers - support neutral variation and skipping of intron code during fitness evaluation, where noneffective code can be tuned effective by variation operators. In this case, a population of variable length strings represent byte-code of a virtual programmable machine. The byte-code is designed to perform the following actions:

- Read from fixed number of input locations
- Read and write to a fixed number of working memory locations
- Perform simple mathematical functions

In this research, we implemented a linear GP based detector within the artificial arms race framework to be able to co-evolve the attack and detector side simultaneously. However, we also implemented a linear GP and a decision tree (C5.0) based malware detector outside of the artificial arms race framework. In doing so, we aim to evaluate the performance of the framework against standalone machine learning based detectors. To this end, we choose C5.0 to represent the state of the art supervised learning algorithms that are not based on evolutionary computation as GP. C5.0 is a more effective and efficient decision tree algorithm than C4.5 [16]. It is designed to maximize interpretability, and takes the form of if-then rules, which are generally easy to understand for the human expert.

All the malware detectors discussed above analyze mobile apps to identify whether they are malicious or not. In this work, the detector analyzes the features of an app that are extracted from an .apk file. APK stands for Android Package Kit. This is the package file format used by the Android operating system for distribution and installation of mobile apps. There are multiple feature sets that could be used to analyze a mobile app. In general, these include in some shape and form: (i) Permissions, and (ii) Code attributes.

Android permissions that an app uses are declared in a metadata file known as the Android manifest. Permissions can enable hardware access, operating system features, or access to other apps. There are over 100 officially supported permissions. Either the full list of official permissions or a subset can be employed. In the first phase of this research, we employed the full list of official permissions on 600 of the apps we had, 300 malware and 300 benign, and evaluated two classifiers on these permissions to test which classifier would identify a malicious apps more accurately. To this end, we use the C5.0 decision tree classifier as a representative of the state of the art classifiers and the Linear GP classifier, which we aim to use in our artificial arms race framework. The results of these tests are displayed in Tables I and II. These results show that C5.0 classifier reaches up to 91% using all permissions whereas the Linear GP classifier reaches in average 76% accuracy using all the permissions. To improve this, we focus on a smaller set of permission list features based on previous research [2], [4] and further empirical evaluations. At the end of these evaluations, we choose the most relevant 15 permissions that are likely to be used for malware detection. These 15 permissions are listed in Table III, and results of the evaluations are shown in Tables VI and VII. Given both classifiers are very comparable in their accuracy, we conclude that these 15 permissions are more consistent representing the normal and benign behaviours of the apps.

TABLE I
RESULTS OF GP WITH 149 PERMISSIONS

		True class		
		Malware	Benign	Total
Predicted class	Malware	167	12	93%
	Benign	133	288	68%
	Total	56%	96%	76%

TABLE II
RESULTS OF C5.0 WITH 149 PERMISSIONS

		True class		
		Malware	Benign	Total
Predicted class	Malware	272	28	91%
	Benign	27	273	91%
	Total	91%	91%	91%

On the other hand, Android code features are associated with the creation of the code of an app. In most cases, these are frequencies of different code components. These include (but are not limited to): the number of classes, the number of interfaces, and the number of instanced variables. In the literature, these features are considered to be representative of the structure and the use of the code [17]. The chosen internal features are listed in Table IV

Using the aforementioned Android permission and code based features, the linear GP based detector is trained on a subset of Android app samples to set up the artificial arms race framework. In this case, the method of evaluation for this malware detector is defined by a single goal: performance – the number of correctly classified malware samples.

TABLE III
THE SELECTED 15 PERMISSIONS

INTERNET
READ_SMS
SEND_SMS
READ_CONTACTS
READ_EXTERNAL_STORAGE
WRITE_EXTERNAL_STORAGE
INSTALL_PACKAGES
BIND_DEVICE_ADMIN
BIND_ACCESSIBILITY_SERVICE
RECEIVE_BOOT_COMPLETED
READ_PHONE_STATE
CAMERA
RECORD_AUDIO
READ_CALENDAR
ACCESS_FINE_LOCATION

In should be noted here that the performance value is

TABLE IV
ANDROID CODE FEATURES

Number of Classes
Number of Classes using interfaces
Number of Classes containing annotations
Number of Direct methods
Number of Virtual methods
Number of Abstract methods
Number of Static Member variables
Number of Instanced Member variables

measured evenly, so that the in-balances in the dataset will not affect the overall accuracy.

B. Malware Generator

The goal of the malware generator in the proposed artificial arms race framework is to build a population of malware samples that can evade the malware detector while performing malicious actions. In this case, the method of evaluation for the malware generator is defined by three goals:

- Aggressiveness: The number of potential actions able to be taken. The more actions taken by the malware, the closer it is to complete its goal. Therefore, aggressiveness is a trait to maximize.
- Evasion: The number of non-vital actions required to avoid detection. Using a minimum viable set of evasion tactics is preferable, as it adds to the complexity of construction, helping to reduce detection while not adding other benefits. While evasion does not directly affect the goal, the least actions required to reach a successful malware sample is preferable.
- Detection: This is a measure of how well the malware can go undetected by the detector. In practice, this means blending in with benign software to such a degree that there is no meaningful way of distinguishing it.

The malware generation module used in this work is based on the Mystique method introduced by Meng et. al. [13]. More specifically, a malware template was built to fill the requirements of privacy leaking malware. In this case, the goal of privacy leaking malware is to collect information from the target, and send it to another device. Aggressive actions are considered to be any method of collecting or sending data.

Each malware sample is given the genotypic representation of a list of features to be generated. The features represent either an aggressive or evasive action. Each list is ensured to represent a malicious application because of restrictive rules that enforces the selection of features to complete the goal of the malware. If a feature list cannot be considered as malware, it is rejected [13].

To evolve the population from an initial random state, the evaluation of the three measures are used: aggressiveness, evasion, and the detection ability. Due to this multiple objective nature of evaluating malware, the evolutionary method used is the IBEA algorithm [13]. IBEA allows for multiple objectives to be targeted, and builds a diverse array of solutions. Pareto optimality is used for comparing the malware samples. Given the three scalar values (aggressiveness, evasion, and detection) associated with the performance, this method measures the relative performance in all values, instead of measuring absolute performance at a fixed combination of the three. IBEA builds on dominant elements of each generation, so that each non-dominant sample from each generation is added to a list of bases. Then, each future generation uses these bases to build a modified list of features, either via crossover with another base, or random mutation.

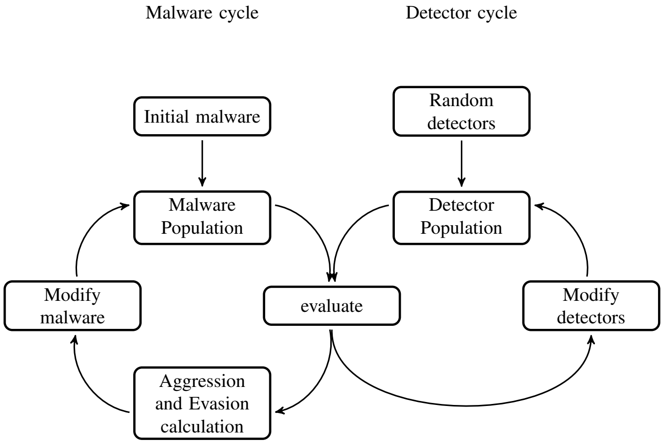

Fig. 1. Evolution diagram

It should be noted here that each aggressive feature is weighted evenly, so that the total number of aggressive features becomes the direct aggressive value for the sample. Similarly, each evasive feature is also weighted evenly, so that the total number of evasive features becomes the direct evasive value of the sample. Finally, the detection measure is computed with the malware detection module of the artificial arms race framework that is used to evaluate the malware for detectability.

C. Competitive Co-evolution

The competitive co-evolution is the technique we employ to implement the artificial arms race between the malware detector and the malware generator. Again, this is a GP based system. The structure of the system takes the form of a feedback loop as seen in Figure 1. In this framework, the natural tension between the goals of the malware generator module and the malware detector module are used to increase the performance of each module.

- The malware detector attempts to classify malware, while avoiding classifying safe software as malware. Naturally, the classifier is only considered to be successful, if it can detect the malware.
- The malware generator creates malware. Malware is only considered to be successful, if it can evade the detection of the malware detector.

Due to the malware specific goal of avoiding classification, the classifier is a direct adversary. Likewise, malware attempting to be unclassifiable becomes an adversary of any detector. Detectability can swing between the malware and the detectors, but both cannot be optimal at once.

Algorithm 1 Co-evolution main loop

```
Input: input_dataset
Output: malware_detector_population
  Initialisation :
  pop_d ← random_population_GP()
  pop_m ← random_population_Malware()
  archive ← []
  LOOP for each generation
  for $i = 0$ to $max\_generations$ do
      malware_dataset = input_dataset ∪ pop_m ∪ archive
      eval ← Evaluate(pop_d, malware_dataset)
      Sort pop_m by eval
      Append first $archive\_amount$ of pop_m to archive
      Append last $archive\_amount$ of pop_m to archive
      pop_d ← GP_Evolve(pop_d, eval_d)
      pop_m ← IBEA_Evolve(pop_m, eval ∩ pop_m)
  end for
  return pop_d
```

Algorithm 1 summarizes the co-evolution process. Starting with random initialization populations, to perform a generation, the malware detector population is evaluated using both a constant dataset and the malware population. In this framework, the detection rate of the malware is the evaluation performed by the detector during the co-evolution process. Aggressiveness and evasion are computed using only the population, as those measures only use the samples genotype. With the population evaluations done, both are evolved using their respective evolutionary procedures.

To better adapt to the changes in the apps / malware, and to ensure that the final result is robust, as the populations change, an archive is used to select features of the malware population to preserve and to use for detection (evaluations) over generations.

IV. Experiment setup

The dataset used in this work for the input malware samples is a random subset, 1000 malware apps, of the DREBIN dataset [6]. Along with this, we also use 1000 benign apps to train a malware detector. Selecting these apps was performed randomly. In total, we have 2000 apps where half of them are malware and half of them are benign. We then use 70% of this dataset for training and 30% for testing. Therefore, there are 1400 samples in the training and 600 samples in the testing datasets. We have employed two machine learning algorithms - namely C5.0 and GP - to generate a malware detector. Thus, we aim to perform the following sets of evaluations in order to explore how far we could push the proposed framework:

1) The C5.0 based detector uses just the DREBIN dataset without the addition of generated malware. This creates a decision tree to classify the apps into malware and benign. The complexity of the generated decision tree is the number of nodes present in the tree. Additionally, because C5.0 algorithm uses information gain, the decision tree is capable of choosing the most important features among all the features given. This enabled us to choose the most relevant 15 permissions from the set of all available permissions (149) that represents an app.
2) The GP based detector uses the DREBIN dataset without the addition of generated malware. In this case, we can also measure the complexity of the classifier using the number of instructions in the chosen solution's program. Again, GP classifier is able to identify the most important features from the set of all features given. This enabled us to choose the most relevant 15 features agreed on by all used classifiers.
3) Finally, the experimental Co-evolved method. The GP based detector, chosen because of its co-evolution capability, is used in the artificial arms race framework. The same measurements of complexity can be directly compared to other GP evolved detectors, giving us a good comparative benchmark between the static and the co-evolved classifiers.

It should also be noted here that C5.0 classifier was used to verify that the selected apps were comparable to the full version of the dataset. We achieved this by splitting the dataset into the intended size of a training set using random apps from the full dataset. Building a tree using this dataset, we measured the accuracy and calculated the number of features most used in the tree. Randomly selecting the number of apps for the training set, and comparing the trees built with the subset and the full dataset enables the verification of the selected apps. It is important to choose the training set to be as small as possible to decrease the training time. However, at the same time, it is important not to loose accuracy for detection. In the end, the selected training dataset has 0.4% difference in accuracy and include the top 10 features which were also present in the tree generated by the full dataset.

V. Results

In the test phase of this work, ten runs are performed for each technique to get an idea of the performance differences. The measures used for these evaluations are: (i) the number of apps (samples) correctly classified, (ii) the minimization of the number of features used, and (iii) the simplification of the solutions generated.

The goal of detecting malware accurately is the only guiding principal in the artificial arms race framework. Features used and the complexity of the solution were selected during the training phase of the classifiers. While there is no bias for simple solutions or minimal features, creating simpler solutions is a bi-product of this method.

Table V shows the results of a set of experiments we conducted to observe the sensitivity of the training data size on the performance of stand alone detectors. In these experiments, we varied the training set size for each category of apps from 300 to 1000, and observe the precision and recall rates to determine the most suitable training dataset size. Based on

```
Legend: 'c' is Clean or Benign class
        'm' is Malware class
 READ_PHONE_STATE = f:
 :.SEND_SMS = t:
 :  :.classes_with_annotation <= 37: m
 :  :   classes_with_annotation > 37: c
 :  SEND_SMS = f:
 :  :.abstract_count > 562: c
 :     abstract_count <= 562:
 :     :.INTERNET = f: c
 :       INTERNET = t:
 :       :.static_count <= 70: m
 :          static_count > 70:
 :          :.classes_with_annotation <= 2: c
 :             classes_with_annotation > 2:
 :             :.classes_with_annotation <= 23: m
 :                classes_with_annotation > 23: c
 READ_PHONE_STATE = t:
 :.classes_with_interface > 558:
    :.ACCESS_FINE_LOCATION = t: m
    :   ACCESS_FINE_LOCATION = f: c
    classes_with_interface <= 558:
    :.BIND_ACCESSIBILITY_SERVICE = t: c
      BIND_ACCESSIBILITY_SERVICE = f:
      :.INTERNET = f:
        :.SEND_SMS = t: m
        :   SEND_SMS = f: c
        INTERNET = t:
        :.READ_CALENDAR = t:
          :.WRITE_EXTERNAL_STORAGE = t: m
          :   WRITE_EXTERNAL_STORAGE = f: c
          READ_CALENDAR = f:
          :.abstract_count <= 51: m
             abstract_count > 51:
             :.direct_count > 418: m
                direct_count <= 418:
                :.classes_with_interface <= 37: m
                   classes_with_interface > 37: c
```

Fig. 2. C5.0 tree using 15 permissions and 8 code features

these results, we chose the training dataset size to be 700 benign and 700 malware apps.

Tables I, II, VI and VII show the performance results of the GP and C5.0 classifiers trained on the training dataset and tested the on the unseen test dataset. The first two tables show the performance of these classifiers when the apps were represented by using all the official permissions list. The latter two tables show the performance of these classifiers when only 15 permissions are used to represent an app. All these results were obtained when these classifiers were used outside of the artificial arms race framework as stand alone (static) malware detectors.

As the next step, we employed the aforementioned eight code features together with the 15 permission features to represent an app using 23 features to the detector. Table VIII shows the performance of the GP based standalone malware detector under this representation. Table IX and Figure 2 show the performance of the C5.0 based detector and the resulting tree rules using 23 features on the same data. By adding the code features to the representation of an app, we get a 5% increase in the performance of GP and 8% increase in the performance of the C5.0. When we analyzed what the detectors learned in these experiments, we observed that GP used only 12 of the 23 features whereas C5.0 used only 11.

TABLE V
PRECISION AND RECALL OF DIFFERENT DATA SIZES

Test	Precision	Recall
GP 300	97%	88%
GP 500	96%	89%
GP 700	97%	89%
GP 1000	96%	89%
C5.0 300	99%	96%
C5.0 500	86%	96%
C5.0 700	96%	97%
C5.0 1000	96%	97%

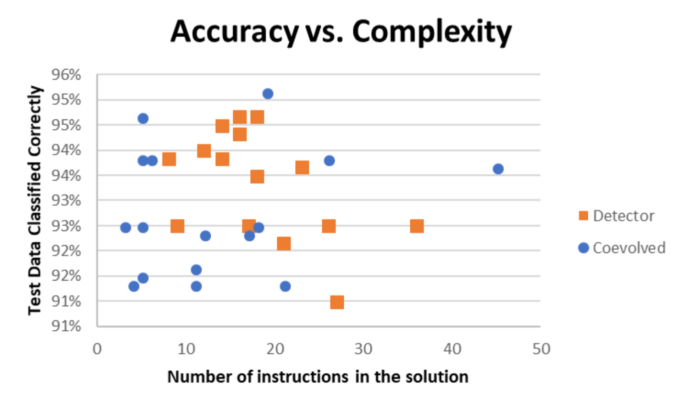

Fig. 3. Accuracy vs. Complexity

Then, using the same training and test data, we implement the artificial arms race framework and add in the co-evolution system. Table X shows the performance of one of the GP based malware detectors evolved via this arms race. The results of the evolved detectors are comparable with the results of the C5.0 and GP based detectors that are trained stand alone, i.e. outside of the artificial arms race framework. Moreover, the co-evolved malware detector uses fewer features compared to the stand alone ones. In summary, the addition of co-evolution seems to reduce the overall complexity of the detectors evolved without impacting their performance.

While there is no significant difference in accuracy with this kind of approach, the quality of solutions appears to improve. Figure 3 shows that the average number of instructions, which

TABLE VI
RESULTS OF GP WITH 15 PERMISSIONS ONLY

		True class		
		Malware	Benign	Total
Predicted class	Malware	272	28	90%
	Benign	56	244	81%
	Total	83%	90%	86%

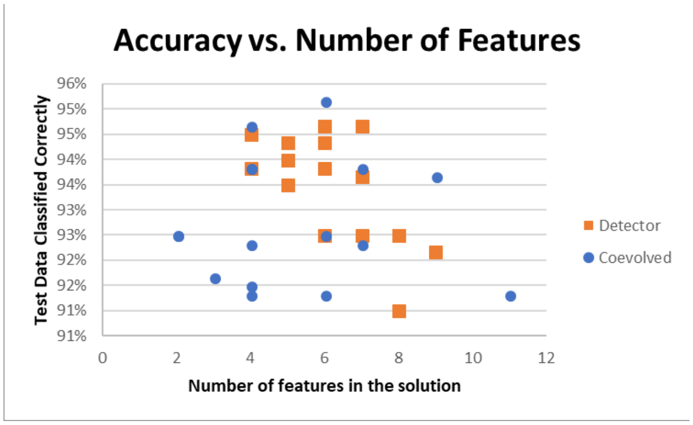

Fig. 4. Accuracy vs. Number of Features

TABLE VII
RESULTS OF C5.0 WITH 15 PERMISSIONS ONLY

		True class		
		Malware	Benign	Total
Predicted class	Malware	260	29	90%
	Benign	40	271	87%
	Total	87%	90%	88%

TABLE VIII
RESULTS OF GP WITH 15 PERMISSIONS AND 8 CODE FEATURES

		True class		
		Malware	Benign	Total
Predicted class	Malware	295	5	98%
	Benign	27	273	91%
	Total	92%	98%	95%

TABLE IX
RESULTS OF C5.0 WITH 15 PERMISSIONS AND 8 CODE FEATURES

		True class		
		Malware	Benign	Total
Predicted class	Malware	289	12	96%
	Benign	11	288	96%
	Total	96%	96%	96%

TABLE X
A SAMPLE RESULT OF GP WITH 15 PERMISSIONS AND 8 CODE FEATURES WITH CO-EVOLUTION

		True class		
		Malware	Benign	Total
Predicted class	Malware	294	6	98%
	Benign	39	261	87%
	Total	88%	98%	92%

TABLE XI
A SAMPLE RESULT OF GP WITH 15 PERMISSIONS AND 8 CODE FEATURES WITHOUT CO-EVOLUTION

		True class		
		Malware	Benign	Total
Predicted class	Malware	278	15	95%
	Benign	22	285	93%
	Total	93%	95%	94%

is used to measure complexity, is reduced without significantly alternating the accuracy of the process. The same holds true regarding the average number of features selected by the learning algorithm from the given set of features, as is seen in Figure 4.

Below is an example of a co-evolved solution to detect a malware using the proposed artificial arms race framework. As discussed earlier, in this framework, the detector is co-evolved against the malware generator to detect the generated malware. The resulting detector solution is in the form of a program:

```
r[6] = exp(in[READ_PHONE_STATE])
Bid Malware = r[6] − 200
Bid Clean = exp(in[direct method count])
```

This particular solution focuses on two values:

1) READ_PHONE_STATE
2) Number of Direct methods

The resulting rules followed by this program amount to: If app has READ_PHONE_STATE and allow values for Direct methods, then it is malware. The performance of this solution program's rules is presented in Table X.

On the other hand, if we analyze a solution of a GP based detector that was trained stand alone, outside of the artificial arms race framework, the solution program appears as the following:

```
if( r[5] <= 62 )
if( r[0] > r[1] )
r[4] = r[1] / in[INSTALL_PACKAGES]
if( r[0] > in[SEND_SMS] )
r[0] = r[4] − 31
r[4] = log(r[0])
if( r[4] > in[READ_PHONE_STATE] )
r[1] = sin(in[RECEIVE_BOOT_COMPLETED])
```

The performance of the above program is given in Table XI. The resulting rules followed by this program are: If the program has READ_PHONE_STATE and SEND_SMS, then it is malware.

To better understand how the evolved solutions - the population - under the artificial arms race framework compares with the single best solution, we analyze the diversity of the solutions in Figure 5. The population's combined coverage of knowledge reaches a very high accuracy (100%) given the collaboration of the top 20 solutions (programs) out of the 100 solutions evolved. This indicates that the solutions evolved have enough diversity to be able to recognize different malicious behaviours in the apps. From a co-evolutionary perspective, this ensures the survival of useful programs even if they are not the single best solution. In other words, this diversity enables us to generate (evolve) different rules that have the potential of detecting different variants of malware.

VI. CONCLUSION AND FUTURE WORK

Exploring whether an emulated artificial arms race between mobile malware and detectors could improve the ability of the detector was the primary goal of this research. The first

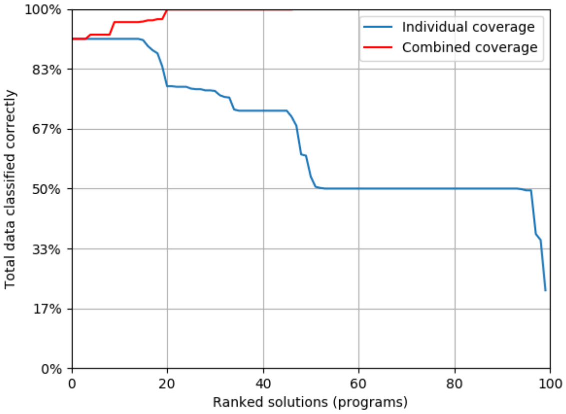

Fig. 5. Accuracy vs. Populations accumulated knowledge

step in studying this goal commenced with the analysis of the population's shared knowledge, and how it might be affected by the co-evolution process. Initial findings indicate that the building of teams from the detectors may lead to a more robust solution compared to building teams from stand alone detectors. The performance of the framework demonstrate that it is possible to generate new rules with high accuracy against the new variants of malware using a co-evolution process to automatically emulate an artificial arms race between malware and detectors. Moreover, the proposed artificial arms race framework shows an increase in knowledge about the task. This observation is also supported by the previous work [1], which demonstrates that the more knowledge the detector has, the better it can adapt to future malware behaviours. Last but not the least, the proposed framework could be used to generate realistic and different malware / app behaviours.

In this work, the proposed framework was evaluated using Android apps. However, given that the approach is not using features dependent of the platform and / or architecture, it could be applied to other mobile platforms, apps and malware. Furthermore, other features could be added and extracted based on the information provided by the mobile platform employed. Using one or more malware generators and replacing the malware detectors with other methods of classification / detection is also possible and may be used in the future to improve the performance.

Acknowledgments

This research is supported by Raytheon SAS. The research is conducted as part of the Dalhousie NIMS Lab at: https://projects.cs.dal.ca/projectx/.

References

[1] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu, "Context-aware, adaptive, and scalable android malware detection through online learning," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 157–175, June 2017.

[2] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik, "A new android malware detection approach using bayesian classification," in *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, March 2013, pp. 121–128.

[3] A. Skovoroda and D. Gamayunov, "Review of the mobile malware detection approaches," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, March 2015, pp. 600–603.

[4] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in android applications for malicious application detection," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 11, pp. 1869–1882, Nov 2014.

[5] P. Chester, C. Jones, M. W. Mkaouer, and D. E. Krutz, "M-perm: A lightweight detector for android permission gaps," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, May 2017, pp. 217–218.

[6] D. Arp, M. Spreitzenbarth, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA*, pp. 1–12, ISBN 1–891 562–35–5, Febuary 2014. [Online]. Available: https://www.sec.cs.tu-bs.de/~danarp/drebin/

[7] H. G. Kayacık, A. N. Zincir-Heywood, and M. I. Heywood, "Evolutionary computation as an artificial attacker: generating evasion attacks for detector vulnerability testing," *Evolutionary Intelligence*, vol. 4, no. 4, pp. 243–266, Dec 2011. [Online]. Available: https://doi.org/10.1007/s12065-011-0065-0

[8] G. Kayack, A. Zincir-Heywood, and M. I. Heywood, "Can a good offense be a good defense? vulnerability testing of anomaly detectors through an artificial arms race," vol. 11, pp. 4366–4383, 10 2011.

[9] P. Fogla and W. Lee, "Evading network anomaly detection systems: formal reasoning and practical techniques," in *Proceedings of the 13th ACM conference on Computer and communications security*. ACM, 2006, pp. 59–68.

[10] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks." in *Lisa*, vol. 99, no. 1, 1999, pp. 229–238.

[11] O. L. Fraser, N. Zincir-Heywood, M. Heywood, and J. T. Jacobs, "Return-oriented programme evolution with roper: A proof of concept," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '17. New York, NY, USA: ACM, 2017, pp. 1447–1454. [Online]. Available: http://doi.acm.org/10.1145/3067695.3082508

[12] S. Noreen, S. Murtaza, M. Z. Shafiq, and M. Farooq, "Evolvable malware," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. ACM, 2009, pp. 1569–1576.

[13] G. Meng, Y. Xue, C. Mahinthan, A. Narayanan, Y. Liu, J. Zhang, and T. Chen, "Mystique: Evolving android malware for auditing anti-malware tools," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 365–376.

[14] J. Sahs and L. Khan, "A machine learning approach to android malware detection," in *Intelligence and security informatics conference (eisic), 2012 european*. IEEE, 2012, pp. 141–147.

[15] M. Brameier and W. Banzhaf, "Effective linear genetic programming," Neural Networks in Medical Data Mining IEEE Transactions on Evolutionary Computation, Tech. Rep., 2001.

[16] R. Quinlan, "Data mining tools see5 and c5.0," 2018. [Online]. Available: https://www.rulequest.com/see5-info.html

[17] L. Sun, X. Wei, J. Zhang, L. He, P. S. Yu, and W. Srisa-an, "Contaminant removal for android malware detection systems," *arXiv preprint arXiv:1711.02715*, 2017.