Brown *et al. EURASIP Journal on Information Security* (2017) 2017:7
DOI 10.1186/s13635-017-0059-2

EURASIP Journal on Information Security

RESEARCH Open Access

An artificial immunity approach to malware detection in a mobile platform

James Brown, Mohd Anwar* and Gerry Dozier

Abstract

Inspired by the human immune system, we explore the development of a new Multiple-Detector Set Artificial Immune System (mAIS) for the detection of mobile malware based on the information flows in Android apps. mAISs differ from conventional AISs in that multiple-detector sets are evolved concurrently via negative selection. Typically, the first detector set is composed of detectors that match information flows associated with malicious apps while the second detector set is composed of detectors that match the information flows associated with benign apps. The mAIS presented in this paper incorporates feature selection along with a negative selection technique known as the split detector method (SDM). This new mAIS has been compared with a variety of conventional AISs and mAISs using a dataset of information flows captured from malicious and benign Android applications. This approach achieved a 93.33% accuracy with a true positive rate of 86.67% and a false positive rate of 0.00%.

Keywords: Mobile malware, Static flow analysis, Artificial immunity, Negative selection, Feature selection

1 Introduction

Mobile and embedded devices have proliferated through contemporary society at an exponential rate. The number of mobile devices in use will grow from over 11 billion in 2016 to over 16 billion by 2020 [1]. The Android operating system, specifically, accounts for 82.8% of current smartphones in use for 2015 Q2 [2]. Android encompasses a broad and far-reaching range of users and retailers, with its appeal being convenience, low cost, and customization. With the Android OS, and mobile industry in general, experiencing a meteoric rise, mobile malware has become a significant threat.

Mobile malware has undergone a large increase in sheer number and a diversification in type. Android malware accounts for nearly 97% of all mobile malware [3]. As of February 2014, over 10 million malicious Android apps have been released [4] to the public through various official and unofficial mediums. As of 2014, SophosLabs has recorded well over 650,000 individual pieces of malware for Android [5].

Through the use of malware on mobile devices, cybercriminals can steal private user information. Mobile malware is also capable of tracking user GPS and WiFi location and capturing contact data. Cyber criminals also aim to deconstruct and decompile popular apps like Angry Birds and publish versions with embedded malicious code for free download. Unfortunately, many users fall victim to this sort of repackaging attack.

Typically, mobile malware is designed to target a specific vulnerability in the OS to attack and exploit. Common vulnerabilities exploited by malware developers on the Android platform include:

- Over-granting permissions
- Insecure transmission
- Insecure data storage
- Leakage through logs
- IPC endpoint vulnerabilities

These vulnerabilities make end-users prone to previously discovered as well as zero-day [6] mobile malware. Therefore, to address the growing threat of Android malware, we developed a Multiple-Detector Set Artificial Immune System (mAIS) and a validation scheme. The mAIS is a variant of the AIS, where a detector set for "non-self" detection is also generated, much the same as the standard AIS. The key difference between mAIS and

* Correspondence: manwar@ncat.edu
Department of Computer Science, North Carolina A&T State University, Greensboro, NC, USA

standard AIS is that the mAIS generates an additional detector set for "self" detection. The outputs from both the non-self detector set and self detector set are then used for classification. For Android malware detection, we were able to achieve 93.33% accuracy with a true positive rate of 86.67% and a false positive rate of 0.00%.

The contributions of this research are as follows:

- The development of a Multiple-Detector Set Artificial Immune System model for Android app classification into malicious or benign categories.
- Training, validation, and testing of mAIS classifier using k-fold validation with variable fold sizes to generalize our results across our dataset.
- A comparative study of the classification performances of different variants of mAIS.

The paper is organized as follows: in Section 2, we present review of related works, Section 3 describes our mAIS[1] implementation, and Section 4 is an overview of our system model. Our dataset and experiments are detailed in Section 5. In Section 6, we present the results. Conclusions drawn and future work are explained in Section 7.

2 Related work

2.1 Malware detection techniques

A common approach for malware detection is pattern recognition, where a malicious entity is detected by comparing deterministic patterns found in previously discovered malware. Popular antivirus software such as Avast [7] and McAfee [8] primarily use pattern recognition for computer security. Patterns could include coding techniques unique to a particular family of apps or malicious developer, third-party libraries, or suspicious code obfuscation techniques. A major drawback of this approach is its inability to identify previously unrecognized malware or zero-day malware. Since pattern recognition relies upon analysis of pre-existing malware, new malware samples which have not been discovered have the potential to go undetected.

Machine learning techniques are widely used for malware detection. These techniques are specifically designed to differentiate between benign software and malware by identifying discriminating features [9–12]. The key difference between each literature example is the features used during the training and classification phase of their model generation. API calls, permissions, flows, etc. are commonly used features from which researchers capture from an app through static code analysis or dynamic runtime analysis. Once these features are extracted, a two-class classification algorithm, such as Decision Tree, Naïve Bayes, Bayesian Networks, k-Means, Deep Learning, and Logistic Regression, is applied to dataset.

2.2 Overview of Artificial Immune Systems

AISs are biologically inspired problem-solving methods modeled after the human immune system [13, 14]. AISs have been used to solve real-world complex problems in the areas of cybersecurity, robotics, fraud detection, and anomaly detection [15], just to name a few. AISs are robust, self-healing, and self-contained problem-solving solutions, with an ability to dynamically adapt to its environment. These traits led us to the AIS, and other solutions derived from it, as a potential solution for mobile malware detection.

The standard AIS [16] is typically used to solve two-class classification problems (0/1, self/non-self, etc.). The standard AIS is composed of a set of detectors and employs an evolutionary process [17] as follows. Initially, a set of detectors is randomly generated. These detectors are called immature detectors and undergo a process known as the negative selection [16, 17]. During negative selection, each immature detector is checked to see if they match a "self" instance. If this occurs, the immature detector is replaced by a randomly generated immature detector. Those immature detectors that fail to match a "self" instance are promoted to being a mature detector. Mature detectors are given a lifetime of t_{mature} and must match at least m_{mature} number non-self instances during their lifetime. Mature detectors that match the requisite number of non-self instances over their lifetime are promoted to being memory detectors and are assigned a lifetime of t_{memory} where $t_{memory} >> t_{mature}$. This composition of immature, mature, and memory detectors make AISs well suited for dynamic environments because memory and mature detectors ensure detection of previously encountered non-self, and immature detectors afford relative protection from unseen non-self.

Since the development of the standard AIS, a number of AIS variants have been developed. In [18, 19], Hou developed an mAIS that evolved three detector sets in an effort to solve user authentication and identification through iris recognition. In [20, 21], Idris and Muhammed developed mAISs for solving e-mail spam classification problem. The spam detection was a two-class classification problem; however, Idris and Muhammed used a standard set of detectors for detecting non-self (spam) instances and used a second detector set for detecting self (good e-mail) instances. The net effect was that the mAIS was just as good at detecting spam and did so with a dramatically lower false positive rate.

An advantage of artificial immunity over pattern recognition and other machine learning techniques are as follows: (1) an AIS' aim is not to explicitly search for discriminating features between two-classes but to implicitly find those salient features through negative selection and (2) a robust, dynamic, and self-healing solution which can easily be applied in real-time environments. Typically, AISs have been applied to traditional desktop computer environments to protect against internet-based attacks, such as the work of Ramakrishnan and Srinivasan [22], where they reviewed several AIS-based security systems. In [23], Zhao applied an AIS for smartphone malware detection in a dynamic environment. Log preprocessing and system API access were dynamically captured and used as training features. They were able to achieve an 87.7% true positive rate with an 8.2% false positive rate. The differences between Zhao's and our approach lie in feature extraction and the use of multiple-detector sets. Instead of dynamically capturing features, our approach utilizes static flow analysis. When using dynamic analysis, the app is treated as a black box. Therefore, an app can inconspicuously hide its true behavior using advanced obfuscation techniques. In static flow analysis, app behavior is much harder to obfuscate because the static analysis algorithm tracks data-flow through offline code analysis and simulated runtime environment behavior.

3 Multiple-Detector Set Artificial Immune System

False positives (FPs), type I errors occur when a benign app is misclassified as malicious, creating confusion and unnecessary hassle for users. False negatives (FN), type II errors, pose a greater threat to user data security and privacy because this occurs when a malicious app is classified as benign. Therefore, a malicious app is able to gain access to the user's device without impediment or detection. Our results show the standard AIS has the ability to detect malicious apps with a true positive rate (TPR), correct classification rate, of 80.00%. Unfortunately, the standard AIS also resulted in a false positive rate (FPR), benign app misclassified as malicious rate, of 73.33% and a true negative rate (TNR), malicious app misclassified as benign, of 20.00%. The goal of this research is to increase the TPR and decrease FPR and TNR, which would signify a robust and optimal model. As evidenced in Idris and Muhammed [20] for e-mail spam identification, the mAIS is proficient at detecting "non-self" instances and significantly reducing FPs and FNs. Therefore, we applied the mAIS to the Android malware detection problem. When used with the Split Detector Method (SDM) [24] and Genetic and Evolutionary Feature Selection (GEFeS) [5], the mAIS outperformed the standard AIS with a TPR of 86.67% and a FPR and FNR of 0.00% and 13.33%, respectively.

3.1 Detector set generation

To develop the mAIS, two independent detector sets are generated: one detects "non-self" app instances and the other detects "self" app instances. During detector set generation, the non-self detector set is trained similarly to the standard AIS. During negative selection, if an immature detector matches a "self" app instance, it is discarded, if SDM is used, the detector is split. Theoretically, this results in a detector set which only detects "non-self". This is shown in Fig. 1. For the self detector set generation, "self" is replaced by "non-self" during training. Therefore, if an immature detector matches a "non-self" instance, it is either discarded or spilt. This is shown in Fig. 2. Detectors that survived negative selection during the generation of non-self detectors and self detectors are considered mature non-self detectors and mature self detectors, respectively.

3.2 Proportion-based classification

The two detector sets are used in a proportion-based classification method to detect unknown instances. If the proportion of non-self detectors that match an instance is greater than the proportion of self detectors that match the instance, the instance is classified as "non-self". Likewise, if the proportion of self detectors that match are greater than the proportion of "non-self" detectors, the instance is classified as "self". Since FNs have the potential to cause significantly more damage than FPs, in the rare occurrence that the proportions of the two detector sets are equal, the instance is classified as "non-self".

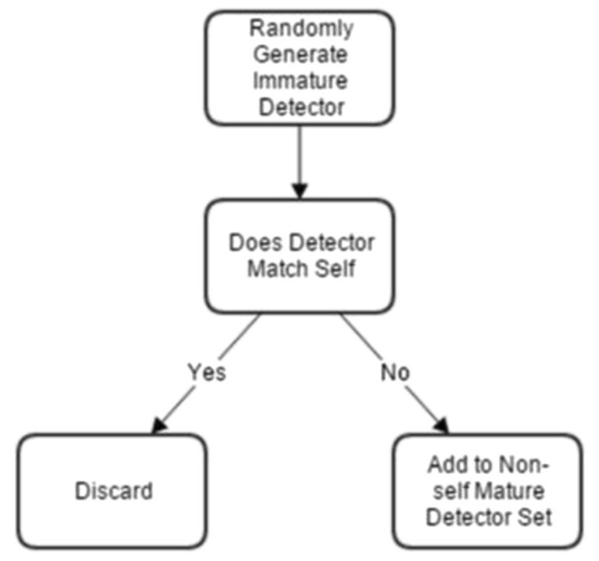

Randomly Generate Immature Detector
Does Detector Match Self
Yes
No
Discard
Add to Non-self Mature Detector Set

Fig. 1 Non-self detector set generation

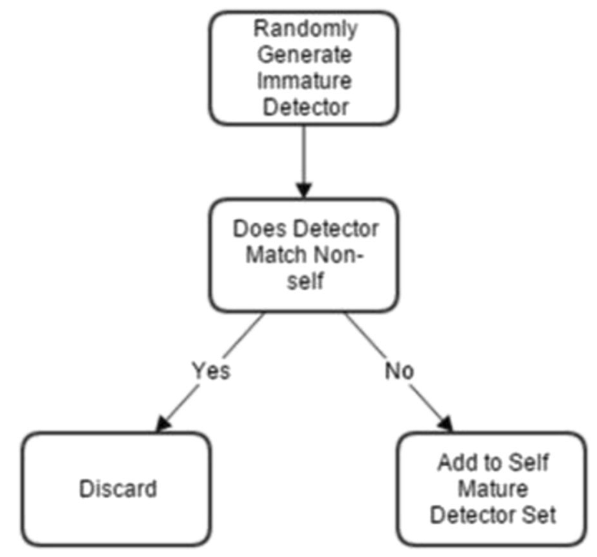

Randomly Generate Immature Detector
Does Detector Match Non-self
Yes
No
Discard
Add to Self Mature Detector Set

Fig. 2 Self detector set generation

3.3 Any-*r* interval matching rule

The any-*r* interval matching rule is used to determine if a detector matches an instance (app). Each detector is composed of 590 intervals, where each interval corresponds to a specific feature from the dataset feature vectors. To determine if a detector matches an instance, first, an r value is selected. If ≥*r*, from the particular instance, are contained in the detector's intervals, the detector matches.

3.4 Variations

GEFeS and SDM were used to create four variations of the mAIS. Our results show that the use of GEFeS in conjunction with SDM improves the performances of the standard AIS and mAIS. The techniques tested and compared in this paper are as follows:

- GEFeS—GEFeS only was used as the baseline approach. GEFeS evolves a subset of salient features (feature mask) which reduces computational complexity and increases accuracy. Feature masks are evolved to identify and select optimal/near-optimal features from a training set and eliminate irrelevant or redundant features, which create a well-defined distinction between self (benign) and non-self (malicious) apps. To evaluate a feature mask, the k-nearest neighbor, where $k = 1$, is used to classify an app using the Manhattan Distance as the similarity score. During the Manhattan Distance calculation, a feature mask is applied. The equation with a feature mask applied is shown in Fig. 3 where m represents a feature mask and X and Y represent apps within the dataset. GEFeS aims to reduce the number of classification errors, which produces an optimal/near-optimal feature mask.

$$d(X,Y) = \sum_{i=0}^{n-1} m_i * |x_i - y_i|$$

Fig. 3 Manhattan distance calculation with a feature mask

A Steady-State Genetic Algorithm was used as the evolution algorithm. The parameters were as such: a population size of 20 candidate feature masks, Tournament Selection, BLX-0.0 crossover, Gaussian Mutation where $\mu = 0$ with a standard deviation of 20% of the range (upper and lower bounds) of an associated feature. GEFeS was run 30 times on the 30-28 dataset where training set consisted of 20 feature vectors associated with 10 benign and 10 malicious apps, the validation set consisted of 19 feature vectors associated with 10 benign and 9 malicious apps, and the test set also consisted of 19 feature vectors associated with 10 benign and 9 malicious apps.

- SDM—During the negative selection training process for the standard AIS and the non-self detector set for the mAIS, SDM requires that if a detector matches a self instance, instead of being discarded, it is split into upper- and lower-bound detectors. The two resulting detectors undergo a negative selection but are not split again. For the self detector set during mAIS negative selection, if a detector matches a non-self instance, it is split into upper- and lower-bound detectors, which, subsequently, undergo negative selection without a split. This process increases search space coverage, by generating detectors which wrap tighter around non-self hypothesis space for self detector set generation and self hypothesis space for standard AIS and non-self detector set generation.

With a computational complexity for detector set generation of O(n^3), where n = the initial detector set size, SDM requires significantly more resources than the standard AIS and mAIS without SDM, where detector set generation has a computation complexity of O(n). Although, SDM can increase true positive rates, it can also adversely increase false positives, which reduces the overall accuracy of the system. This is another drawback of the SDM. In our experiments, we were able to combat this by combining SDM with GEFeS, due to GEFeS' ability to clearly discriminate between "self" and "non-self."

GEFeS and SDM were applied to the standard AIS and mAIS, therefore, creating four variants:

- (−SDM, −GEFeS)—Neither SDM nor GEFeS were applied.
- (+SDM, −GEFeS)—SDM only.
- (−SDM, +GEFeS)—GEFeS only.
- (+SDM, +GEFeS)—Both SDM and GEFeS were applied.

4 System model overview

We used static flow analysis to capture information flows in Android apps. Based on the flows, we create a feature vector of all the information flows captured in each app. This process is outlined in the following subsections:

A. *Flow extraction*. The first step involves static flow analysis on an app's code. This is used to capture the flows from an app, which are then used for feature vector creation.
B. *Feature vector development*. A feature vector is a set of all flow combinations (i.e., source-to-sink combinations) possible in an app. Each index inside the feature vectors is the sum of each individual flow found in the app. Therefore, if a flow is found to exist inside the app, the corresponding index in the feature vector is incremented. The feature vectors of flows are used for malicious app classification.

4.1 Flow extraction

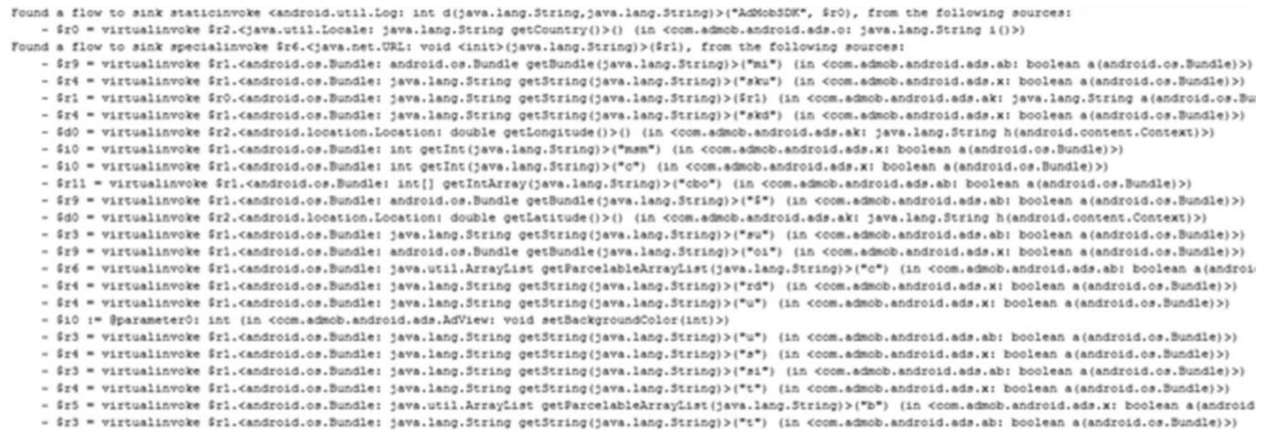
Flow extraction is the process of gathering source-to-sink connections (flows) from an app. A source is a java method which can access and return sensitive user or device information, e. g., getContacts() returns contact data stored in a user's mobile device. A sink is a java method which can relay sensitive information from a source method, e.g., sendSMS(). The connection between a source method and a sink method is called a flow. FlowDroid [25] was used for flow analysis on the dataset.

1. *FlowDroid results*: FlowDroid is a combination of various Java archive (.jar) files, which are consolidated into a single command prompt. FlowDroid results give source-to-sink connections in a particular app. Figure 4 shows a truncated FlowDroid report which describes the captured flows from an app.

4.2 Feature vector development

The FlowDroid results are parsed to find all flows in a particular app, which will then be used to create a unique feature vector. This section describes: (1) how flows are mapped into a matrix and (2) how they are then copied into a one-dimensional feature vector.

1. *Flow matrix creation*: Firstly, the list of all sources and sinks that can be found in an Android app's source file are gathered from a predefined text file, which was downloaded from the FlowDroid GitHub repository. These sources and sinks are then inserted into two individual vectors. The two vectors are combined into a single two-dimensional matrix. Each position within the matrix represents a unique source-to-sink connection (flow) dependent on the corresponding positions of the source's and sink's in their original respective vectors.
2. *Feature vector creation*: For each individual app, we create a copy of the matrix. The two-dimensional matrix is populated with extracted flows from the



```
Found a flow to sink staticinvoke <android.util.Log: int d(java.lang.String,java.lang.String)>("AdMobSDK", $r0), from the following sources:
    - $r0 = virtualinvoke $r2.<java.util.Locale: java.lang.String getCountry()>() (in <com.admob.android.ads.o: java.lang.String i()>)
Found a flow to sink specialinvoke $r6.<java.net.URL: void <init>(java.lang.String)>($r1), from the following sources:
    - $r9 = virtualinvoke $r1.<android.os.Bundle: android.os.Bundle getBundle(java.lang.String)>("mi") (in <com.admob.android.ads.ab: boolean a(android.os.Bundle)>)
    - $r4 = virtualinvoke $r1.<android.os.Bundle: java.lang.String getString(java.lang.String)>("sku") (in <com.admob.android.ads.x: boolean a(android.os.Bundle)>)
    - $r1 = virtualinvoke $r0.<android.os.Bundle: java.lang.String getString(java.lang.String)>($r1) (in <com.admob.android.ads.ak: java.lang.String a(android.os.Bu
    - $r4 = virtualinvoke $r1.<android.os.Bundle: java.lang.String getString(java.lang.String)>("skd") (in <com.admob.android.ads.x: boolean a(android.os.Bundle)>)
    - $d0 = virtualinvoke $r2.<android.location.Location: double getLongitude()>() (in <com.admob.android.ads.ak: java.lang.String h(android.content.Context)>)
    - $i0 = virtualinvoke $r1.<android.os.Bundle: int getInt(java.lang.String)>("msm") (in <com.admob.android.ads.x: boolean a(android.os.Bundle)>)
    - $i0 = virtualinvoke $r1.<android.os.Bundle: int getInt(java.lang.String)>("c") (in <com.admob.android.ads.x: boolean a(android.os.Bundle)>)
    - $r11 = virtualinvoke $r1.<android.os.Bundle: int[] getIntArray(java.lang.String)>("cbo") (in <com.admob.android.ads.ab: boolean a(android.os.Bundle)>)
    - $r9 = virtualinvoke $r1.<android.os.Bundle: android.os.Bundle getBundle(java.lang.String)>("$") (in <com.admob.android.ads.ab: boolean a(android.os.Bundle)>)
    - $d0 = virtualinvoke $r2.<android.location.Location: double getLatitude()>() (in <com.admob.android.ads.ak: java.lang.String h(android.content.Context)>)
    - $r3 = virtualinvoke $r1.<android.os.Bundle: java.lang.String getString(java.lang.String)>("su") (in <com.admob.android.ads.ab: boolean a(android.os.Bundle)>)
    - $r9 = virtualinvoke $r1.<android.os.Bundle: android.os.Bundle getBundle(java.lang.String)>("oi") (in <com.admob.android.ads.x: boolean a(android.os.Bundle)>)
    - $r6 = virtualinvoke $r1.<android.os.Bundle: java.util.ArrayList getParcelableArrayList(java.lang.String)>("c") (in <com.admob.android.ads.ab: boolean a(androi
    - $r4 = virtualinvoke $r1.<android.os.Bundle: java.lang.String getString(java.lang.String)>("rd") (in <com.admob.android.ads.x: boolean a(android.os.Bundle)>)
    - $r4 = virtualinvoke $r1.<android.os.Bundle: java.lang.String getString(java.lang.String)>("u") (in <com.admob.android.ads.x: boolean a(android.os.Bundle)>)
    - $i0 := @parameter0: int (in <com.admob.android.ads.AdView: void setBackgroundColor(int)>)
    - $r3 = virtualinvoke $r1.<android.os.Bundle: java.lang.String getString(java.lang.String)>("u") (in <com.admob.android.ads.ab: boolean a(android.os.Bundle)>)
    - $r4 = virtualinvoke $r1.<android.os.Bundle: java.lang.String getString(java.lang.String)>("s") (in <com.admob.android.ads.x: boolean a(android.os.Bundle)>)
    - $r3 = virtualinvoke $r1.<android.os.Bundle: java.lang.String getString(java.lang.String)>("si") (in <com.admob.android.ads.ab: boolean a(android.os.Bundle)>)
    - $r4 = virtualinvoke $r1.<android.os.Bundle: java.lang.String getString(java.lang.String)>("t") (in <com.admob.android.ads.x: boolean a(android.os.Bundle)>)
    - $r5 = virtualinvoke $r1.<android.os.Bundle: java.util.ArrayList getParcelableArrayList(java.lang.String)>("b") (in <com.admob.android.ads.x: boolean a(android
    - $r3 = virtualinvoke $r1.<android.os.Bundle: java.lang.String getString(java.lang.String)>("t") (in <com.admob.android.ads.ab: boolean a(android.os.Bundle)>)
```

Fig. 4 Sample FlowDroid results for Adobe Reader App. Sinks are shown on lines beginning with "Found a flow…" and sources that flow to the sink are on the *dashed lines* underneath

FlowDroid results. Each position within the matrix represents a sum of that particular flow within the app. The process is as follows and delineated in Fig. 5.

- Let (so_i, sk_t) represent a unique source-to-sink connection or flow in app *ap*. For every individual connection (so_i, sk_t) in *ap*, the correlating count $c_{i,t}$ is incremented by one.
- $c_{i,t}$ is inserted into a two-dimensional matrix *tdm* at index [i, t].
- This is repeated for each unique (so_i, sk_t) where $0 \leq i < N(source)$ and $0 \leq t < N(sink)$. $N(source)$ and $N(sink)$ are the total number of the sources and sinks, respectively.
- Vector v is initialized, and the contents of *tdm* are transferred to v in a row-major order.

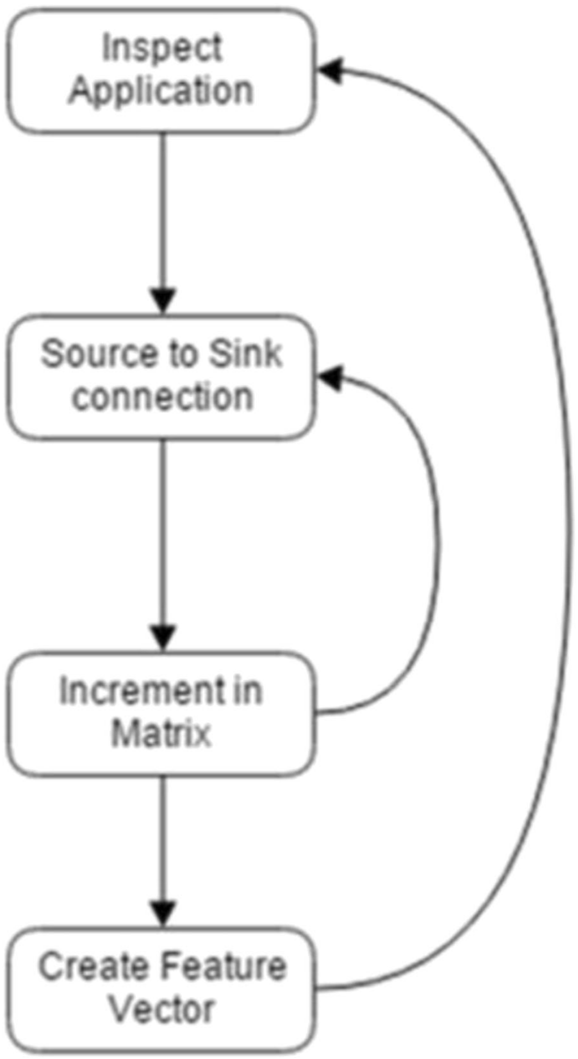


Fig. 5 Feature vector creation with flows

Suppose a particular flow f occurs 15 times in an app. Let the index of flow f in the two-dimensional matrix *tdm* be (5, 4), and therefore, *tdm* [1, 4] contains the number 15. When vector v is created, the index of flow f will be 20, in accordance with the row-major order. Therefore, index 20 of vector v will contain 15.

4.2.1 Feature vector truncation

All of the generated vectors (one per app) are appended into a two-dimensional matrix. Each row is a feature vector of an app. If a flow is not present in any of the apps' feature vectors, its corresponding column is removed from the matrix. This is repeated for every flow that was not present in any of the apps. The truncation removes approximately 21,000 placeholders for possible flows that are not present in our dataset. The resulting matrix is written out to a file. Each row is given a new line to represent an independent app feature vector. The feature vectors are subsequently used during detector set generation (Section 3.1) to create real-valued detectors.

5 Dataset and experiments

Our dataset consists of 30 benign apps gathered from the Google Play Store and 28 malicious apps obtained from the Android Malware Genome Project [26]. The Android Malware Genome Project is a repository of 1200+ malicious Android apps, organized in families of related apps. Our primary focus was to gather as many apps as possible and extract their features; therefore, we were unable to explicitly target specific family of apps. Due to Flowdroid's increased runtime for larger apps, we decided to only use apps ≤1 MB. Dataset collection was conducted over the course of several weeks, 24 h a day. Flowdroid was given 24 h to analyze an app. If the analyses process took longer than 24 h, the process was canceled and the app was discarded. The process ran on a 2.3 GHz Intel Xeon server with 120 GB of RAM. This server contained the largest amount of RAM we were able to solely dedicate to our experiments.

Initially, our dataset contained 30 malicious apps but during feature vector similarity scoring using the Manhattan Distance metric, we discovered the existence of two malicious apps that consistently yielded a Manhattan Distance of 0.0 with multiple benign apps. In other words, the two malicious apps are identical to multiple benign apps. These ambiguous apps adversely affect the two-class identification process employed in this research. Therefore, we chose to remove them from our dataset for further study. For this research, the removal of the two apps resulted in a dataset of 30 benign apps and 28 malicious apps.

Six initial parameters were set before conducting each experiment: number of malicious apps, number of benign apps, initial detector set size, width of detectors, Tao (SDM only), r_{ns}, and r_{s}. Initial detector set size specifies the initial number of immature detectors to generate for negative selection. This applies to both non-self detector sets and self detector sets. The width of detectors specifies the interval for each range within a detector. Tao is used for SDM only and signifies the number of standard deviations away from the mean a detector should be split. Lastly, r_{ns} and r_{s} signify the r values for non-self detector set and self detector set negative selection, respectively. Each parameter and their corresponding values can be seen in Table 1.

Limitations of our approach are as follows: (1) a lack of an extensive dataset for experimentation and (2) a lack of categorical information for collected data pertaining to family designations. We were forced to use a relatively small sample size for experimentation because of processing and time constraints. This has limited our ability to determine the scalability and adaptability of the AIS and mAIS for real-world applications. We were unable to label the sample size collected with the appropriate family tag because of our current processing power. This will be remedied when we are able to upgrade our server configuration to analyze the larger and more complex apps.

6 Results

For the standard AISs and mAISs, a sixfold cross-validation was used where the training set consisted of 38 feature vectors associated with 20 benign and 18 malicious apps, the tuning set consisted of 10 feature vectors associated with 5 benign and 5 malicious apps and the test set also consisted of 10 feature vectors associated with 5 benign and 5 malicious apps similar to the setup used in [27].

Table 1 Parameters used for AIS and mAIS development

Parameter	Model	Description	Value/range
Number of malicious apps	AIS, mAIS	The number of malicious apps in our test set.	5
Number of benign apps	AIS, mAIS	The number of benign apps in our test set	5
Initial detector set	AIS, mAIS	Number of detectors randomly generated	1000
Width of detector	AIS, mAIS	The range for each interval during detector generation	1.0
Tao	AIS, mAIS	Number of standard deviations to add or subtract from the mean when detectors are split	1
rns	AIS, mAIS	The *r* value for the "non-self" identifying detector set	+GEFeS (1…100)[a] −GEFeS (100…200)
rs	mAIS	The *r* value for the "self" identifying detector set	+GEFeS (1…100)[a] −GEFeS (100…200)
Non-self detector set size	AIS, mAIS	The number of non-self mature detectors.	(1…40000)[b]
Self detector set size	mAIS	The number self mature detectors	(1…40000)[b]
Number of malicious apps detected	AIS, mAIS	Standard AIS: The number of malicious apps the mature detector set matched. mAIS: the number of malicious apps our committee machine was able to successfully classify.	(0…5)
Number of benign apps detected	AIS, mAIS	Standard AIS: The number of benign apps the mature detector set did not match. mAIS: The number of benign apps our committee machine was able to unsuccessfully classify.	(0…5)
Accuracy	AIS, mAIS	Percentage of correctly classified apps.	(0…100%)
True positive rate	AIS, mAIS	Percentage of malicious apps correctly classified as malicious	(0…100%)
True negative rate	AIS, mAIS	Percentage of benign apps correctly classified as benign	(0…100%)
False positive rate	AIS, mAIS	Percentage of classified benign apps incorrectly classified as malicious	(0…100%)
False negative rate	AIS, mAIS	Percentage of malicious apps incorrectly classified as benign	(0…100%)

[a]+GEFeS denotes a variation that used GEFeS, −GEFeS denotes a variation that does not use GEFeS
[b]The maximum number for split detectors is $2 \times n \times d$ where n = training set size and d = initial detector set size. Therefore, $2 \times 20 \times 1000 = 40{,}000$

The cross-validation training was as follows. For the first fold, the training, tuning, and test sets were as explained earlier and the standard AISs and mAISs were 30 times. On a particular run, a sweep of the *r* values, for the self and non-self detector sets, was used to discover the best *r* value, for the any-*r* intervals matching rule (where the self and non-self detector sets have their own independent *r* value). For a particular *r* value, the two detector sets were randomly generated and exposed to the self and non-self training instances. Those detectors that failed to match an instance were promoted to mature detectors. The mature detectors of each detector set were exposed to their respective instances of the tuning set. The best-performing self *r* value/detector set pairing and the best-performing non-self r value/detector set pairing on the tuning set were retained. They were retained to be exposed to the test set as a proportion based. After being exposed to the test set, the statistics, such as accuracy, TPR, FPR, TNR, and FNR were recorded.

After the recording of the statistics for the first fold, the 10 instances from the test set were removed and appended to the training set. The 10 instances from the tuning set became the new test set. The first five self and five non-self instances (five benign, five malicious apps) were removed from the training set and appended to the tuning set. After the sets have been modified/rotated, the second fold begins using the same training method explained earlier. This process is completed for the third, fourth, fifth, and sixth folds, where the standard AISs and mAISs were a total of 30 times for each fold, resulting in a total of 180 runs.

Table 2 presents the performances of the nine classification methods on the 30-28 dataset. In Table 2, the first column represents the methods, the second column represents the accuracies associated with each method. For GEFeS, the top number represents the best performance of the 30 runs while the number in parenthesis represents the average performance of the 30 runs. For the standard AISs and mAISs, the top number represents the average of the best performance on each of the six folds and the number in parenthesis represents the average performance of the 180 runs.

In Table 2, the best performances given a particular statistical measure are italicized. The best performance for a statistical measure was determined by first looking at the best individual performance of a method for a particular statistical measure. In the event of a tie between two methods, the average performances of the methods were used to break the tie. In terms of selecting an overall best-performing method, we used accuracy. In Table 2, one can see that all of the mAISs outperform GEFeS and all of the standard AISs in terms of accuracy with the $\text{mAIS}_{+\text{SDM},+\text{GEFeS}}$ having the best accuracy at 93.33%. One can see also that GEFeS alone outperforms all of the standard AISs in terms of accuracy as well. In terms of TPR and FNR, the $\text{AIS}_{+\text{SDM},+\text{GEFeS}}$ has the best performance, and for FPR and TNR, GEFeS has the best performance.

The mAIS was able to outperform GEFeS only and the standard AIS variants because of its ability to explicitly generate detector sets which identify with a single class. Unlike the standard AIS, where only one class is considered during training, the mAIS is able to capitalize on training using both "self" and "non-self" and create

Table 2 The results of comparing the nine methods on the 30-28 dataset

Method	Accuracy	TPR	TNR	FPR	FNR
GEFeS	70% (53.67)	40% (14.00%)	*100.0% (93.33%)*	*0.00% (6.67%)*	60.00% (86.00%)
$AIS_{-SDM,-GEFeS}$	53.33% (35.39%)	80.00% (49.00%)	26.67% (21.78%)	73.33% (78.22%)	20.00% (51.00%)
$AIS_{+SDM,-GEFeS}$	50.00% (33.50%)	80.00% (46.11%)	20.00% (20.89%)	80.00% (79.11%)	20.00% (53.89%)
$AIS_{-SDM,+GEFeS}$	51.67% (32.11%)	83.33% (42.44%)	20.00% (21.78%)	80.00% (78.22%)	16.67% (57.56%)
$AIS_{+SDM,+GEFeS}$	56.67% (32.82%)	*93.33%* (43.56%)	20.00% (22.07%)	80.00% (77.93%)	*6.67%* (56.44%)
$mAIS_{-SDM,-GEFeS}$	88.33% (66.72%)	76.67% (46.00%)	100.00% (87.44%)	0.00% (12.56%)	23.33% (54.00%)
$mAIS_{+SDM,-GEFeS}$	86.67% (64.39%)	73.33% (44.22%)	100.00% (84.56%)	0.00% (15.44%)	26.67% (55.78%)
$mAIS_{-SDM,+GEFeS}$	86.67% (60.72%)	73.33% (41.11%)	100.00% (80.33%)	0.00% (19.67%)	26.67% (58.89%)
$mAIS_{+SDM,+GEFeS}$	*93.33%* (60.29%)	86.67% (42.86%)	100.00% (77.71%)	0.00% (22.29%)	13.33% (57.14%)

The best performances given a particular statistical measure are italicized corresponding detector sets. This creates a more distinct difference between "self" and "non-self" within the hypothesis space. Coupled with GEFeS, which adds more distinction, and SDM, which fills in potential holes, the mAIS was able to increase accuracy, TPR and TNR, while also decreasing FPR and FNR.

7 Conclusions

In this paper, we introduce the Multiple-Detector Set Artificial Immune System (mAIS). The mAIS is a modified version of the standard Artificial Immune System (AIS) introduced by Hofmeyr [16]. Instead of producing a single-detector set for identifying "non-self", the mAIS generates two detector sets, one for identifying "self" and one for identifying "non-self." The two detector sets are used in a proportion-based classification technique. The classification technique uses an "exclusive or" model based on the proportion of fired detectors per set. The mAIS approach produced a true positive rate of 88.33%.

We extended the mAIS further with two variants: Genetic and Evolutionary Feature Selection (GEFeS), to eliminate redundant and irrelevant features, and a variant of the negative selection algorithm called Split Detector Method (SDM). Both of these modifications improved accuracy while reducing the FNR; however, the FNR still remained relatively high. Our best results occurred when both GEFeS and SDM were applied with the mAIS simultaneously. This produced an accuracy of 93.33% with a true positive rate of 86.67% and a false positive rate of 0.00%.

The results show the importance of using a self detector set in reducing the FPR and FNR. This was observed by [20, 21]. In essence, the self detector set guards against potential autoimmune responses in that its detectors represent instances belonging to "unknown" self. This is similar to the "non-self" detectors representing instances of "unknown" non-self.

The results show that the AISs seem to have relatively high FNRs. One solution to this can be found in [18, 26] where Hou and Dozier use a genetic algorithm to search for "holes" within an AIS. These holes are then patched with additional non-self detectors. This requires potentially a human in the loop or at least another system that has a better understanding of "unknown" non-self. Equipping AISs with the ability to work with humans in an effort to reduce the FNR is critical in that "self" as well as "non-self" can change over time.

In the future, we plan to pursue research with a human in the loop AIS and apply this to the mAIS. With a human in the loop, this would allow us to incorporate the full detector set generation scenario detailed in Hofmeyr [17], where a detector set consists of immature, mature, and memory detector. In a dynamic environment, the human in loop would determine the promotion of a detector from mature to memory. We could also explore implementing lifespans to corresponding detectors, where lifespan $t_{\text{immature}} < t_{\text{mature}} < t_{\text{memory}}$. This would allow for a progressively diverse set of detectors as the model evolves. Currently, our implementation only allows for immature detectors with unlimited lifespan in a static environment. Immature detectors are generally less accurate at differentiating between "self" and "non-self" than mature and memory detectors. With the inclusion of mature and possibly memory detectors, we should expect an increase in classification accuracy as a result.

We also plan to continue increasing our dataset size. Our goal is to have a dataset which consists of 500 malicious app samples and 500 benign app samples. With a larger dataset size, we will be able to test the scalability and portability of the mAIS. A larger dataset size not only increases the number of mobile app samples, but it could also increase the dimensionality of our problem and each app feature vector. This could occur because the number of features in each feature vector is determined by the existence of a particular feature across the entire dataset, as explained in Section 4. Empirical evidence is needed to determine if an increase in dataset will significantly increase the computational complexity of the mAIS. To accomplish this goal, we will continue to run data collection and increase the computation processing power available to us, which should aid in our ability to gather larger mobile apps. With this increased runtime and potential to analyze larger apps, we also plan to reorganize the mobile apps into their appropriate families, to determine if there is a significant difference in detection.

8 Endnotes

[1]A lowercase "m" was used in the spelling of mAIS to signify the similarity the mAIS shares with the Standard Artificial Immune System (AIS).

Acknowledgements
This research is based upon work supported by the United States Government including the National Science Foundation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon.

Authors' contributions
JB carried out the data collection, feature vector creation, experimentation setup, experiment results, and drafted the manuscript. MA provided oversight for data collection and experimentation. MA also participated in manuscript editing as well as malware detection field analysis. GD participated in the manuscript drafting. All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Received: 20 October 2016 Accepted: 6 March 2017
Published online: 27 March 2017

References

1. S Radicati (Ed.), Mobile Statistics Report, 2016-2020 (2016). Retrieved March 24, 2016, from http://www.radicati.com/wp/wp-content/uploads/2016/01/Mobile-Growth-Forecast-2016-2020-Executive-Summary.pdf
2. IDC: Smartphone OS Market Share. (n.d.). Retrieved March 24, 2016, from http://www.idc.com/prodserv/smartphone-os-market-share.jsp
3. G Kelly, in *Report: 97% Of Mobile Malware Is On Android*. This Is The Easy Way You Stay Safe, 2014. Retrieved June 19, 2015, from http://www.forbes.com/sites/gordonkelly/2014/03/24/report-97-of-mobile-malware-is-on-android-this-is-the-easy-way-you-stay-safe/
4. Number of the week: List of malicious Android apps hits 10 million. Retrieved June 19, 2015, from https://blog.kaspersky.com/number-of-the-week-10-million-malicious-android-apps/3683/. (2014). Accessed March 15, 2017
5. V Svajcer, *Sophos Mobile Security Threat Report, SophosLabs* (2014). Retrieved March 15, 2017, from https://www.sophos.com/en-us/medialibrary/PDFs/other/sophos-mobile-security-threat-report.pdf
6. M Grace, Y Zhou, Q Zhang, S Zou, X Jiang, in *Riskranker: scalable and accurate zero-day android malware detection*. Proceedings of the 10th international conference on Mobile systems, apps, and services (ACM, New York, 2012), pp. 281–294
7. Avast Mobile Security. [Online]. https://www.avast.com/en-us/free-mobile-security
8. McAfee Security. [Online]. https://www.mcafeemobilesecurity.com/
9. L Batyuk, M Herpich, SA Camtepe, K Raddatz, AD Schmidt, S Albayrak, Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android apps. In Malicious and Unwanted Software (MALWARE), 2011 6th International Conference, (IEEE, Washington, 2011), pp. 66–72
10. A Shabtai, Y Elovici, Applying behavioral detection on android-based devices. In Mobile Wireless Middleware, Operating Systems, and Apps, (Springer, Berlin Heidelberg, 2010), pp. 235–249
11. DJ Wu, CH Mao, TE Wei, HM Lee, KP Wu, Droidmat: Android malware detection through manifest and api calls tracing. In Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference, (IEEE, Tokyo, 2012), pp. 62–69
12. Z Yuan, Y Lu, Z Wang, Y Xue, Droid-Sec: deep learning in android malware detection, in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, pp. 371–372. ACM
13. D Dasgupta, "An Overview of Artificial Immune Systems and Their Applications", *Artificial Immune Systems and Their Applications*, D Dasgupta, ed., (Springer, Berlin Heidelberg, 1999), pp. 3–21
14. D Dasgupta, F Gonzalez, An immunity-based technique to characterize intrusions in computer networks. IEEE Trans Evol Comput **6**(3), 281–291 (2002). IEEE Press
15. J Timmis, T Knight, LN de Castro, E Hart, An overview of artificial immune systems, in *Computation in Cells and Tissues: Perspectives and Tools for Thought*, ed. by R Paton, H Bolouri, M Holcombe, JH Parish, R Tateson. Natural Computation Series, 2004
16. S Hofmeyr, in *An Immunological Model of Distributed Detection and Its Application to Computer Security*, Ph.D. Dissertation, Department of Computer Science (The University of New Mexico, Albuquerque, 1999)
17. S Hofmeyr, S Forrest, in *"Immunity by Design: An Artificial Immune System"*. The Proceedings of the 1999 Genetic and Evolutionary Computation Conference (GECCO-1999) (Morgan-Kaufmann, San Francisco, 1999)
18. H Hou, G Dozier, in *Immunity-based intrusion detection system design, vulnerability analysis, and GENERTIA's genetic arms race*, Proceedings of the ACM Symposium on Applied Computing (2005), p. 952–956
19. H Hou, GENERTIA: A System for Vulnerability Analysis, Design and Redesign of Immunity-Based Anomaly Detection Systems, Ph.D. Dissertation, Computer Science & Software Engineering Department, (Auburn University, Auburn, 2006)
20. I Idris, Model and Algorithm in Artificial Immune System for Spam Detection, International Journal of Artificial Intelligence & Applications (IJAIA). 3(1) (2012), pp. 83–94
21. I Idris, SM Abdulhamid, An Improved AIS Based E-mail Classification Technique for Spam Detection. *arXiv preprint arXiv:1402.1242*. (2014)
22. S Ramakrishnan, S Srinivasan, Intelligent agent based artificial immune system for computer security—a review. Artif Intell Rev **32**(1-4), 13–43 (2009)
23. M Zhao, T Zhang, J Wang, Z Yuan, A smartphone malware detection framework based on artificial immunology. J Networks **8**(2), 469–476 (2013)
24. J Zhu, *Use of an Immune Model to Improve Intrusion Detection on Dynamic Broadcast Local Area Networks* (Masters Thesis, Department of Computer Science & Software Engineering, Auburn University, Auburn, 2002)
25. S Arzt, S Rasthofer, C Fritz, E Bodden, A Bartel, J Klein, P McDaniel, in *Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps*. ACM SIGPLAN Notices, vol. 49, no. 6 (ACM, New York, 2014), pp. 259–269
26. Y Zhou, X Jiang, *Android malware genome project* (2012). Available at http://www.malgenomeproject.org
27. K Casey, A Garrett, J Gay, L Montgomery, G Dozier, An evolutionary approach for achieving scalability with general regression neural networks. Nat Comput **8**(1), 133–148 (2009)