SAND2015-0841C

Attacking DBSCAN for Fun and Profit, Supplementary Material

Jonathan Crussell Philip Kegelmeyer

A-1 Data Mine Generation

In Section 5.4, we described generating data mines when the apps have approximately equal number of features and when $MinPts = 2$. In this Section, we describe how to remove those assumptions and derive a new equation for the number of mines required to bridge the gap between two clusters based on T and $MinPts$.

A-1.1 Dissimilar Sizes AnDarwin detects cloned apps on a large scale. It represents each app as a set of features as follows: 1) it converts Android's DEX bytecode to Java bytecode, 2) it uses WALA [1] to compute program dependency graphs (PDGs) for methods found within the app, 3) it encodes the PDGs as *semantic vectors* that it clusters to form features and, finally, 4) it represents each app as a set of binary features where a feature, f, is set if an app contains a PDG whose semantic vector clusters into cluster f. This may lead to some apps having many more features than others. During the upper bound analysis, we made the assumption that the two target points, p_0 and p_n, had roughly the same number of features. However, if the points had drastically different numbers of features, the adversary would have to first generate an extra series of points to scale the smaller point's feature set to a similar size as the larger point. Then, she may proceed using the same approach outlined in Section 5.4. The process of scaling up the smaller point requires creating points in the following manner:

$$p_i = p_{i-1} + x_i p_n \tag{A-1.1}$$

Where x_i represents a portion of the features in p_n. In this case, x_i must be selected based on T and the relative sizes of p_n and p_{i-1}:

$$J(p_{i-1}, p_i) = \frac{p_{i-1} \cap (p_{i-1} + x_i p_n)}{p_{i-1} \cup (p_{i-1} + x_i p_n)} \tag{A-1.2}$$

$$= \frac{|p_{i-1}|}{|p_{i-1}| + |x_i p_n|} = T \tag{A-1.3}$$

$$\Rightarrow x_i = \frac{(1-T)|p_{i-1}|}{T|p_n|} \tag{A-1.4}$$

Unlike Equation 5.7, x_i is dependent on the size of p_{i-1}. This means that as we scale the original (and smaller) point, p_0, we can add larger and larger portions of p_n to the next mine. In fact, the total number of mines to scale p_0 to the same size as p_n is logarithmic in the size of $\frac{p_n}{p_0}$. The base of the logarithm is $\frac{1}{T}$ as the portion of p_n that can be used for p_i is inversely dependent on the similarity threshold.

A-1.2 $MinPts >= 3$ During the analysis, we made the assumption that $MinPts = 2$. This effectively made every point a core point which simplified the algorithm to generate data mines. In fact, the original algorithm works for $MinPts$ values 2 and 3 (see Figure 1) since the point, p_i, and its predecessor and successor, p_{i-1} and p_{i+1}, are in the T-neighborhood of p_i.

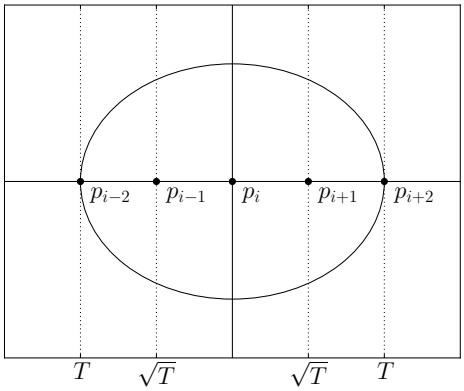


Figure A-1: Geometry for a five-point chain. This chain shows how to generate datamines when $MinPts = 5$.

Figure A-1 depicts a geometry that the attacker can use to make each p_i a core point when $MinPts$ is 5. For odd values of $MinPts$, the attacker can generate similar geometries with $k = \frac{MinPts-1}{2}$ equally spaced points on both sides of p_i. The distance between these points should be such that:

$$\forall i \in [k, n) : J(p_{i-k}, p_i) = T \tag{A-1.5}$$

For the Jaccard distance, this means that:

$$\text{(A-1.6)} \qquad \forall i \in [0, n) : \; J(p_i, p_{i+1}) = \sqrt[\frac{MinPts-1}{2}]{T}$$

Then, the geometry of p_i and its neighbors, p_{i-1} and p_{i+1}, is the same as Figure 1 except that T has been replaced with $T' = \sqrt[\frac{MinPts-1}{2}]{T}$. Therefore, we can substitute T' into Equation 5.9 to determine the number of mines required to merge two clusters as a parameter of both T and $MinPts$:

$$\text{(A-1.7)} \quad UBAC(T, MinPts) = \frac{1 + T'}{1 - T'} - 1$$

$$\text{(A-1.8)} \quad = \frac{1 + \sqrt[\frac{MinPts-1}{2}]{T}}{1 - \sqrt[\frac{MinPts-1}{2}]{T}} - 1$$

There is a caveat to the above approach: it requires that both p_0 and p_n be core points in their respective clustering. This can be accounted for by adding enough data mines to the T-neighbors of p_0 and p_n such that they are now core points. This requires at most $2 * MinPts$ more data mines be added. This additive constant is left out for simplicity.

A-2 Evaluation

In this section, we present additional results from our evaluation. Specifically, we present the degradation plots for two of the performance metrics, an inadvertent merge that occurred when performing our attacks, and, finally, additional results from our remediation experiments.

A-2.1 Cluster Merging In Figure A-2, we plot the values of two of the four clustering performance metrics described in Section 5.2. Comparing these plots to Figure 2 we can see that Adjusted Rand Index is much more sensitive to our attacks than Homogeneity or Adjusted Mutual Info.

A-2.2 Inadvertent Merges In our implementation of the attacks, we split the stages of merge ordering and data mine generation into two separate phases. We compute the pairwise similarity matrix for the points and the clustering before running the merge ordering algorithms but we do not update the matrix or rerun the clustering after each data mine is added. This could lead to *inadvertent merges* where, when merging two clusters, some of our data mines cause a third or more cluster to be merged in with the two that we intended to merge and could cause us to "overspend" as an attacker.

Rather than merge the two stages of the attack, we instead post-process a pairwise similarity matrix that contains all the original points and the data mines. First, we record the number of clusters found when clustering the original points. Then, we add a set of data mines forming a bridge to the matrix, recompute the clustering, and record the number of clusters. We repeat this process for each of the bridges. Each bridge should merge exactly two clusters so the number of clusters should decrease by one for each bridge added. If the number of clusters decreases by more than one, an inadvertent merge has occurred.

We ran the methodology described above on all the merge algorithms and found no inadvertent merges. The chosen feature space for Android apps is very sparse so these merges are unlikely to occur. For DBSCAN-based tools in a more dense feature space, inadvertent merges may help reduce an attacker's cost.

A-2.3 Remediation In Figure A-3, we plot the plagiarism detection accuracy for the remediated dataset when we train a classifier with varied levels presumed tampering and test it on varied levels of actual tampering. These heatmaps show that as long as the defender presumes a lower level of tampering than there actually is, remediation is effective. Surprisingly, training a classifier with as few as five presumed merges leads to near-perfect recovery.

In Figure A-4 and Figure A-5, we show similar remediation results for an additional merge algorithm: Decreasing Original Size. As described in Section 5.3, this algorithm merges clusters based on the number of original apps found in the cluster, in decreasing order. From Figure A-4, we can see that the tampered performance drops off much more quickly than it did in Figure 3. Additionally, we can see that the remediation was less stable at higher numbers of merges.

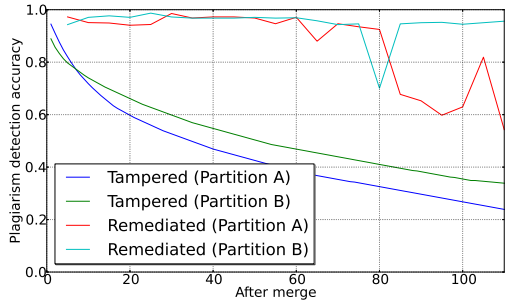
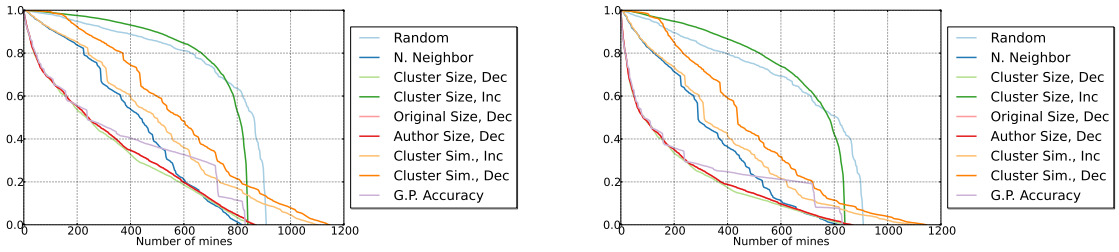


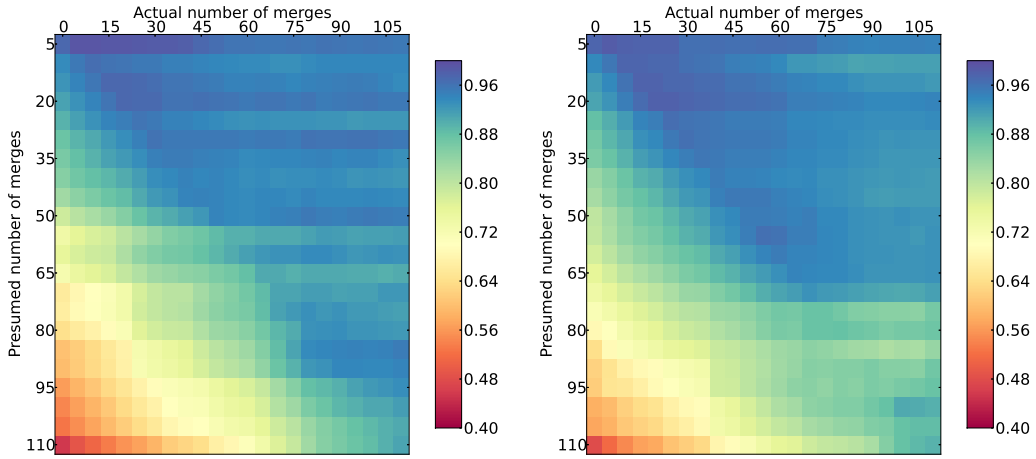
Figure A-4: The plagiarism detection accuracy with and without remediation for partitions `A` and `B`. For the outlier remediation curve, the presumed number of merges in the training partition matched the actual number of merges in the testing partition. Clusters were ordered using the Decreasing Original Size merge algorithm.



(a) Homogeneity

(b) Adjusted Mutual Info

Figure A-2: Cluster degradation plots. These show how two of the four clustering performance metrics degrade as a function of the number of data mines the attacker has injected into the dataset. From an attacker's perspective, algorithms with less area under the curve are better since they drop the clustering performance quicker.



(a) Using partition `A` for testing.

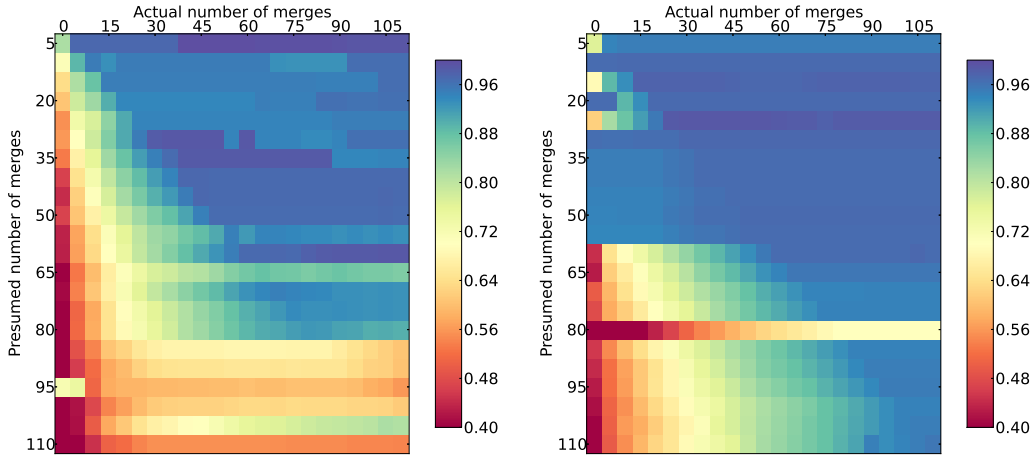
(b) Using partition `B` for testing.

Figure A-3: Plagiarism detection accuracy for the various levels of presumed and actual tampering for the two partition. Clusters were ordered using the Random merge algorithm.

Another interesting experiment to pursue in future work is to train a classifier assuming one attacker merge algorithm and test on another. This is a more realistic scenario as we are unlikely to know the attacker's strategy ahead of time.

References

[1] IBM T.J. Watson Research Center. T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net.



(a) Using partition `A` for testing.

(b) Using partition `B` for testing.

Figure A-5: Plagiarism detection accuracy for the various levels of presumed and actual tampering for the two partition. Clusters were ordered using the Decreasing Original Size merge algorithm.