Alligator

Ludovic Apvrille, Axelle Apvrille

October 26, 2016

1 Introduction

Alligator [AA13, AA14] stands for **AnaLyzing maLware wIth partitioninG and probAbiliTy-based algORithms**[1]. Basically, it is a probabilistic tool to help partition elements in two different categories.

Conceptually, this means that, given two distinct sets of elements R and M, alligator is able to decide whether another element e looks like elements of R or of M.

Each set of element is called a *cluster*. Clusters R and M are characterized by a common set of properties, which qualify for instance the nature or the behaviour of elements. R and M have no intersection.

In our case, clusters R and M happen to be regular (clean) and malicious samples, so alligator helps us decide whether a given sample is likely to be malicious or not. In reality, however, alligator does not know anything about malware, it is purely a probabilitic tool. Moreover, note that alligator cannot be guaranteed to make the right decision: it is only a helper tool for malware analysts, as [Coh89] proved the undecidability of computer viruses.

- **Can alligator be used for something else than computer viruses? Yes**. Alligator does not know anything about malware, viruses. Alligator can be used to partition elements between any distinct set of elements. For instance, if we divide the world between smart people and lame ones, alligator could help us decide whether person p is smart or lame (provided there is no intersection between the smart and the lame cluster).
- **Does alligator have access to mobile virus samples? No**. Alligator only reads as input files of properties which qualify a given (clean or malicious) sample. The values of those properties are typically booleans, strings or numeric values. So, basically, alligator just analyzes a bunch of data and tries to make sense out of it, in terms of statistics.
- **Does alligator understand this or that property? No**. Alligator does not have this level of understanding. It is of no importance to alligator if the 3rd property is a file size or a number of classes. Alligator only understands columns and expects the 3rd property to have the same type (e.g boolean, string, ...) across all clusters.

2 Theory

2.1 Alligator methodology

To decide whether an element e is probably part of set R or M (also called *clusters*), alligator computes various probabilistic formulas. The results are turned into two scores: a score of ressemblance for R and one for M. Alligator displays the scores, and obviously the highest score bares the strongest ressemblance.

However, like everybody else on Earth, alligator benefits from learning. Without any training, alligator will probably make wrong decisions are classify a given element e in the wrong cluster. With good training, alligator can significantly improve its decisions. So, usually, alligator's methodology consists in 2 steps:

[1] We also happen to be fans of plush crocodiles ;)

1. **Learning**. In this step, alligator learns the expected nature and behaviour of elements of set R and M. The teacher (i.e end-user) feeds to the alligator learning program the values of properties of elements of R and of M. Each element is described against the same set of properties. For those properties, to partition sets, alligator uses probabilistic computations: deviation, proximity, epsilon clusters, svm, etc. The learning program applies rules and weights to each computation in a manner than differentiates at best elements of cluster R from those of cluster M. It tries multiple combinations of rules and weights, according to different *algorithms* (e.g random values within a range). Implemented algorithms are detailed at 3.4. When alligator has found the best possible configuration, it generates a configuration file, called a *script*, which explains how to partition samples at best. This stage can be (moderately) long, depending on the quantity of data to process. Furthermore, the end-user is likely to try several algorithms and options to get the best partitioning of clusters.

2. **Partitioning**. Using the generated script, alligator stands better chances of correctly partitioning samples to analyze. For each sample to partition, it computes ressemblance scores for cluster R and M, and displays the results. This stage is quite fast to compute.

2.2 Definitions

Definition 1 ***Property***
A Property p is defined as $p = (l)$ where:

- *l is a label*

Example: the label of property number 6 could be "send_sms_permission".

Definition 2 ***Sample***
A Sample s is defined as $s = (l, v_p, type)$ where:

- *l is a label*
- *v_p is a function v_p: $v : p \mapsto \mathbb{R}_{\geq 0}$. v_p returns the value of the sample for a given Property p.*
- *$type \in \{R, M, unknown\}$.*

So, for a given sample, the *value* of its property p is $v_p(s)$, and the *type* of sample s is $type(s)$.
Example: the label of a sample is "./Activator.apk". Its type is "M". Its value for property ♯6 is $v_6(s) = 1$.

Definition 3 ***Cluster***
A Cluster C is defined as $C = (l, P, S)$ where:

- *l l is a label*
- *P is a set of properties, also denoted as C_P. Property number i of $\mathbb{P}$ is denoted p_i*
- *S is a set of samples, also denoted as C_S*

Definition 4 ***Alligator Algorithm***
An Alligator Algorithm AA is defined as $AA = (algoName, algo, computeScore)$ where:

- *algoName is a label among:*
  algoName $\in L = \{$StandardDeviation,
  ProbabilityDifference, probabilityFactor,
  Proximity, ProximityRatio, Degressive Proximity, ProximityWithLimitedProperties,
  Correlation, EpsilonCluster, SVM, AddValueToScore$\}$.

- *algo is a clustering function: algo* : $a \in$ *algoName*, $g \in G, (R, M)$ *two clusters* $\mapsto \mathbb{R}^2$. *This function is denoted* AA_{algo}.*The first value returned by* AA_{algo} *is denoted as* AA_r *and the second one* AA_m*, i.e.,* $(AA_r, AA_m) = AA_{algo}$(*algoName*, g, R, M)
- *computeScore is a function computeScore* : $\mathbb{R}, C \mapsto \mathbb{R}$*, also denoted* $AA_{compScore}$. *computeScore converts the result of a clustering algorithm into a risk score, and also applies a weight to the algorithm.*

Basically, a clustering algorithm applies to a sample g of a guess cluster G. It also takes as argument two clusters (R and M) and returns two real values, the "raw" scores AA_r and AA_m for each cluster R and M. The "final" scores AA_{rf} and AA_{mf} can then be computed from AA_r and AA_m as follows: $AA_{rf} =$ *computeScore*(r, R) and $AA_{mf} =$ *computeScore*(m, M). As we said previously, the function *computeScore* is to be discovered automatically by the learning phase of Alligator.

Definition 5 ***Alligator System***
An Alligator System AS is defined as $AS = (R, M, G, A, score)$ *where:*

- R *is a cluster such that* $\forall s \in R, type(s) = R$
- M *is a cluster such that* $\forall s \in M, type(s) = M$
- G *is a cluster such that* $\forall s \in G, type(s) = unknown$ *(guess cluster)*
- A *is a set of Alligator Algorithms. It is denoted* AS_A.
- $score$ *is a function defined as follows:* $score : g \in G, C \in \{R, M\} \mapsto \mathbb{R}_{\geq 0}$. *In other terms,* $score(g, R)$ *returns the score of a sample* g *of* G *for cluster* R*, and* $score(g, M)$ *returns the score of a sample* g *for cluster* M.

Also, in an ***Alligator System****, we always have:*

$$R_P = M_P = G_P \tag{1}$$

2.3 Computing scores

For a given Alligator System AS, the goal is to determine the type of each sample of G such that $\forall s \in G, type(s) = R \vee M$, i.e., the type is either *Regular* or *Malware*.
To do so, $score(s, R)$ and $score(s, M)$ are progressively computed using different Alligator Algorithms (AA), explained hereafter: *StandardDeviation*, *ProbabilityDifference*, *probabilityFactor*, *Proximity*, *ProximityWithLimitedProperties*, *Correlation*, *EpsilonCluster*. Then, according to the scores, the guess sample is either classified as malware, or as regular:

$(score(s, R) > score(s, M) \implies type(s) = R) \wedge (score(s, M) \geq score(s, R) \implies type(s) = M)$.

For each algorithm AA, and for each sample g of G, we compute the value given by AA_{algo} using clusters R and M, respectively. So, for a given sample $g \in G$, we have in AS, for cluster R:
$score(g, R) = score(g, R) + AA_{compScore}(AA_{algo}($*algoName*$, g, R, M)_r), R)$
Similarly, for Cluster M, we have:
$score(g, M) = score(g, M) + AA_{compScore}(AA_{algo}($*algoName*$, g, R, M)_m), M)$

Note: we take the assumption that for *each* property there exists at least one sample s in each set R and M so that this sample has not an $invalid$ value for that property:
$\forall p \in P, \exists s \in S$ such that $v_p(s) \neq$ invalid $\wedge \exists s \in T$ such that $v_p(s) \neq$ invalid

By applying this to all algorithms $a \in A$, we can compute for each $s \in G$ its score with regards to clusters R and M. Various alligator algorithms are now further detailed.

The number of element of a set E is denoted by $|E|$.

2.4 Standard deviation

The standard deviation is a well known metric for computing a dispersion with regards to an average value. In the case of clustering with R and M, the standard deviation intends to measure the distance between a given sample $g \in G$ and the average of cluster R and M for each property. We obvisouly assume that the smallest this distance is with a given cluster center, the better is the chance to be part of that cluster.
More formally, the standard deviation can be defined as follows:
For a given Alligator System AS, the deviation score of a sample s with regards to a cluster C is denoted $dev(s)_C$. We also denote by $\overline{C_{p_i}}$ the mean value of all samples of C for property p_i with $p_i \in C_P$. We also denote by Q the largest possible set of properties of C_P such that $\forall p \in Q v_p(s) \neq$ invalid.

$$dev(s)_C = \sqrt{\frac{1}{|C_Q|}\sum_{i=1}^{|C_Q|}(v_p(s) - \overline{C_{p_i}})^2}$$

2.5 Deviation with weight

This deviation is quite similar to the previous one part that a weight can be applied to properties. The computation of the mean is modified as follows:

Thus, the deviation formulae is as follows, with $w(p_i)$ being the weight of property p_i:

$$dev_w(s)_C = \sqrt{\frac{1}{\sum_{i=1}^{|C_p|} w(p_i)}\sum_{i=1}^{|C_Q|} w(p_i) * (v_p(s) - \overline{C_{p_i}})^2}$$

2.6 Probability difference

The score computed by this algorithm is based on the percentage of "typical" values that are respected in R and M. "Typical" is defined with a probability difference *diff*.
For example, let's assume that the probability of property ♯6: "send_SMS" to be equal to "1" is 0.8 in cluster M and equal to 0.2 in cluster R, and the probability difference *diff* $= 0.5$: We say that "1" is a typical value of cluster M for property "send_SMS" because $0.8 > 0.2 + 0.5$. Thus, if for $g \in G, v_6(g) = 1$ then g "respects" one typical value of of M.
More formally, the probability difference is as follows.
This computation is made according to a given probability difference *diff* such that $0 <$ *diff* < 1.
We denote by $proba(v, p, C)$ the function that returns the probability for a sample of cluster C of having the value $v \in \mathbb{R}$ for the property p of C_P.
For each Cluster R and M, we first compute the set of properties ***DIFP***$_R \subset R_P$ et ***DIFP***$_M \subset M_P$ as follows.

DIFP$_R$: $\forall p \in$ ***DIFP***$_R, \exists r \in \mathbb{R} \mid proba(r, p, R) - proba(r, p, M) \geq$ *diff*

Similarly,

DIFP$_M$: $\forall p \in$ ***DIFP***$_M, \exists r \in \mathbb{R} \mid proba(r, p, M) - proba(r, p, R) \geq$ *diff*

Then, for a given sample g of cluster G of an Alligator System AS, the score of probability difference for cluster R is computed as follows:

probaDiff$R(g) = \frac{|DIFP_g|}{|DIFP_R|}$

with ***DIFP***$_g = \{p \mid proba(v_p(g), p, R) - proba(v_p(g), p, M) \geq$ *diff*$\}$.

It can be similarly computed for cluster M.

Finally, for each sample g of G, we compute the number n_R and n_M of property values for which the probability difference holds in R and M, respectively. Then, we compare n_R and n_M with the maximum number of possible probability differences in each cluster R and M: the more probability differences are respected in a cluster, the more chance the sample has to be an element of that cluster.

2.7 Probability factor

The score for probability factor is basically computed as for the probability difference, apart from the identification of "typical" values. Indeed, typical values are identified with a given probability factor *fact*, $where fact \geq 0$: The value v of a property p is typical for cluster C_1 with regards to cluster C_2 if and only if: $proba(v, p, C_1) \geq proba(v, p, C_2) * fact$.

$FACTP : \forall p \in FACTP, \exists r \in \mathbb{R} \mid proba(r, p, R) \geq proba(r, p, M) * fact$

The probability factor score is computed as follows:

$probaFactR(g) = \frac{|P_g|}{|FACTP_R|}$
with $P_g = \{p \mid proba(v_p(g), p, R) \geq proba(v_p(g), p, M) * fact\}$

2.8 Proximity

The proximity algorithm is based on the identification of the samples of R and M that are the closest of $g \in G$ with regards to a distance relation *dist()*.

The distance between two samples $g \in G$ and $s \in R \cup M$ is computed as follows:
$dist(g, s) = \sqrt{\sum_{i=1}^{i=|P|}(v_{p_i}(g)^2 - v_{p_i}(s)^2)}$

The algorithm takes as input the number n of closest neighbours that shall be considered. Then, the score for R (resp. M) corresponds to the percentage of elements of R (resp. M) that are in the set of closest neighbours. Thus, contrary to the standard deviation that targets the distance with the centers of the two R and M clusters, the proximity cares only with the closest elements of each clusters R and M. We assume that the more elements of a given cluster are close to g, the more chance has g to belong to this cluster.
More formally, $proximity(n, g)$ computes for $g \in G$ the set of samples $PROX$ that contains samples of R and M which are the closest of g, with $|PROX| = n$, where $n \in \mathbb{R}$, and assuming that $|R| + |M| > n$.

Let $minSet(E, n)$ be a function that returns a set F such that $|F| = n$ and $\forall f \in F, \forall e \in E, f \leq e$, i.e., $minSet(E, n)$ extracts the n minimal values of E. It is assumed that $|E| \geq n$.

To compute the score for cluster R of sample $g \in G$, we first define a set built using the $minSet$ function:
$PROX = minSet(\{dist(g, s)\}_{s \in S \cup T}, n)$.

Finally, the proximity score for R and M is respectively:
$proximityR(g) = \frac{|\{s \in R \bigcap PROX\}|}{n}$
and
$proximityM(g) = \frac{|\{s \in M \bigcap PROX\}|}{n}$

2.9 Proximity with limited properties

The proximity algorithm computes a distance with a sample taking into account all properties. Thus, two samples which are very close - if not equal - on all properties, but very far with only one property, may be classified as very distant.

On the contrary, other samples quite close on most properties may be classify as close, even if most property values are different. We do think that "abnormal" properties - that is, property values which are the more distant - could be ignored so as to identify close samples with regards to most relevant properties i.e. with regards to most properties apart from the ones with the highest distance values.
Thus, proximity with limited property considers only the most n "relevant" properties for each distance computed between two samples $g \in G$ and $s \in R \cup M$. n most relevant properties means that we select n properties of P for which the distance between the value of g and s for properties $p \in P$ are the smallest. Mathematically speaking, this distance with limited properties can be defined as follows:

Using the *minSet* function, the distance with n limited properties between two samples $g \in G$ and $s \in R \cup M$ is computed as follows:
$F = minSet(\{v_{p_i}(g)^2 - v_{p_i}(s)^2\}_{0 \leq i \leq |P|}, n)$ and $dist(g, s) = \sqrt{\sum_{i=1}^{i=|F|} f_i}$.
Then $PROX$ and the scores for R and M can be computed as for the regular proximity.

2.10 Proximity ratio

The proximity algorithm 2.8 takes into account the category of n neighbours. This approach may be seens as unfair in the case where the cluster R and M contain a quite different number of samples: the neighborhood in such a case may contain a large number of samples of the largest of R and M, whathever g being more part of R or M.
To address that issue, the proxmity ratio applies a ratio $ra = \frac{|M|}{|R|}$, or the inverse of ra to the proximity score.
Finally, the proximity score for R and M is respectively:
proximityR$(g) = \frac{|\{s \in R \bigcap PROX\}|}{n} * \frac{|M|}{|R|}$
and
proximityM$(g) = \frac{|\{s \in M \bigcap PROX\}|}{n} * \frac{|R|}{|M|}$

2.11 Degressive proximity

The proximity algorithm 2.8 considers all neihbours the same. On the contrary, the "degressive proximity" algorithm gives a higher score to closest neighbours, and a lower one to further ones. Alligator relies on a parabolic degressive score, given as follows:
proximityR$(g) = \sum_{s \in R \bigcap PROX} (a * s_{ind(R)}^2 + c)$;
where:

- $a = \frac{1-n}{nb^2-1}$
- $c = n - a$
- $s_{ind(X)}$ represents the index of s in $X \bigcap PROX$

Similarly,
proximityM$(g) = \sum_{s \in M \bigcap PROX} (a * s_{ind(M)}^2 + c)$;

2.12 Correlation

The correlation algorithm intends to compute whether a given guess sample g respects value correlations between properties which are specific to a given cluster. We therefore target whether the sample g follows specific similarities of one of the two cluster R and M.

More precisely, let's assume that values of properties "send_SMS" and "Internet_Access" are correlated in M but not in R. If the values in g respect that correlation, we'll give g more chance to be in M than in R.

More generally, the correlation algorithm computes all exclusive correlations of R and M, and then computes the percentage of correlations of R and M that are respected by g.

The correlation algorithms first needs to identify which properties are correlated. Being correlated depends on a correlation threshold that we call *corr*: if the correlation value between those two properties p_1 and p_2 for a cluster C is higher than *corr*, the two properties are said correlated. This property is denoted as $validCorr_C(p_1, p_2)$

The correlation value between two properties p_1, p_2 of cluster R relies on the usual correlation formula :

$$corr_S(p_1, p_2) = \frac{\sum_{s \in S}(v_{p_1}(s_i) - \overline{S_{p_1}}) * (v_{p_2}(s_i) - \overline{S_{p_2}})}{\sqrt{|S| * (\sum_{s \in S} v_{p_1}(s_i)^2)) - ((\sum_{s \in S} v_{p_1}(s_i))^2)}} * \frac{1}{\sqrt{|S| * (\sum_{s \in S} v_{p_2}(s_i)^2)) - ((\sum_{s \in S} v_{p_2}(s_i))^2)}}$$

The correlation for two properties of cluster M can be computed using a similar formulae.

From $corr_R$, the set of correlated properties of R can be defined as follows:

$validCorr_R = \{(p_1, p_2) \in P \mid corr_S(p_1, p_2) > corr\}$

Similarly, for M:

$validCorr_M = \{(p_1, p_2) \in P \mid corr_M(p_1, p_2) > corr\}$

As explained before, we are interested in correlations exclusive to R, and correlations exclusive to M. We thus define the two following sets which represent the exclusive correlations of R and M, respectively:

$validCorr_{R-M} = \{(p_1, p_2) \in P\ mid(corr_S(p_1, p_2) > textitcorr) \wedge \neg((p_1, p_2) \in validCorr_M)\}$

and:

$validCorr_{M-R} = \{(p_1, p_2) \in P\ mid(corr_M(p_1, p_2) > textitcorr) \wedge \neg((p_1, p_2) \in validCorr_R)\}$

We can finally define the score of a sample g of G as follows. We take the two most frequent values for each couple of properties (p_1,p_2) in $validCorr_{S-M}$, and using a linear equation with those two values, we can thus deduce a and b such that $valueOf(p_2) = a * valueOf(p_1) + b$. a and b are denoted as $a_{(p_1,p_2)/R}$ and $b_{(p_1,p_2)/R}$, respectively

$$correlationR(g) = \frac{|\{(p_1,p_2) \in validCorr_R | v_{p_2}(g) = a_{(p_1,p_2)/R} * v_{p_1}(g) + b_{(p_1,p_2)/R}\}|}{|validCorr_{R-M}|}$$

Similarly,

$$correlationM(g) = \frac{|\{(p_1,p_2) \in validCorr_M | v_{p_2}(g) = a_{(p_1,p_2)/R} * v_{p_1}(g) + b_{(p_1,p_2)/M}\}|}{|validCorr_{M-R}|}$$

2.13 Epsilon cluster

Epsilon clustering relies on the notion of distance between samples. In epsilon clusters, samples are grouped according to a minimal distance ϵ. A sample s is added to a given epsilon cluster if and only if the epsilon cluster is empty or there exists one sample in this cluster so that the distance between that sample and s is less than ϵ. We can thus say than in an epsilon cluster, it is always possible to "go" from one sample to another one following a path of samples whose distance is less than ϵ (see Figure 1).

We think that epsilon clusters can be useful to create sub-groups of R and M for each g of the guess cluster, and then figure out the proportion of elements of R and M in that group. Thus, the more elements of R (resp. M) in the sub-group of g, the more chance has g to be an element of R (resp. M).

More formally, an epsilon cluster C_ϵ is a set of samples defined as follows:

$\forall c1, c2 \in C_\epsilon$ ϵ-*path*$(c1, c2)$ with ϵ-*path* being a function: ϵ-*path* : $C \times C \mapsto boolean$. The function means: ϵ-*path*$(c_1, c_n) \Leftrightarrow (\exists (c2, \ldots c_{n-1}) \in C \mid \forall (0 \leq i < n) : dist(c_i, c_i + 1) < \epsilon$

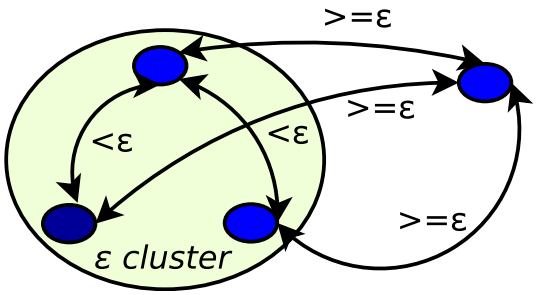

Figure 1: Epsilon cluster

The set $\mathbb{C}_\epsilon$ of epsilon clusters is defined as follows: $(\forall C \in \mathbb{C}_\epsilon, \forall c1, c2 \in C, \epsilon\text{-}path(c1, c2)) \wedge (\forall s \in S \bigcup T, \exists! C \mid s \in C) \wedge (c1 \in C1, c2 \in C2, C1 \neq C2 \Rightarrow dist(c1, c2) \geq \epsilon)$
Basically, all samples of R and M are grouped into sub-clusters called *epsilon clusters*. One sample belongs to exactly one Epsilon cluster. In an epsilon cluster, the distance between two samples $s1$ and s_n is either less than ϵ or there exists a path of samples $s2, \ldots, s_{n-1}$ so that the distance between samples of the path, and between $s1$ and $s2$, and between s_{n-1} and s_n is less than ϵ. At last, the distance between two samples of two different subclusters is necessarily equal or greater than ϵ.
The score for epsilon clusters is computed as follows. First, $\mathbb{C}_\epsilon$ is built from R and M. Then, for each sample g of G, the following set is built: $\mathbb{C}_\epsilon(g) = \{C \in \mathbb{C}_\epsilon \mid \exists c \in C, dist(g, c) < \epsilon\}$
Then, the score for R is the number of samples of s being in $\mathbb{C}_\epsilon(g)$:
$epsilonR(g) = |R \cap \mathbb{C}_\epsilon(g)|$.
Similary, the score for M is the following:
$epsilonM(g) = |M \cap \mathbb{C}_\epsilon(g)|$.

2.14 SVM

SVM (Support Vector Machine) is based on the computation of separation borders between elements. For this well-known algorith, Alligator relies on the jlibsvm implementation by David Soergel [Soe14], which is itself a fork of the libsm developed at Berkeley university [CL11].

Let's define Mod as the model computed by SVM: $Mod = computeModel(R, M)$. Then, that model can be used for classifying guess samples. For each guess sample, a score of -1 or 1 is returned: -1 means malware, 1 means regular. We denote by $classSVM(Mod, g)$ the value returned by the classification of g with regards to the model Mod.

Thus, the score of g for R with SVM, denoted as $SVM_R(g)$, is defined as follows:
$classSVM(Mod, g) == 1 \Rightarrow SVM_R(g) = 1 \wedge classSVM(Mod, g) == -1 \Rightarrow SVM_R(g) = 0$

Similarly, the score of g for M with SVM, denoted as $SVM_M(g)$, is:
$classSVM(Mod, g) == 1 \Rightarrow SVM_M(g) = 0 \wedge classSVM(Mod, g) == -1 \Rightarrow SVM_M(g) = 1$

2.15 AddValueToScore

This "algorithm" simply adds a given constant value to the overall score of each cluster given cluster.
Let us assume that the score returned by addValueToScore() is twofold: s_R and s_M for Regular and Malware, respectively.

Then, the score for R is :
$addValueToScoreR(g) = s_R$.
Similary, the score for M is the following:
$addValueToScoreM(g) = s_M$.

3 Usage and Reference

3.1 Cluster

A cluster provides values for a fixed set of properties of a given set of samples. Clusters are simple CSV files.

- Each column is separated by a comma
- There are three types of lines:
  - Sample lines. Those lines describe the values of properties of a given sample. There are usually many of these lines in a cluster.
  - Commented lines. They started with a hash #. They are ignored. They are useful to write comments in the cluster.
  - Property lines. A property line provides the name of each property of the cluster. Usually, there is only one property line per cluster and it is the first line (but this is not mandatory).
- All sample and property lines of the cluster must have the same number of columns. For instance, it is not possible to provide less properties for a given sample or only some property names but not all. Clusters with different number of columns would not be consistent and are strictly forbidden.
- 0, 1 or more lines may start with a !. Those lines are called property lines.
- If several lines start with !, the last one supersedes the previous ones.
- Property lines must have the same number of columns as any other line in the CSV.
- The first column of sample lines is a sample identifier string. For example, it may be a filename concatenated with a hash. This identifier being a string and not a number, it is not used in computations of Alligator, only to identify a sample.
- All other columns of sample lines must be positive doubles, whose values $v \in [0, 1]$. Note this usually means that values must be **normalized** to be sure to fit in that interval. The value may also be the reserver keyword 'unknown', which means the value is not set, not applicable or unknown.

Example:

```
# This is a sample cluster
!filename, size, red, blue, green, shade, bold
blah.txt, 0.12345, 0, 0, 1, unknown, 1
example.c, 0.7244, 1, 0, 1, 0, 1
...
```

3.2 Learning

```
java [-XmxSIZE] Learning -regular <CLEAN CLUSTER>
     -malware <MALICIOUS CLUSTER>
     -execute <SCRIPT>
     [-maxclustersize <int>]
     [-maxpercentageofinvalidproperties <int in [1..100]>]
     [-generatescript <FILENAME>]
     [-randomandonemultipass <RANDOM TIME> <NB OF PASSES>|-bruteforce|
      -randomeandtwomultipass <RANDOM TIME> <NB OF PASSES>|
      -bruteforce2|-random <RANDOM TIME>|-oneotheraverage|-twootheraverage]
     [-bestRegular|-bestMalware|-bestAll|-bestSetOnPercentageOnly]
     [-nocsvname]
```

- XmxSIZE: it might be necessary to adjust the usable memory by Java. This depends on the size of your clusters and the script. This is a Java option, not an option of Alligator.
- **regular**: provide here the properties of all elements of set R.
- **malware**: provide here the properties of all elements of set M.
- **generatescript**: when specified, the learning program generates the best script to use in the given file.
- **maxclustersize** $<$ `int` $>$: to read only the first n elements given in the cluster files.
- **maxpercentageofinvalidproperties** $<$ `int` $\in [1..100] >$: cluster elements having more than p percent of invalid properties are ignored.
- **Algorithms**: randomandonemultipass, bruteforce, random, oneotheraverage, twootheraverage are algorithms the learning program implements. See 3.4 for more details.
- **bestRegular**, **bestMalware**, **bestAll**, **bestSetOnPercentageOnly** are options to help choose the best values to partition samples. The default option is *bestAll*. The latter considers as equivalent the fact to better identify malware or regular samples. *BestRegular* means the learning program will choose the set of rules and algorithms which identifies the best regular samples. *BestMalware*, on the contrary, identifies the best malware. At last, by default, when comparing two learning results with the same recognition rate, Alligator selects the one which maximizes the standard deviation between regular and malware samples. If the option *bestSetOnPercentageOnly* is used, then the comparison between the learning results is made only according the percentage.
- **nocsvname**: when set, it means that the first column of the csv files does not contain the name of the sample. In that case, Alligator generates a name based on the line of the element in the csv file.

3.3 Partitioning / classification

3.3.1 Standalone partitioning

```
java Main [-debug] [-help] -regular <CLEAN CLUSTER>
       -malware <MALICIOUS CLUSTER>
       -toanalyze <GUESS CLUSTERS>
       -execute <SCRIPT>
       [-maxclustersize <int>]
       [-maxpercentageofinvalidproperties <int in [1..100]>]
       [-maxclustersize <SIZE>]
       [-report <REPORTFILE>]
         [-nocvsname]
```

Mandatory arguments are:

- **regular**: provide here the properties of all elements of set R.
- **malware**: provide here the properties of all elements of set M.
- **toanalyze**: provide here the properties of all elements for which we want alligator to decide/guess whether they are regular or malicious. It is perfectly acceptable to provide properties for a single element. You may provide several files
- **execute**: specify an alligator script as parameter. This script explains which weights and/or rules to apply computations. See 3.5 for details on the format of the script file.

Other options are:

- **debug**: verbose printing
- **help**: displays usage
- **report**: by default the output of alligator goes to stdout. Use this option to write it to a given file
- **maxclustersize**: will only consider the *size* first lines of each cluster.
- **maxpercentageofinvalidproperties**: cluster elements having more than p percent of invalid properties are ignored.
- **nocsvname**: when set, it means that the first column of the csv files does not contain the name of the sample. In that case, Alligator generates a name based on the line of the element in the csv file.

3.3.2 Learning and classification in a row

you may also start a clearning that will be immediately followed by a classification. The syntax is as follows, retaking the same arguments as the one defined in learning and classification:

```
java [-XmxSIZE] Learning -regular <CLEAN CLUSTER>
     -malware <MALICIOUS CLUSTER>
     -execute <SCRIPT>
     -toanalyze <GUESS CLUSTERS>
     [-maxclustersize <int>]
     [-maxpercentageofinvalidproperties <int in [1..100]>]
     [-generatescript <FILENAME>]
     [-randomandonemultipass <RANDOM TIME> <NB OF PASSES>|-bruteforce|
      -randomeandtwomultipass <RANDOM TIME> <NB OF PASSES>|
      -bruteforce2|-random <RANDOM TIME>|-oneotheraverage|-twootheraverage]
     [-bestRegular|-bestMalware|-bestAll|-bestSetOnPercentageOnly]
     [-nocsvname]
     -test
```

Note that you muts provide exactly two guess cluster files. Also, the "-test" option is mandatory.

3.3.3 Client / Server architecture

Each time Alligator is launched to classify a given guess cluster, it has to do some computations on the regular and malware clusters which are supplied. If you are going to call Alligator several times, it is interesting to use the client / server architecture to speed up the process and do some computations only once.

To do so, you need first to launch the Alligator daemon (server): `java -Xmx2048m -cp ./src AlligatorDaemon -regular clean.csv -malware malware.csv -execute script.ali`.

Then, you just launch the Alligator client each time you wish to classify samples: `java -Xmx2048m -cp ./src AlligatorClient -guess guess.csv`.

The usage for Alligator daemon is the following:

```
java AlligatorDaemon [-debug] [-help] -regular <CLEAN CLUSTER>
       -malware <MALICIOUS CLUSTER>
       -execute <SCRIPT>
       [-maxclustersize <int>]
       [-maxpercentageofinvalidproperties <int in [1..100]>]
       [-port <PORT NUMBER>]
```

The meaning of these options are common to other Alligator commands, except the port option: Alligator use a socket on that port number. The default port number is **7078**.

The usage for Alligator's client is the following:

```
java AlligatorClient [-debug] [-help]
       -guess <GUESS>
         [-nbofsamples <int>]
         [-allsamples]
       [-fullreport]
```

- guess: specify a guess cluster. By default, only the first sample in this guess cluster will be classified. See nbofsamples and allsamples options.
- nbofsamples: if this option is specified, the n first samples of the guess cluster are classified
- allsamples: if this option is specified, all samples in the guess cluster are classified. This option overrides the nbofsamples option.
- fullreport: by default, the alligator client only returns the regular score and the malware score for the sample(s) it classified. With this option, Alligator generates a report of each of its analysis.

3.4 Learning Algorithms

- **bruteforce**: parse all combinations, build a binary tree of all combinations and then compute results for all leaves of the tree. The entire tree is kept in memory, so this is quite demanding.
  Syntax:

  ```
  -bruteforce
  ```

- **bruteforce2**: this is an optimized version of bruteforce. The tree is built in parallel with computations: each time there is a leaf, alligator computes the result. Alligator does not wait to have built the entire tree to compute results.
  Syntax:

  ```
  -bruteforce2
  ```

- **oneotheraverage**: of all weights which have a range specified, the algorithm starts working on the first one, and sets all other ranges to their average value. When the algorithm has found the best value for the first variation, it memorizes the value. Then, it works on the second range. The first range is set to the best value, all others are set to average etc. At the end, the learning program display the best values for each ranges.
  This algorithm has the drawback of not taking into account the correlation between some parameters.
  Syntax:

`-oneotheraverage`

- **random**: tests all weights randomly for a given time and outputs the best combination
  Syntax:

  `-random <SECONDS>`

  where seconds is the time to randomly look for the best combination.

- **randomandonemultipass**: same as random, and then optimizes each weight one by one (like in oneotheraverage). It does the specified number of passes to optimize. Syntax:

  `-randomandonemultipass <SECONDS> <NB OF PASSES>`

  where:

  - seconds: a positive integer. The time to randomly look for the best combination (random algorithm)
  - nb of passes: a positive integer. The number of times to run the OneOtherAverage algorithm to optimize weights

- **randomandtwomultipass**: The syntax is the same as random and one multipass.

- **twootheraverage**: same as oneotheraverage except the algorithm works simultaneously on two ranges at a time. The others are set to the average value. The combinations are much larger, thus this algorithm takes longer to complete. Syntax:

  `-twootheraverage`

3.5 Alligator Script Language

An alligator script explains to alligator which probabilistic computations seem to be the most important. There are two different types of alligator scripts: scripts for the learning, and scripts for partitioning. Both scripts use the same syntax, it's only that some commands do not make sense in the context of learning and reciprocally for partitioning.

For each sample to partition, alligator always computes *two scores*: a score related to cluster R and a score related to cluster M. A score is a numeric value indicating whether the given sample bares strong similarity with elements of R or of M. The higher the score is, the strongest is the similarity.

3.5.1 Syntax of a script file

Each line of the script is a command:

- comments are lines that begin with a sharp #
- empty lines are meaningless
- commands **are not case-sentive**

There are 3 main different types of commands:

- output information: `printClusterSummary`, `jumpLineInReport`...
- set a weight: `setcomplexweight`, `setmultiplierweight`... Those commands apply until they are superseded by another one. For alligator learning, the weights are typically ranges, whereas for partitioning they are fixed values.
- do computations: `compute`. Alligator computes deviations, proximity, correlations, epsilon clusters SVM, addValueToScore. See 3.5.2.

Finally, there are scripts for Alligator, and scripts for Alligator learning. Both are extremely similar except an Alligator learning script contains ranges of values, whereas for Alligator, all values are fixed.

Command	Classification	Learning
compute	√	√
computeOverallScores	√	
generatePlot	√	
jumpLineInReport	√	
setbasicscore	√	√
percentageMin		√
percentageWeight		√
printClusterAverageValues	√	
printClusterSummary	√	
printExclusivePropertyCorrelations	√	
printPropertyCorrelations	√	
printResultSummary	√	
printStatistics	√	
removePropertiesAfter	√	√
removePropertiesBefore	√	√
selectPropertiesRange	√	√
setMultiplierWeight	√	√
setClusterName	√	√
setComplexWeight	√	√
setPrintIntermediateScore	√	
setPropertyWeightByName	√	√
setPropertyWeightsFromColumn	√	√
setMaxNbOfValuesPerProperty	√	√
setdiffvalue	√	
usePropertyNames	√	√
printProbaStatistics	√	
printcorrelationstatistics	√	
printproximityelements	√	
printepsiloncclustercontent	√	

Table 1: Utility of commands for alligator or alligator learning. Some commands don't make sense or are useless for learning.

3.5.2 Script Language Reference

compute

```
compute <WHAT> [parameters]
```

Specifies a probability computation to perform. $what$ is one of the following:

- addValueToScore. Alligator computes an increment to add to the score of each of the clusters. This is used to make scores between both clusters comparable. Syntax:

  ```
  compute addvaluetoscore <INTEGER>
  ```

  The integer is the increment to add, multiplied by the multiplier weight that Alligator's learning will select.

- correlation. Alligator computes the correlations between 2 properties, for all properties. Syntax:

```
compute correlation <THRESHOLD>
```

where $threshold$ is the minimum correlation value to consider.

- degressiveproximity. This computes a higher score if you are very close to a given cluster, and a lower score if you are far from that cluster.

```
compute degressiveproximity <NEIGHBOUR>
```

- deviation. This computes how far from the mass a given sample is. Syntax:

```
compute deviation
```

- inversedeviation. This computes how far from the mass a given sample is. With regards to the deviation, it returns the inverse value, e.g. $x = \frac{10}{x+0.000000001}$ Syntax:

```
compute inversedeviation
```

- epsilonCluster. An epsilon cluster is the set of elements which have a distance of d with a given element. The size and number of epsilon clusters depend on the distance to be used. Syntax:

```
compute epsiloncluster <DISTANCE>
```

where $distance$ is the maximum distance within a given epsilon cluster.

- SVM. Syntax:

```
compute SVM
```

- proximity. This consists in counting which samples are closest to one another (based on its properties). Syntax:

```
compute proximity <NEIGHBOUR>
```

where $neighbour$ is the number of neighbours to take into account

- proximityratio. This is similar to proximity except the score takes into account the number of elements in the cluster. For example, if the regular cluster has 10 items, and the malware cluster has 10,000, there are higher chances to find neighbours in the malware cluster than in the regular one. So, the proximity score will probably output a rather high score for malware. With proximity ratio, the score is leveled with the number of items in the cluster, and so this will no longer be the case.

```
compute proximityratio <NEIGHBOUR>
```

- proximityWithLimitedProperties. This is like proximity but only considering a subset of properties. Syntax:

```
compute proximityWithLimitedProperties <NEIGHBOUR> <NB OF PROPERTIES>
```

where $neighbour$ is the number of neighbours to consider, and $nbofproperties$ is the number of most relevant properties to consider.

- probabilityDifference. This highlights the properties which are the most different from one another. Syntax:

  ```
  compute probabilityDifference <NUMBER>
  ```

  where $number$ is the probability difference (so $0 \leq number \leq 1$)

- probabilityFactor. This is the same as proximity differences except the difference is measured in terms of factor, i.e multiplicative difference and not an arithmetic difference. Syntax:

  ```
  compute probabilityFactor <FACTOR>
  ```

  where $factor$ is the probability factor (multiplication). $factor$ is a positive integer.

- weightdeviation. Like deviation, but uses different weights on each properties. Syntax:

  ```
  compute weightdeviation
  ```

- inverseweightdeviation. Like weightdeviation, but returns the inverse of the result. Syntax:

  ```
  compute inverseweightdeviation
  ```

- addValueToScore. Adds a constant value to the score of each cluster. Syntax:

  ```
  compute addValueToScore
  ```

  For example, let's assume the following subpart of an alligator script:

  ```
  # This is a sample cluster
  setMultiplier regular 5
  setMultiplier malware 12
  compute addValueToScore 3
  ```

  So, the constant value 15 $(= 3 * 5)$ is added to the score of the regular cluster. Similarly, the constant value 36 $(= 13 * 3)$ is added to the score of the malware cluster.

See 3.6 to understand exactly what each case computes.

computeOverallScores

```
computeOverallScores
```

Tells alligator to gather all scores and compute the final score. This command does not make sense (useless) for learning.

generatePlot

`generatePlot <WHAT> <fileGToStoreData> <fileRToStoreData> <fileMToStoreData>`

where:

- $what$ is one of the following. It indicates what to plot. The 'all' versions generate plot data for 3 clusters: regular, malware and to analyze. In all cases, the clusters used for generating the plot data are the one given as argument to alligator.
  - deviation
  - alldeviations
  - proximity
  - allproximity
  - probabilityfactor
  - allprobabilityfactor
  - correlation
  - allcorrelation
  - probabilitydifference
  - allprobabilitydifference
  - epsilon
  - allepsilon
- $fileGToStoreData$ is the file used for writing the results for the guess cluster.
- $fileCToStoreData$ is the file used for writing the results for the clean cluster. This file must be given as argument only when the $what$ starts with "all". Otherwise, this file is not used.
- $fileCToStoreData$ is the file used for writing the results for the malware cluster. This file must be given as argument only when the $what$ starts with "all". Otherwise, this file is not used.

This command generates a data file which can be plotted using a tool such as GnuPlot. The output below corresponds to the generated data file for $what = deviation$:

```
#Regular and malware deviation
blah 0.3537359468058742 0.18677817838643504
wooo 0.30286719830490794 0.33113014797193
droid 0.36394339907056134 0.27820369378662163
```

The first column is the sample's name. The second column is the deviation score with regards to the regular cluster (R) and the third column is the deviation score with regards to the malware cluster (M).

jumpLineInReport

`jumpLineInReport`

adds an empty line to the output.

setBasicScore sets a different mode for computing scores. Indeed, with "setBasicScore", scores are in $0, 1$ that is, let's consider the two learning clusters $c0$ and $c1$. The results of the computation of a given classification algorithm is used to give a score for cluster $c0$ and $c1$. It the score for $c0$ is higher than the score for $c1$, then the "1" score is given to $c0$ and "0" to $c1$, and reciprocally. Usage:

`setBasicScore`

adds an empty line to the output.

percentageMin

`percentageMin <cluster> <percentage>`

Specifies the desired mininum percentage of score for a given cluster. Cluster is `regular` or `malware` (it cannot be 'guess' as this does not make sense), and percentage is a positive double between 0 and 100. By default, percentageMin is set to 0 for both clusters. Example:

```
percentageMin regular 95
percentageMin malware 92
```

With such a configuration, alligator learning will select configuration that provide at least 92% of correctness for malware and 95% for regular.

percentageWeight Specifies the importance of the score for a given cluster. Format is

`percentageWeight <cluster> <weight>`

This weight is used to compute the overall average percentage the alligator learning achieves. So, for instance, if we have:

```
percentageWeight regular 3
percentageWeight malware 2
```

Then, the resulting percentage for the regular cluster is more important than the one for the malicious cluster.

The overall average percentage will be: $\frac{percentage\_regular*3+percentage\_malware*2}{3+2}$

printClusterAverageValues

`printClusterAverageValues`

Prints the average values for each properties.

printClusterSummary

`printClusterSummary <CLUSTER>`

Prints a line stating how many elements and properties there are in a cluster. This command takes a cluster name as argument: guess, regular or malware.
Example:

```
printClusterSummary guess
```

outputs for instance:

```
ToBeAnalyzed - 19 elements in cluster, nb of properties: 72
```

This command is not useful in an alligator learning script.

printExclusivePropertyCorrelations

`printExclusivePropertyCorrelations <CLUSTER>`

cluster: displays correlations whose value is $\geq$ to the threshold mentioned by `compute correlations` and which does not exist in the other cluster. In other words, those are correlations which are interesting for a single given cluster (exclusive). Example of output:

```
Exclusive interesting correlations of regular
```

In that case, there were *no* exclusive correlations.

printPropertyCorrelations

```
printPropertyCorrelations <CLUSTER>
```

cluster: displays correlations whose value is $\geq$ to the threshold mentioned by `compute correlations`.

Example of output:

```
Interesting correlations of malware:
Correlation between property 3 (nb_classes) and 4 (nb_dir) = 0.87339202115768
Correlation between property 50 (embed_exec) and 51 (arm_exec) = 0.7796355742651
Correlation between property 52 (load_lib) and 64 (jni) = 0.98362549280080
```

printResultSummary

```
printResultSummary
```

Prints the best guesses of alligator for each file of the guess cluster (files to analyze). Example:

```
printResultSummary
```

```
Results summary:
./scanning/guess/7a4eb625d7c24004a735d3334cc881a0: regular
  (regular:23097.069341625, malware:1298.6062567437907)
./scanning/guess//Activator.apk: *malware*
  (regular:10815.493623392516, malware:21731.317753346888)
```

printStatistics

```
printStatistics
```

Prints statistics about classified elements, e.g., the one with the highest malware score, etc.. Example:

```
printStatistics
```

```
*** STATS ***
     highest scores:
Element of guess with the highest regular score: com.invoice2go.invoice2goplus.apk
(classified as regular)
Element of guess with the highest malware score: Spammer.apk (classified as malware)
     lowest scores:
Element of guess with the lowest regular score: wirelesstether20.apk
(classified as malware)
Element of guess with the lowest malware score: net.bible.android.activity107.apk
(classified as regular)
     closest to the average score:
Element of guess being the closest to the average of regular scores:
me.scan.android.client.apk (classified as regular)
Element of guess being the closest to the average of malware scores: A6aRurV6Rw.apk
(classified as regular)
     Highest score difference:
Element of guess having the highest absolute score difference between regular and
malware: com.invoice2go.invoice2goplus.apk (classified as regular)
     Lowest score difference:
Element of guess having the lowest absolute score difference between regular and
malware: de.gdata.mobilesecurity76.apk (classified as malware)
```

removePropertiesAfter Tells Alligator to ignore all properties after a given index. This applies to each line of the cluster. Syntax:

```
removePropertiesAfter <n=INTEGER>
```

In clusters, column indexes start at 1. So, if a cluster has this format: `filename, prop1, prop2, prop3, prop4`
then the command: `removePropertiesAfter 2`
then, this will actually ignore **prop2, prop3, prop4...** and only keep: `filename, prop1`
This is because clusters consider filename as a 'first' property.

removePropertiesBefore Similar to removePropertiesAfter except properties up to a given index are ignored. Syntax:

```
removePropertiesBefore <n=INTEGER>
```

Example: If a cluster has this format: `filename, prop1, prop2, prop3, prop4`
and we specify `removePropertiesBefore 3`

Then, this removes **prop1** (filename cannot be removed), which means Alligator considers: `filename, prop2, prop3,`

selectPropertiesRange Tells Alligator to ignore all properties that are not in the provided range. This applies to each line of the cluster. Syntax:

```
selectPropertiesRange <INTEGER1> <NB>
```

INTEGER1 corresponds to the first properties to be taken into account, and NB corresponds to the number of properties that are taken into account. Column indexes start at 1. All columns between 1 and INTEGER1 (excluded) are ignored. All columns after $(INTEGER1 + NB)$ are ignored.

Example: If a cluster has this format: `filename, prop1, prop2, prop3, prop4, prop5`
and we specify `selectPropertiesRange 3 2`
then we ignore prop1 (filename cannot be ignored), and we ignore prop5. `filename, prop3, prop4`

setMultiplierWeight Specifies a weight. For alligator scripts, the syntax is:

```
setMultiplierWeight <cluster> <x>
```

where $cluster$ is *malware* or *regular*, and x is the weight (a positive double). Example:

```
setMultiplierWeight regular 2
compute proximity 5
```

See 3.6 to understand how scores are computed.

For alligator learning scripts, the syntax also supports ranges of weights such as:

```
setMultiplierWeight <cluster> <min>-<max>,<step>
```

In that case, the learning program will use weights from min to max with a step of $step$ ($step$ being a positive double). The multiplier weight range applies for each subsequent computations, until it is superseded by another directive. So, for instance, if we have:

```
1 setMultiplierWeight regular 0-10,1
2 compute correlations 0.75
3 compute correlations 0.85
```

In that case, the learning process will compute the best multiplier weight for correlation 0.75, and the best weight for 0.85. For example, the output generated script may be as follows:

```
setmultiplierweight regular 100.0
compute correlations 0.75
setmultiplierweight regular 60.0
compute correlations 0.85
```

If one **does not want** to compute the best multiplier for correlation 0.85, then line 3 (compute correlations 0.85) should be removed from the alligator *learning* script, and only added in the alligator *partitioning* script.

```
setClusterName <oldName> <NewName>
```

means that cluster named "oldName" is now name "newName".
Clusters are - by default - named "regular" , "malware" and "guess". The setClusterName gives the possibility to give them another name, athan can be used on all further commands of the script, instead of using e.g. "regular ". Yet, the three default names ("regular", "malware", "guess") can still be used even when clusters have been renamed. Here a a few examples:

```
setClusterName malware Virus
setClusterName regular Application
setClusterName guess Unknown
```

"regular" and "malware" (with any uper/lower case letters combinations) are not allowed as new names. Also, the two clusters must have a different name.

setComplexWeight Specifies formulas as weights. For alligator scripts, the syntax is:

```
setComplexWeight <cluster> <weight> <formula>
```

where $cluster$ is *malware* or *regular*, $weight$ is a positive double and $formula$ is a formula that uses a single variable, x. When computing the score, x is replaced by the result (i.e the deviation, or proximity etc). For example, a valid formula is illustrated below:

```
setComplexWeight malware 16 (1/x)*10
compute deviation
```

The score is $score_{malware} = \frac{1}{score_{malware}(deviation)} * 10 * 16$

For a given computation, the last multiplier or complex weight supersedes the others. See 3.6.

For alligator learning scripts, the syntax also supports ranges of weights such as:

```
setComplexWeight <cluster> <min>-<max>,<step> <formula>
```

Note that the formula does not change, only the weight multiplier. So, for instance, with:

```
setComplexWeight regular 20-22,11 (1/x)*10
compute deviation
```

The learning program will compute:

$$\frac{1}{score_{regular}(deviation)} * 10 * 20 \quad (2)$$

$$\frac{1}{score_{regular}(deviation)} * 10 * 21 \quad (3)$$

$$\frac{1}{score_{regular}(deviation)} * 10 * 22 \quad (4)$$

setPrintIntermediateScore

```
setPrintIntermediateScore
```

By default, alligator provides a final score for each sample, i.e. a malware score and regular score. The "setPrintIntermediateScore" specifies that a regular and malware score shall also be printed for each intermediate algorithm, thus helping to determine how each algorithm impacts the final score.

setPropertyWeightByName specifies the weight for a given column (property) of a cluster, where the column is referenced by its name.

```
setPropertyWeightByName <column name> <weight>
```

This tells alligator to apply weight *weight* to the column with name *column name*. A column's name is defined in the property line of each cluster: see section .

For example,

```
setPropertyWeightByName embed_exec 10
```

sets a weight of 10 to the column named "embed_exec".

setPropertyWeightsFromColumn specifies different weights for columns (properties) of a cluster.

```
setPropertyWeightsFromColumn <column> <weight> [<weight2> ...]
```

This tells alligator to apply weight *weight* to *column-th* column.

Precisely, a cluster hash this format: `filename, prop1, prop2, prop3`
then,

- the 1st column contains the filename
- the 2nd column contains prop1
- the 3rd column contains prop2
- the 4th column contains prop3

So, for example if you set `setPropertyWeightsFromColumn 2 10`
then this means you apply a weight of 10 to the 2nd column, i.e to "prop1".

If more than one weight is specified, then the next weights apply to the next columns. For example,

```
setPropertyWeightsFromColumn 25 5 2 1 3
```

means that column 25 has weight 5, column 26 has weight 2, column 27 has weight 1, column 28 has weight 3.
It is possible to issue several setPropertyWeightsFromColumn directives in a single file. For example, the following is equivalent to the previous syntax:

```
setPropertyWeightsFromColumn 25 5
setPropertyWeightsFromColumn 26 2
setPropertyWeightsFromColumn 27 1
setPropertyWeightsFromColumn 28 3
```

setMaxNbOfValuesPerProperty specifies how many different values can be used at most for each property of all clusters.

```
setMaxNbOfValuesPerProperty <nbOfValues>
```

nbOfValues must be a positive integer. For example, if $nbOfValues = 3$, then the three possible values for properties are $\{0, 0.5, 1\}$.

setDiffValue specifies a value, computed during the learning phase, that is used by Alligator in classification so as to classify elements in light/medium/strong. That value is not meant to be changed by the used of Alligator.

```
setDiffValue <value>
```

nbOfValues must be a positive float.

usePropertyNames specifies that both the index and the names of properties are used in the generated report and traces of Alligator.

```
usePropertyNames
```

printProbaStatistics prints in the classification report, and for each property, the probability for each possible property value.

```
printProbaStatistics
```

printCorrelationStatistics prints statistics on correlations in the classification report.

```
printCorrelationStatistics
```

printProximityElements prints in the classification report, and for each kind of proximity and for each element of the guess cluster, the elements that are the closest to the guess element.

```
printCorrelationStatistics
```

printepsiloncluster content prints in the classification report, and for each element of the guess cluster, the elements of the two konwn clusters that are in the same epsilon cluster as the considered guess element. For each guess element, at most 1000 elements of the same epsilon cluster are printed.

```
printCorrelationStatistics
```

3.6 Alligator Score

3.6.1 Partial scores

Each time a line "compute" is in the weight script, a partial score is computed using the last multiplier or complex weight in action. Then, each partial scores are added and make the total score.

Example:

```
setMultiplierWeight malware 2
compute proximity 5
compute correlations 0.75
```

The score for proximity is the percentage of malware in the 5 closest samples to the one analyzed. The score for correlation is the percentages of properties with a correlation ≥ 0.75 compared to those for malware which have a correlation ≥ 0.75. For example, if malware have 5 properties with correlations ≥ 0.75 and the sample analyzed has only 1, then the score is $20\% = 0.2$. The total score is:

$$score_{malware} = 2 * (score_{malware}(proximity, 5) + score_{malware}(correlation, 0.75)) \quad (5)$$

Example 2:

```
setMultiplierWeight malware 3
setMultiplierWeight regular 5
compute addValueToScore 2
```

The partial scores are

$$score_{cluster} = weightmultiplier * addValueToScoreInteger \tag{6}$$

In this example, partial scores are:

$$score_{malware} = 3 * 2 \tag{7}$$

$$score_{regular} = 5 * 2 \tag{8}$$

3.6.2 Weights to scores

Weights apply to computations and help indicate which computations are the most important. For instance, in the following example, the deviation is much more important for malicious samples than for regular ones:

```
setMultiplierWeight regular 1
setMultiplierWeight malware 16
compute deviation
```

The deviation is multiplied by 16 for a malware, and by 1 for a regular sample. (This is only an example).

3.6.3 Superseding weights

Another rule is that weights supersede each other, whether they are complex or multiplier weights.

```
setMultiplierWeight regular 1
setComplexWeight regular 16 (1/x)*10
setMultiplierWeight regular 4
compute deviation
```

In the example above, the deviation for regular is multiplied by 4. The first two lines are superseded by the 3rd one.

On the other hand, until a new weight is mentioned, the previous weight applies.

```
setMultiplierWeight regular 4
compute deviation
compute proximity 5
```

So, for instance, in this example, the multiplier 4 is applied to both the deviation score and the proximity score.

4 Results

4.1 Understanding an Alligator report file

```
Regular - 8187 elements in cluster, nb of properties: 76
```

There are 8187 elements in the cluster regular. Each element is described by 76 properties.

```
Regular: Average values of properties:  (0,filetype,0.0)
(1,sha256,invalid) (2,filesize,0.018768353952744683)
(3,nb_classes,0.040012912923863074)  (4,nb_dir,0.015866756580088206)
(5,appstore,invalid) (6,os_version,0.18741886216112907)
..
```

The average file type (column 0) for regular samples is 0. The average sha256 (column 1) is... invalid. Indeed, SHA-256 are provided as hexadecimal strings, so an average cannot be computed. etc.

This report was generated using named columns (each column is labeled with a corresponding name).

```
Interesting correlations of regular:
Correlation between property 3 (nb_classes) and 4 (nb_dir) = 0.7805926262450
Correlation between property 50 (embed_exec) and 51 (arm_exec) = 0.957931618369
..
```

Alligator reports there is an 'interesting' correlation between column 3 and 4, where column 3 corresponds to the number of classes and column 4 the number of directories. By 'interesting', alligator means the correlation is above the threshold that was fixed in the alligator script (`compute correlations`). Those lines are generated when the alligator script contains a `printPropertyCorrelations` command.

```
./scanning/guess//Activator.apk: *malware*        (regular:10815.493623392516,
  malware:21731.317753346888)
./scanning/guess//Gmail.apk: regular        (regular:23829.131072270342,
  malware:1658.7239829070968)
..
1 malware(s) found
```

Those are results summary for two samples. Sample Activator.apk seems to fall in the malware category. Its score for regular is 10815. Its score for malware is 21731.
Sample Gmail.apk looks like a regular file as its score for regular (23829) is far higher than the score for malware (1658).

```
----------------------------------------
./scanning/guess//Activator.apk:
Deviation:  with Regular = 0.27545160848521444
 with Malware = 0.30317046938836306

Proximity 50 elements: Malware/50
Proximity 25 elements: Malware/25
Proximity 10 elements: Malware/10
Proximity 5 elements: Malware/5
```

This is the beginning of a detailed report for sample Activator.apk. The deviation score for that sample is of 0.275 for regular and 0.303 for malware. This means that the sample deviates less from the regular cluster than from the malicious one.

However, there are other computations to take into account. Alligator searched for the 50 closest elements to Activator.apk in the regular and the malware cluster. Those 50 closest elements where only from the malware cluster! Obviously so where the 25 closest, 10 closest and 5 closest.

```
Properties with >= 0.5 probability difference in a given value
between regular and malware; 9 properties in regular
(2,3,4,7,17,21,62,74,75); 8 properties in malware: (0,2,3,4,6,7,21,56)
- In Regular sample respects: (); In Malware sample respects: ()
```

There are 9 properties (columns 2, 3, 4, 7, 17, 21, 62, 74 and 75) which have a probability difference ≥ 0.5 TO DO.

```
  Regular score: 10815.493623392516
  Malware score: 21731.317753346888
* ->>>>> Malware
```

Outputs the total scores for the sample. This repeats what has already been said in the result summary. Again, alligator says the sample is probably malicious.

4.2 Generating graphs

For each `generatePlot` command in the alligator script, alligator generates data. Data may be plotted using GnuPlot, for instance using such a script:

```
set terminal png
set output 'deviation.png'
set   autoscale                        # scale axes automatically
unset log                              # remove any log-scaling
unset label                            # remove any previous labels
set xtic auto                          # set xtics automatically
set ytic auto                          # set ytics automatically
set title "Deviation"
set xlabel "With regular"
set ylabel "With malware"
set xr [0:1]
set yr [0:1]
plot    "regulardeviationvalues" using 2:3 title 'Regular' with points lt 2 pt 1, \
 "malwaredeviationvalues" using 2:3 title 'Malware' with points lt 1 pt 2, \
 "guessdeviationvalues" using 2:3 title 'Guess' with points lt 3 pt 6
```

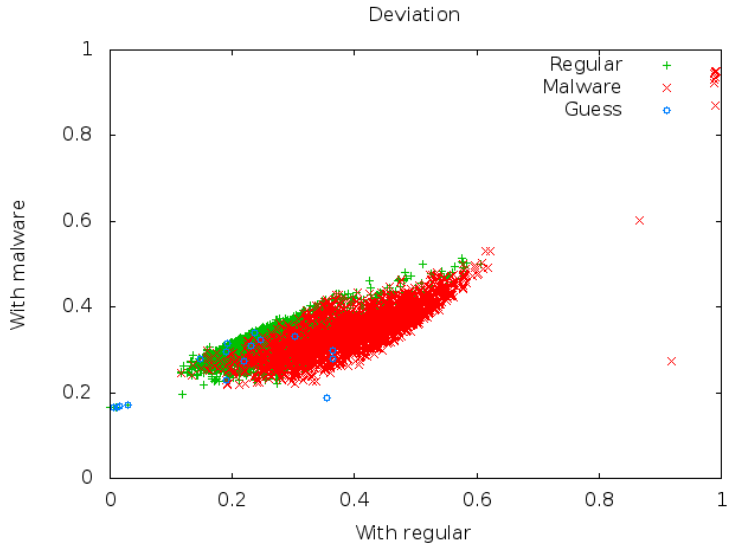

Figure 2: Deviations generated by GnuPlot

References

[AA13] L. Apvrille and A. Apvrille. Pre-filtering mobile malware with heuristic techniques. In *GreHaCk'2013*, Grenoble, France, November 2013.

[AA14] A. Apvrille and L. Apvrille. Sherlockdroid, an inspector for android marketplaces. In *Hack.lu*, Luxembourg, October 2014.

[CL11] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[Coh89] Fred Cohen. Computational aspects of computer viruses. *Computers & Security*, 8(4):297–298, 1989.

[Soe14] David Soergel. Efficient training of Support Vector Machines in Java, 2014. `https://github.com/davidsoergel/jlibsvm`.