J Comput Virol Hack Tech (2016) 12:225–233
DOI 10.1007/s11416-015-0258-7

ORIGINAL PAPER

Subroutine based detection of APT malware

Joseph Sexton[1] · **Curtis Storlie**[1] · **Blake Anderson**[1]

Received: 7 May 2015 / Accepted: 20 November 2015 / Published online: 21 December 2015

Abstract Statistical detection of mass malware has been shown to be highly successful. However, this type of malware is less interesting to cyber security officers of larger organizations, who are more concerned with detecting malware indicative of a targeted attack. Here we investigate the potential of statistically based approaches to detect such malware using a malware family associated with a large number of targeted network intrusions. Our approach is complementary to the bulk of statistical based malware classifiers, which are typically based on measures of overall similarity between executable files. One problem with this approach is that a malicious executable that shares some, but limited, functionality with known malware is likely to be misclassified as benign. Here a new approach to malware classification is introduced that classifies programs based on their similarity with known malware subroutines. It is illustrated that malware and benign programs can share a substantial amount of code, implying that classification should be based on malicious subroutines that occur infrequently, or not at all in benign programs. Various approaches to accomplishing this task are investigated, and a particularly simple approach appears the most effective. This approach simply computes the fraction of subroutines of a program that are similar to malware subroutines whose likes have not been found in a larger benign set. If this fraction exceeds around 1.5 %, the corresponding program can be classified as malicious at a 1 in 1000 false alarm rate. It is further shown that combining a local and overall similarity based approach can lead to considerably better prediction due to the relatively low correlation of their predictions.

✉ Joseph Sexton
joesexton0@gmail.com

[1] Los Alamos National Laboratory, Los Alamos, NM, USA

Keywords APT · Malware detection · Static analysis · Subroutine similarity

1 Introduction

Advanced Persistent Threat, or APT, has emerged in recent years as a significant concern to the network security of larger businesses and government agencies [28]. This type of cyber attack is used to denote attacks from well organized groups who persistently attempt to gain and maintain access to a targeted network to obtain a specific objective, typically the exfiltration of intellectual property.

A targeted network intrusion commonly involves the installation of one or more malicious programs. Therefore, a natural approach to detecting an APT intrusion is looking for tell tale signs of known APT malware, such as file hashes and byte-sequences found in other targeted malware. These approaches are valuable, however, it is known that signature based detection can fail in the presence of even minor modifications of a malicious program, see e.g. [3]. This fact has prompted significant research into statistical malware classification, which has shown great success at detecting so-called mass malware. A key reason for this success is the high similarity between malware programs, and Microsoft estimates that 75 % of the most prevalent mass malware can be grouped into as few as 26 families [16].

Statistical malware classification of APT malware has received little attention in the literature. However, there is reason to believe that the approach will be valuable, due to reported commonality between various APT malware. For instance, [7] examined 11 seemingly unrelated APT campaigns targeting a range of industries. The report uncovered that the campaigns used malware sharing common code elements, and concluded that the attacks shared a common

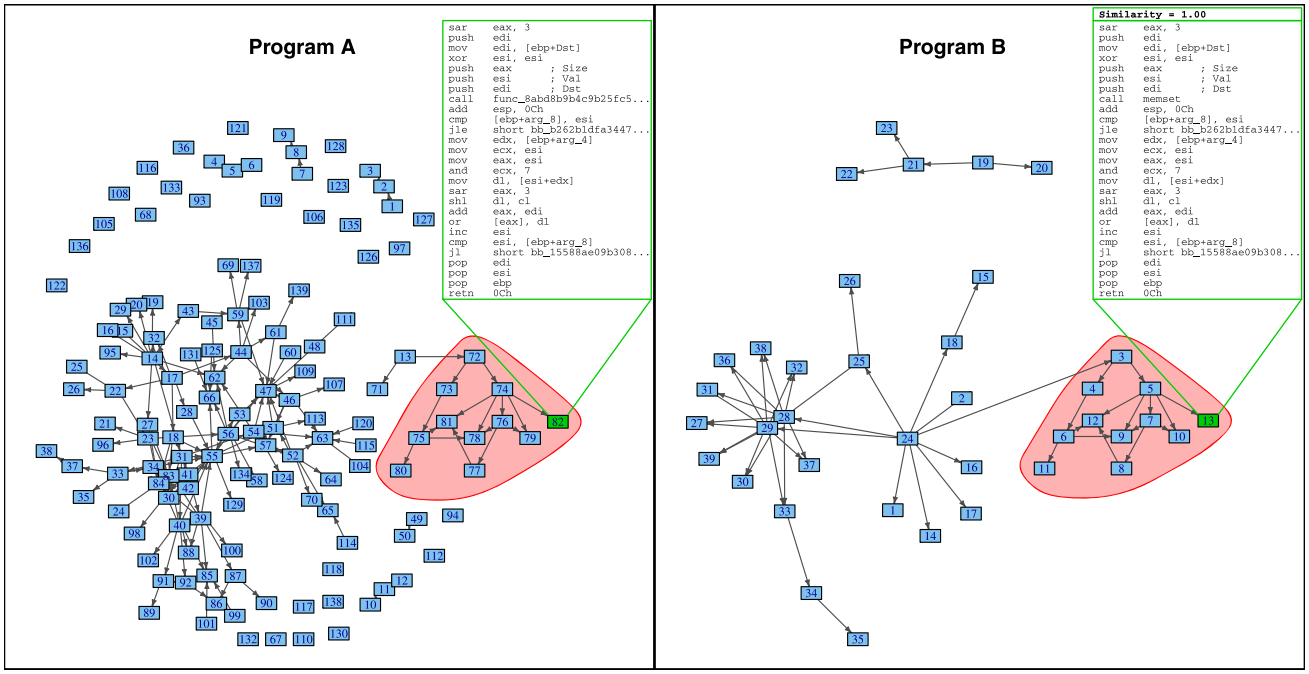

Fig. 1 Two programs. Boxes represent subroutines of program, and arrows depict one subroutine calling another. The shaded region of subroutines for Program A and Program B represents a shared set of subroutines. Program A was correctly classified as malicious by an svm classifier based on overall 2-gram opcode similarity, while Program B was incorrectly classified as benign, despite Program A being in APT malware training sample. The subroutines in the shaded region were not matched in benign set of 4500+ programs

development infrastructure. [21] provide further evidence of the reuse of code in APT malware, and describe a clustering methodology for extracting common functional elements. Also using clustering, [13] graphically depict the relationships between several APT malware samples, showing how they can be naturally grouped into families. These observations indicate that statistical detection of APT malware will be fruitful, and a primary goal of this work is to investigate this issue. As a case study we use malicious executables from the APT 1 family, which were released along with a highly publicized report [15].

The bulk of statistical malware classification approaches are based on different notions of overall program similarity. For instance, [12] summarized an executable using counts of byte-sequence n-grams across the entire program, see also [19]. [22] used a related approach and considered the n-grams of assembly level operational codes (opcodes), while [1,27] showed that treating the opcode sequence as a Markov chain can be highly beneficial. [20] describe a variant of the Markov chain approach and apply it to metamorphic malware detection, a subject that received considerable interest [10,23,26,29,30]. [4] described an approach to comparing the similarity between two binary files without the need for feature extraction, and [2,25] use the concept of structural entropy to identify similar binary files. Methods using API calls have also received considerable attention, such as [9] where program similarity is based on the edges in the API call-graph. Subroutine call-graph similarity has also been extensively studied. For instance, [31] describe an approach where the similarity between programs is defined as the number of edges in the program graph that are shared, [11] use graph edit distance to measure similarity between programs, and [17] describe behavioral graph clustering.

Though many of the above overall similarity measures have shown to produce highly accurate classifiers, they do have one undesirable property. In particular, they will have difficulty in accurately classifying programs that reuse a smaller fraction of known malware code. One such classifier is applied in Sect. 5 of this article, namely the SVM classifier of [1], which is based on overall opcode similarity. This classifier performed well in our investigations, however its results did suggest room for improvement. An example is given in Fig. 1 where two APT programs that share a block of subroutines are graphically depicted. Here the boxes represent the subroutines, and the directed edges their call structure. The classifier correctly classified Program A as being malicious, however it failed to correctly classify Program B, even when Program A was in the training set of APT programs. One reason for this failure could be that the local similarity exhibited by these two programs is diluted by the overall similarity measure used by the classifier.

To avoid the above type of behavior, this article describes a new approach to malware classification. In particular, instead of computing overall similarities between programs, we explicitly focus on the similarity of a programs subroutines with that of known malware subroutines. The rationale for doing this is that when code and functionality is reused this is most naturally done at the subroutine level. The approach has the added advantage that a malware program can be correctly classified even when its overall characteristics are more similar to known benign programs, as long as it contains subroutines similar to the known malicious set. Various approaches are introduced to carry out the subroutine based classification. One is a logistic regression where the features of a program are the similarities to the malware subroutines, another is a related Naive Bayes approach. In addition, two simple summaries are introduced to describe how similar a programs subroutines are to the malware set, taking into consideration the number of times the similar subroutines are found in benign programs.

The remainder of the article is as follows. In Sect. 2 we describe the data used in this article, which are disassembled binaries, and the features extracted from these. In Sect. 3 the subroutine based classifiers are introduced, and Sect. 4 gives the application to the APT sample. Section 5 concludes.

Listing 1 A disassembled function

```
func_223f4
...
    mov     eax, [esp+43Ch+var_410]
    lea     ecx, [esp+43Ch+dwNumberOfBytesRead]
    push    ecx ; lpdwNumberOfBytesRead
    lea     edx, [esp+440h+Buffer]
    mov     ecx, [eax+4]
    push    3FFh ; dwNumberOfBytesToRead
    push    edx ; lpBuffer
    push    ecx ; hFile
    call    ds:InternetReadFile
    mov     edx, [esp+43Ch+dwNumberOfBytesRead]
    mov     [esp+43Ch+var_414], eax
    lea     edi, [esp+43Ch+Buffer]
    or      ecx, 0FFFFFFFFh
    xor     eax, eax
    mov     [esp+edx+43Ch+Buffer], bl
    repne   scasb
    mov     eax, ds:dword_4091EC
    mov     edx, [esp+43Ch+var_420]
    not     ecx
    dec     ecx
    sub     eax, edx
    mov     esi, ecx
    cmp     eax, esi
    ja      short bb_86d4
    call    func_7b5e
...
func_223f4 endp
```

2 The data

There are two high level categorizations of malware detection techniques, static analysis and dynamic analysis. The first refers to analysis of an executable file without actually running it. A common approach to performing static analysis is via a disassembler, such as IDA-Pro [5], which produces a text file of assembly code corresponding the binary executable. The disassembled file breaks the code into subroutines, gives the opcodes used by the subroutines, their calls to other subroutines as well as to any API calls. An example of a subroutine of a disassembled file is given in Listing 1. Dynamic analysis refers to running the executable file, which is typically done in a sandbox environment. There are pros and cons associated with both analysis approaches. Static analysis gives a more complete picture of a program, however disassembly can be challenging if the program has been packed, essentially encrypting the binary file. Dynamic analysis will always be possible, however conditional execution can limit visibility into the true functionality of the program.

The APT 1 malware considered here consists of 197 programs, spread across 37 (sub-)families. Each of the programs were successfully disassembled using IDA-Pro, and we here focus on designing classifiers based on disassembled executables. In addition to the APT malware, a sample of 4622 non APT disassembled programs is also used. These benign programs were taken from a program analysis tool and repository at Los Alamos National Laboratory called *CodeVision*. In the following, the APT 1 malware sample will be referred to as the malware sample, and the 4622 sample of non-APT programs will be referred to as the benign sample.

A disassembler breaks an executable into its subroutines, and gives the assembly code for each subroutine along with any calls to imported libraries. Listing 1 shows an example of such a subroutine, labelled *func_223f4*. The first column gives the opcodes (mov,leav, push etc.) with the following columns giving the operands. In this code sequence two other functions are called. The first is a call to the API function *ds:InternetReadFile*, and the second is a call to another function in the program, labeled *func_7b5e*.

3 Measuring subroutine similarity

The disassembled subroutines need to be reduced to a format amenable to processing. Here we only use the sequence of opcodes to summarize a subroutine, and ignore API and internal calls as well as the opcode operands. This is done for simplicity, as well as the fact that this sequence has been shown to be a powerful basis for classification [1,27]. The sequence of opcodes in a subroutine are summarized using

n-grams [19]. An n-gram is a sequence of n consequential symbols from some alphabet, here the set of opcodes. In our study we have used 2-gram throughout, with the first three 2-gram in Listing 1 being (mov, lea), $(lea, push)$, $(push, lea)$. Our decision to use 2-gram was based on simplicity as well as the fact that this choice has performed well in other studies [1,27].

There are a large number of opcodes in the X86 instruction set. In [27] it is shown that using a coarser categorization of these codes can be beneficial. In that work a categorization from the Python library 'pydasm' with 86 categories was shown to produce good results. Using 2-gram means that each subroutine is summarized using the count vector of the 86^2 possible 2-gram of the categorized instruction set. More formally, letting u denote a subroutine we use $o_{mk}(u)$ to denote the count of the 2-gram (m, k) in u, summarizing u with the vector $o(u) = \{o_{mk}(u)\}_{m,k=1}^{86}$. A program p is here treated as a collection, or bag, of subroutines and is given by $o(u)$ for $u \in p$.

The distance between the 2-gram opcode sequence in a subroutine u with that of a subroutine v is here defined as

$$d(u, v) = \sum_{m,k} |o_{mk}(u) - o_{mk}(v)|. \tag{1}$$

This distance can be converted to a similarity score, a number between 0 and 1, using

$$s(u, v) = 1 - \frac{\sum_{m,k} |o_{mk}(u) - o_{mk}(v)|}{\sum_{m,k} o_{mk}(u) + \sum_{m,k} o_{mk}(v)} \tag{2}$$

where the denominator on the left gives the total number of 2-gram in the two subroutines.

4 Classifiers

It is perhaps surprising that a number of subroutines found in malware are also common to benign programs. Some of this could be due to the compilation process, some from the borrowing of code from public ally available programs. Whatever the reason, the implication is that classifying a program as malicious simply because it has a subroutine common to a malware program is a poor strategy.

To illustrate, we took the APT 1 sample of programs and computed the similarity between their subroutines, using (2), with those found in a set of around 1700 benign programs. If a benign subroutine was found whose similarity to a malware subroutine was greater than 0.9, a somewhat arbitrary yet high value, the malware subroutine was considered matched. Figure 2 shows the results for three of the malware programs, where the fraction of matched subroutines to the total number of subroutines in each program is plotted. The x-axis in the plot corresponds the number of benign programs examined,

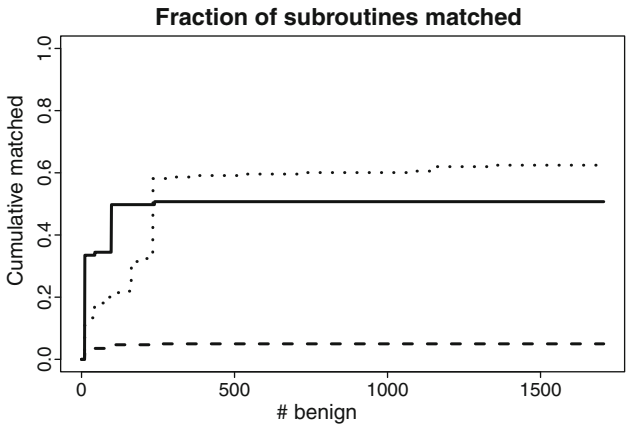

Fig. 2 Cumulative subroutine match. The plot depicts three different APT 1 programs, giving the fraction of subroutines matched by at least one benign program as a function of the number of benign programs examined

while the y-axis gives the cumulative fraction of different subroutines that have been matched. In all three programs, the number of matches climbs rapidly for the first 300 or so benign programs, after which it flattens out.

4.1 Threshold matching classifiers

The plots in Fig. 2 suggest a simple approach to classification, namely basing classification on malware subroutines that are not similar to any, or only a few, of the subroutines in a benign training set. To implement this idea, we say that a subroutine v matches a subroutine u if the similarity between the two equals or exceeds some threshold α, that is if $s(u, v) \geq \alpha$ then u and v match each other, with $s(,)$ defined in (2). This approach can be carried out in two ways. The first is to consider all malware subroutines together, the second is to treat programs separately. These two variants are detailed below.

Combined subroutine matching

Here consider combining the subroutines across all malware programs. To summarize a program p, we define the variable

$$N_k^{(\alpha)}(p) = \sum_{u \in U_k} \sum_{v \in p} I(s(u, v) \geq \alpha), \tag{3}$$

where U_k are all malware subroutines that were matched exactly k times by benign subroutines in the training data. Thus, $N_0^{(\alpha)}(p)$ is the number of subroutines in p that match subroutines that were not matched in the training set, $N_1^{(\alpha)}(p)$ denotes the number of matches to subroutines that were matched only once in the training set, and so on. The probability of N_k exceeding a particular value clearly depends on the number of subroutines in p. Empirical investigation revealed, not surprisingly, that the expectation of

$N_k^{(\alpha)}(p)$ scaled approximately linearly with the number of subroutines in p, denoted n_p. We therefore use the feature vector

$$(N_0^{(\alpha)}(p), N_1^{(\alpha)}(p), .., N_K^{(\alpha)}(p))/n_p \tag{4}$$

as the basis for classifying program p.

The feature vector in (4) can be used as input to a classifier, such as a linear logistic regression or a tree boosted classifier. A somewhat different approach is to view the classification problem as a hypothesis test. In this setting, each $N_k^{(\alpha)}(p)/n_p$ can be thought of as a test statistic, with its null distribution being its distribution on benign programs. To combine across multiple test statistics, p-value combination [14] can be used. Letting $z_{k,p}$ denote the realized value of $N_k^{(\alpha)}(p)/n_p$, then the associated p-value is $P(N_k^{(\alpha)}(p^*)/n_p \geq z_{k,p})$ where p^* is a randomly sampled benign program. A combined test statistic can be formed by, for instance, taking the minimum across the p-values, or using some other combination function, such as Fishers method [14].

Program specific matching

The above approach treats the subroutines across all malware programs equally. It can, and appears to be the case from Fig. 2, that the probability of matching varies across the malware programs. For instance, for the program plotted with dotted lines in Fig. 2, well over half of the subroutines were matched by subroutines in the 1700 benign programs. However, for the program program plotted with dashed lines, only 1/6-th of the subroutines were matched. It may therefore be sensible to define program specific matching statistics. To do so we let $N_{k|m}^{(\alpha)}(p)$ denote the number of subroutines in malware program m that are matched by subroutines from p and were matched k times in the training sample. The feature vector for p is then the vector of $N_{k|m}^{(\alpha)}(p)$ for $k = 1, .., K$ and $m \in M$.

To implement the p-value combination approach with the $N_{k|m}^{(\alpha)}(p)$ statistics, we first form a p-value of each $N_{k|m}^{(\alpha)}(p)$, and take the minimum across these for each malware program m and then finally across all $m \in M$. This minimum forms the basis for classification. The null-distribution of each $N_{k|m}^{(\alpha)}(p)/n_p$ is modelled as a mixture of an exponential and a point mass at zero. The point mass probability is estimated using one minus $(\sum_{p\in B} I(N_{k|m}^{(\alpha)}(p) > 0) + 1)/(|B| + 1)$, with the mean of the exponential set to the observed mean of $N_{k|m}^{(\alpha)}(p)/n_p$ for observations with $N_{k|m}^{(\alpha)}(p) > 0$.

4.2 Rank based classifiers

One possible improvement over the threshold matching approach is to include information on the ranking of a subroutine similarity. To do this, let $s_k(u)$ denote the k-th largest similarity observed in the benign training sample to malware subroutine u. We define

$$R_k^{(\alpha)}(p) = \sum_{v\in p}\sum_{u\in M} I\{\max(s_k(u), \alpha) \leq s(u, v) \leq s_{k-1}(u)\}, \tag{5}$$

with $s_0(u) \equiv 1$. Thus, $R_1^{(\alpha)}(p)$ is the number of malware subroutines that are more similar to subroutines in program p than to any of the subroutines in the benign training set. The expected value of $R_k^{(\alpha)}(p)$ scales approximately with the number of subroutines in p, and the feature vector

$$(R_1^{(\alpha)}(p), R_2^{(\alpha)}(p), ..., R_K^{(\alpha)}(p))/n_p, \tag{6}$$

is used for classifying program p.

This rank based approach can also be applied in a program specific manner. In which case we have the statistics $R_{k|m}^{(\alpha)}(p)$ $k = 1, .., K$ giving the rank statistics for program p associated with malware program m. Classifiers using the rank based features can be formed in a manner similar that for the threshold matching features. That is, the feature vector (6) can be used in, for instance, a logistic regression classifier, or the hypothesis testing approach described in Sect. 4.1 can be used by treating the $R_2^{(\alpha)}(p)/n_p$ quantities as test statistics.

4.3 Logistic regression classifier

The malware subroutines $u \in U$ can be used to form a feature vector for a program p. In particular, for a malware subroutine u, the distance to the closest subroutine of a program p can be found, i.e. $d_{p,u} \equiv \min_{v\in p} d(v, u)$, and the vector defined by $d_{u,p}$ $u \in U$ can be used for classification. Here the malware subroutines across all of the available APT programs were used to form the feature vector. A logistic regression model was used, and flexibility was added using the transformation $x_{p,u}(\theta) = \exp(-\theta \cdot d_{p,u})$ where $\theta > 0$ is a parameter to be estimated. This leads to the model

$$\Pr(p \text{ is malicious}) = \left\{1 + \exp\left(-\beta_0 - \sum_{u\in M} \beta_u \cdot x_{p,u}(\theta)\right)\right\}^{-1}. \tag{7}$$

There were around 7000 unique APT subroutines, and penalized regression methodology was used to fit (7). In particular, we used elastic net logistic regression methodology of [32], implemented in the *R* package *glmnet* by [8].

4.4 Naive Bayes classifier

The final classifier we consider is based on the Naive Bayes approach. In particular, we define E_u to be the event that

$I(\max_{v\in p} s(v,u) \geq \alpha) = 1$ for $u \in U$, i.e. that there is a subroutine in program p whose similarity to malware subroutine u exceeds α. Classification is then based on an estimate of the probability

$$\Pr(p \text{ is malicious} | E_{u_1}, E_{u_2}, ..). \tag{8}$$

which is the probability that p is malware given that it matches the malware subroutines $u_1, u_2, ...$ Treating the E_u events as independent, the Naive Bayes assumption, implies that the logit transform of (8) is

$$\beta_0 + \sum_{k=1}^{K_p} \log \frac{\Pr(E_{u_k}|p \text{ is malicious})}{\Pr(E_{u_k}|p \text{ is benign})}, \tag{9}$$

where $K_p = \sum_u I(\max_{v\in p} s(v,u) \geq \alpha)$.

To form (9), estimates of $\Pr(E_u|p \text{ is malicious})$ and $\Pr(E_u|p \text{ is benign})$ are required. For this task observed frequencies are used, setting $\Pr(E_u|p \text{ is malicious})$ to the fraction of malware programs where event E_u was observed. The probability $\Pr(E_u|p \text{ is benign})$ was treated similarly, however special care was taken for the malware subroutines that were not matched by any of the benign programs. For such subroutines, the probability that they will be matched by a future benign program can be obtained using a non-parametric empirical Bayes result due to Robbins and Good, see [6]. In particular, let n_1 denote the number of malware subroutines that were matched only once in the benign training set, and N the total number of malware subroutines that were matched, including repetitions. In a new benign set of equal size as the benign training set, the fraction of subroutine matches that will come from the previously unmatched subroutines can be estimated n_1/N. Dividing this estimate by the number of benign programs as well as the number of unmatched subroutines gives an average probability that a specific one will be matched by a subroutine from a randomly selected benign program. This value is used here as an estimate of $\Pr(E_u|p \text{ is benign})$ when E_u has not been observed.

5 Application

5.1 Matching and rank based classifiers

Here we investigate how to best make use of the threshold matching and rank-based features described in Sect. 4. The feature vector corresponding the rank-based features for a program p is

$$(R_1^{(\alpha)}(p), R_2^{(\alpha)}(p), ..., R_K^{(\alpha)}(p))/n_p, \tag{10}$$

with a similar vector for the threshold matching features. This vector depends on two parameters α and K, and sensible values of these need to be determined. Further, given these feature vectors, a sensible classification strategy needs to be found.

Table 1 Threshold matching and rank based classifiers. The table gives the true positive rates, at a 5 in 1000 false positive rate, associated with different classifiers using either matching or rank based features

Method	$\alpha = 0.8$	$\alpha = 0.9$	$\alpha = 0.99$
minP(N_1)	0.936	0.959	0.963
minP($N_{1:2}$)	0.931	0.954	0.961
minP($N_{1:3}$)	0.930	0.948	0.956
minP(R_1)	0.963	0.958	0.964
minP($R_{1:2}$)	0.960	0.956	0.961
minP($R_{1:3}$)	0.957	0.950	0.957
boost($R_{1:10}$)	0.958	0.953	0.959
logistic($R_{1:10}$)	0.941	0.937	0.943

$N_{1:K}$ denotes the features of the first K matching statistics, similar for $R_{1:K}$. Results are based on 40 repetitions of sampling 50 % of the malware and 90 % of the benign programs

Table 1 summarizes the results of an investigation into these issues. There three types of classifiers are referenced. The first is based on treating the $N^{(\alpha)}$ and $R^{(\alpha)}$ vectors as test statistics. Computing the p-value for each element of these vectors and then forming a combined score using the minimum across these p-values, the so-called minP combination method. This approach was applied separately to the $N^{(\alpha)}$ and $R^{(\alpha)}$ vectors using an increasing value of K. It is seen from the table that simply using $K = 1$ seems to give the best results for both approaches. For both methods $\alpha = .99$ seems best, and at this value the approaches give almost identical results. In addition to the hypothesis testing approach, the feature vectors with $K = 10$ were used as input to two classifiers. The first was a gradient boosted logistic regression using trees as base-learners, via the R package [18]. The second was a linear logistic regression. Of these two, the boosted tree classifier performed the best, though slightly poorer than the minP method with $K = 1$. These results indicate that a very simple classifier is sufficient for these two feature sets, either $R_1^{(0.99)}(p)/n_p$ or $N_1^{(0.99)}(p)/n_p$.

The simple classifier classifies a program as malicious if $R_1^{(0.99)}(p)/n_p \geq$'threshold'. Figure 3 shows how the false positive rate of this classifier varies with the 'threshold' parameter. It is seen that if 1.5 % of a programs subroutines are highly similar to malware subroutines that previously have not been found similar to benign subroutines, the program can be classified as malicious with about a 1 in 1000 false alarm rate.

5.2 Comparison

Here the results of a comparison between the proposed subroutine based classifiers is given, as well as a comparison with

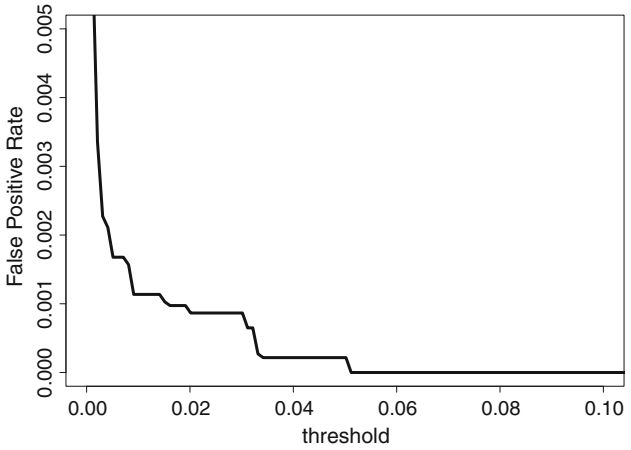

Fig. 3 Cutoff for $R_1^{(\alpha)}$. The figure plots the false positive rate as a function of the threshold used for $R_1^{(0.99)}$ to classify a program as benign or malicious. If $R_1^{(0.99)}/n_p \geq$'threshold' then program is classified as malicious

some overall similarity based classifiers. All classifiers are based only on opcodes. Two variants of the Naive Bayes classifier were implemented. The first used the score obtained (9), the second standardized this score by dividing by the number of subroutines in the a program.

Two overall similarity approaches are investigated. One uses 2-gram to summarize a program, the other a standardized variant that treats the opcodes as a Markov chain, summarizing the program using the transition matrix, see [27]. Letting $o(p)$ denote the count vector associated with the 2-gram, the distance between p and a program q is computed using $||o(p) - o(q)||_1$. The distance between the Markov transition probabilities is computed similarly. Two classifiers using these features sets are used, a kNN classifier and an SVM. The latter requires a similarity matrix which is formed by the entries $\exp\{-\nu||o(p) - o(q)||_1\}$, $\nu > 0$. All parameters associated with these approaches were estimated using cross-validation approaches.

The malware programs are divided into 37 families and we first consider the out-of-family classification accuracy. This comparison was done by 40 repetitions of first selecting 90 % of the benign programs for training, and then training 37 classifiers each time holding out all members of a family. For each repetition the error rates were computed, and these were then averaged.

The results for the out-of-family classification are given in Table 2. At the lowest false alarm rate of 0.1 % the best subroutine based classifier was $R_{1|M}^{(.99)}$, obtained by minimum p-value combination of program specific $R_1^{(.99)}$ statistics, giving 88.7 % true positive rate. The unstandardized naive bayes method and the logistic regression performed less well, however the standardized naive bayes approach performed well. The best performing overall similarity based

Table 2 Out-of-family classification

Method	0.1 % FAR	0.5 % FAR	1 % FAR
Local			
$R_1^{(.99)}$	0.845	0.896	0.907
$R_{1\|M}^{(.99)}$	0.887	0.908	0.908
$N_1^{(.99)}$	0.845	0.896	0.907
naive bayes st.	0.812	0.886	0.894
naive bayes unst.	0.330	0.689	0.824
logistic regression	0.608	0.620	0.635
svm(markov)	0.849	0.941	0.953
svm(ngram)	0.540	0.863	0.899
knn(markov)	0.547	0.776	0.854
knn(ngram)	0.432	0.635	0.719
Combined			
$R_1^{(.99)}$ +svm(markov)	0.889	0.965	0.975
$R_{1\|M}^{(.99)}$ +svm(markov)	0.933	0.974	0.984

The table gives the true positive rates for various classifiers at three different False Alarm Rates (FAR)

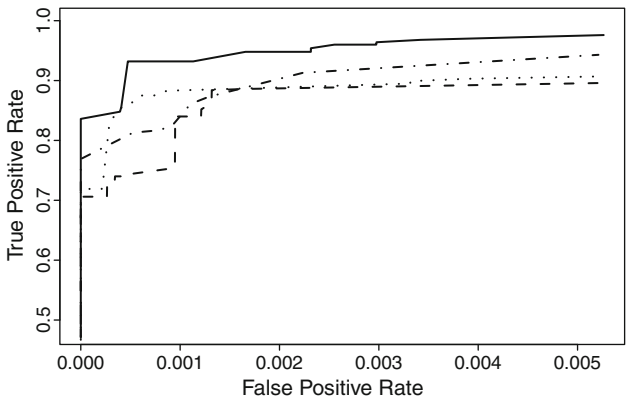

Fig. 4 ROC curves. The plot shows the ROC curves for four different classifiers in the out-of-family comparison. The dashed curve corresponds $R_1^{(.99)}$, the dotted curve corresponds $R_{1|M}^{(.99)}$, the dashed-dotted curve corresponds the svm classifier using Markov chain opcode representation, and the solid curve corresponds the classifier combining $R_{1|M}^{(.99)}$ with the svm classifier

classifier was the SVM using the the markov chain opcode representation, giving a true positive rate of 84.9 %. There were significant differences between which malware programs were misclassified by the SVM approach and the $R_{1|M}^{(.99)}$ and $R_1^{(.99)}$ based classifiers, suggesting that a combination of the two approaches could be beneficial. The results of this combination are given in the table. In particular, combining the SVM approach with $R_{1|M}^{(.99)}$ gave a true positive rate of 93.3 % at a 1 in 1000 false alarm rate, a significant improvement over both approaches individually. The ROC curves for four of the classifiers are given in Fig. 4. These classifiers were $R_1^{(.99)}$, $R_{1|M}^{(.99)}$, the SVM Markov chain, and the clas-

sifier combining the latter two. The (partial) area under the curve statistics for these were 0.860, 0.884, 0.885 and 0.948, respectively.

6 Discussion

This article has introduced a new approach to malware classification that is somewhat intermediary between the byte sequence signatures often used by Anti-Virus software and the overall similarity approach to statistical classification. The results show that statistical detection of APT malware is promising, and in particular that focusing on malware subroutine similarity produces accurate and simple classifiers.

An important question regarding the proposed methodology is its robustness to obfuscation attempts. Table 1 sheds some light on this question, where the detection rates of the various classifiers are given for different levels of the similarity threshold α. This parameter determines the threshold at which two subroutines are considered similar. It is seen that detection rates are qualitatively similar for $\alpha \geq 0.8$, suggesting that the proposed methods might be robust to moderate degrees of obfuscation. However, experiments with α set substantially below 0.8 resulted in considerably poor detection.

The article has, for sake of simplicity, only considered opcode sequences as a basis for subroutine similarity. However, API calls also carry a significant information regarding the functionality of a given piece of code [24]. Including such features could improve detection, and might improve robustness against obfuscation attempts. Further, we have classified programs by treating them as a bag-of-subroutines, ignoring any links between the subroutines, such as those depicted in Fig. 1. Including such information might also increase detection accuracy. A further direction for future research would be to apply the idea of local similarity based detection to data obtained via dynamic analysis. One advantage of this approach is that it circumvents the disassembly process, which can be challenging for some packed binaries.

Acknowledgments The authors would like to thank three reviewers whose comments resulted in a considerably improved manuscript.

References

1. Anderson, B., Quist, D., Neil, J., Storlie, C., Lane, T.: Graph-based malware detection using dynamic traces. J. Comput. Virol. **7**, 247–258 (2011)
2. Baysa, D., Low, R.M., Stamp, M.: Structural entropy and metamorphic malware. J. Comput. Virol. Hack. Tech. **9**, 179–192 (2013)
3. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: Proceedings of the 12th USENIX Security Symposium, USENIX, pp. 169–186 (2003)
4. Deng, W., Liu, Q., Cheng, H., Qin, Z.: A malware detection framework based on Kolmogorov complexity. J. Comput. Inf. Syst. **7**, 2687–2694 (2011)
5. Eagle, C.: The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler, 2nd edn. No Starch Press, San Francisco (2011)
6. Efron, B.: Robbins, empirical Bayes and microarrays. Ann. Stat. **31**, 366–378 (2003)
7. FireEye.: Supply chain analysis: from quartermaster to SunShop-FireEye. www.fireeye.com/content/dam/fireeye-www/global/en/current-threats/pdfs/rpt-malware-supply-chain (2013). Accessed 19 Nov 2015
8. Friedman, J., Hastie, T., Tibshirani, R.: Regularization paths for generalized linear models via coordinate descent. J. Stat. Soft. **33**, 1–22 (2010)
9. Iwamoto, K., Wasaki, K.: Malware classification based on extracted API sequences using static analysis. In: Proceedings of the Asian Internet Engineering Conference, ACM pp. 31–38 (2012)
10. Jidigam, R.K., Austin, T. H., Stamp, M.: Singular value decomposition and metamorphic detection. J. Comput. Virol. Hack. Tech. **11**, 203–216 (2015)
11. Kinable, J., Kostakis, O.: Malware classification based on call graph clustering. J. Comput. Virol. **7**, 233–245 (2011)
12. Kolter, J.Z., Maloof, M.A.: Learning to detect and classify malicious executables in the wild. J. Mach. Learn. Res. **7**, 2721–2744 (2006)
13. Lai, A., Wu, B., Chiu, J.: Balancing the PWN trade deficit series: APT secrets in Asia. http://www.defcon.org/images/defcon-19/dc-19-presentations/Lai-Wu-Chiu-PK/DEFCON-19-Lai-Wu-Chiu-PK-APT-Secrets-2 (2011). Accessed 19 Nov 2015
14. Loughin, T.M.: A systematic comparison of methods for combining p-values from independent tests. Comput. Stat. Data Anal. **47**, 467–485 (2004)
15. Mandiant. APT1: exposing one of China's cyber espionage units. http://intelreport.mandiant.com/Mandiant_APT1_Report (2013). Accessed 19 Nov 2015
16. Microsoft Corporation.: Microsoft security intelligence report, January-June 2006. http://www.microsoft.com/en-us/download/details.aspx?id=5454 (2006). Accessed 19 Nov 2015
17. Park, Y., Reeves, D., Stamp, M.: Deriving common malware behavior through graph clustering. Comput. Secur. **39**, 419–430 (2013)
18. Ridgeway, G.: gbm: Generalized Boosted Regression Models. R package version 2.1 (2013)
19. Reddy, D., Pujari, A.: N-gram analysis for computer virus detection. J. Comput. Virol. **2**, 231–239 (2013)
20. Runwal, N., Low, R.M., Stamp, M.: Opcode graph similarity and metamorphic detection. J. Comput. Virol. **8**, 37–52 (2012)
21. Ruttenberg, B., Miles, C., Kellog, L., Notani, V., Howard, M., LeDoux, C., Lakhtia, A., Pfeffer, A.: Identifying shared software components to support malware forensics. In: Detection of Intrusions and Malware, and Vulnerability Assessment pp. 21–40 (2014)
22. Santos, I., Breze, F., Nieves, J., Penya, Y.K., Sanz, B., Laorden, C., Bringas, P.G.: Idea: Opcode-sequence-based malware detection. In: Engineering Secure Software and Systems, pp. 35-43, Springer, Berlin (2010)
23. Toderici, A.H., Stamp, M.: Simple substitution distance and metamorphic detection. J. Comput. Virol. Hack. Tech. **9**, 159–170 (2013)
24. Sikorski, M., Honig, A.: Practical Malware Analysis: A Hands-on Guide to Dissecting Malicious Software. No Starch Press, San Francisco (2012)
25. Sorokin, I.: Comparing files using structural entropy. J. Comput. Virol. **7**, 259–265 (2011)
26. Sridhara, S.M., Stamp, M.: Metamorphic worm that carries its own morphing engine. J. Comput. Virol. Hack. Tech. **9**, 49–58 (2013)

27. Storlie, C., Anderson, B., Vander Wiel, S., Quist, D., Hash, C., Brown, N.: Stochastic identification of malware with dynamic traces. Ann. Appl. Stat. **8**, 1–18 (2014)
28. Tankard, C.: Persistent threats and how to monitor and deter them. Netw. Secur. **8**, 16–19 (2011)
29. Toderici, A.H., Stamp, M.: Chi-squared distance and metamorphic virus detection. J. Comput. Virol. Hack. Tech. **9**, 1–14 (2013)
30. Wong, W., Stamp, M.: Hunting for metamorphic engines. J. Comput. Virol. **2**, 211–229 (2006)
31. Xu, M., Wu, L., Qi, S., Xu, J., Zhang, H., Ren, Y., Zheng, N.: A similarity metric method for obfuscated malware using function-call graph. J. Comput. Virol. Hack. Tech. **9**, 35–47 (2013)
32. Zou, H., Hastie, T.: Regularization and variable selection via the elastic net. J. R. Stat. Soc. B Meth. **67**, 301–320 (2005)