Cost-Sensitive Detection of Malicious Applications in Mobile Devices

Yael Weiss[1,2], Yuval Fledel[1,2], Yuval Elovici[1,2], and Lior Rokach[1,2]

[1] Department of Information Systems Engineering,
Ben-Gurion University of the Negev, Be'er Sheva 84105, Israel
[2] Duetsche Telekom Laboratories at Ben-Gurion University, Israel
{wiessy,fledely,elovici,liorrk}@bgu.ac.il

Abstract. Mobile phones have become a primary communication device nowadays. In order to maintain proper functionality, various existing security solutions are being integrated into mobile devices. Some of the more sophisticated solutions, such as host-based intrusion detection systems (HIDS) are based on continuously monitoring many parameters in the device such as CPU and memory consumption. Since the continuous monitoring of many parameters consumes considerable computational resources it is necessary to reduce consumption in order to efficiently use HIDS. One way to achieve this is to collect less parameters by means of cost-sensitive feature selection techniques. In this study, we evaluate ProCASH, a new cost-sensitive feature selection algorithm which considers resources consumption, misclassification costs and feature grouping. ProCASH was evaluated on an Android-based mobile device. The data mining task was to distinguish between benign and malicious applications. The evaluation demonstrated the effectiveness of ProCASH compared to other cost sensitive algorithms.

Keywords: Intrusion Detection, Mobile Devices, Malware, Security, Android, sCost sensitive feature selection.

1 Introduction

Smart mobile phones have become a primary communication device for many individuals. In 2008, the converged mobile device segment outpaced the rest of the industry growing 22.5% compared to 2007[1]. Integrating the traditional functionality of mobile phones with special computer-enabled features not previously associated with telephones, smart phones require various security solutions in order to maintain their proper functionality and to protect against malicious behavior. Many of these solutions have migrated from desktop computers where they were initially introduced.

Some of the more sophisticated solutions, such as host-based intrusion detection systems (HIDS), continuously monitor many parameters in the device. Data mining techniques are then applied to the collected data in order to detect abnormal states. As opposed to desktop devices which have evolved over the years into robust instruments with massive resources, mobile devices are constraint-based devices since they are limited, mainly in battery power but in memory and CPU as well. Therefore, since the

M. Griss and G. Yang (Eds.): MOBICASE 2010, LNICST 76, pp. 382–395, 2012.

continuous monitoring of many parameters consumes considerable computational resources it is necessary to reduce consumption in order to efficiently use HIDS on mobile devices. One way to reduce the power consumption is to monitor those features whose acquisition requires less power while maintaining the data mining process performance. To implement this task it is necessary to make a smart decision on which variables to monitor.

Another important aspect involved in determining which subset of features to monitor is feature grouping which arises occasionally when feature costs vary with the choice of a prior feature. For instance, let us assume that the raw data about two feature values lies on the same sector on a rotating disk. In order to monitor the first feature value we need to pay two cost units. The first cost unit is for uploading the page from the hard disk to the main memory; the second supplementary cost unit is paid in order to read the raw data from the main memory and then to perform the required calculations to induce the feature value from the raw data. Then, in order to monitor the second feature, we only need to pay the second supplementary cost unit since the page is already in main memory. Hence, information about the feature grouping must be tailored into the feature selection process.

In addition to the resource costs, when deciding which features subset to monitor, it is also necessary to consider misclassification costs. In some applications the cost of false positive (FP) and false negative (FN) costs bear a different penalty. For example, if the false positive (FP) cost is substantially higher than the false negative (FN) cost, we would prefer to sample features that would reduce the false positive (FP) errors over features that reduce the false negative errors.

In this study, we evaluate ProCASH, a new cost-sensitive feature selection algorithm sensitive to the resources, and misclassification costs as well as feature grouping. ProCASH was evaluated on an Android-based mobile device and the data mining task focused on smart phone security. Extensive experimentation s demonstrated the effectiveness of ProCASH compared to other algorithms.

The rest of the paper is structured as follows. In section 2 we present related work; in section 3 we introduce the ProCASH cost sensitive feature selection algorithm; in section 4 we describe the datasets we used for the evaluation and the evaluation results; section 5 presents a summary and concluding remarks.

2 Related Work

Cost-sensitive learning is an essential task in several real-world applications. Turney et al.[2] presented a taxonomy of the main types of costs involved in inductive concept learning. Two costs, misclassification cost and test cost (which is equivalent to the resource consumption which is consumed when a feature is being monitored) are particularly relevant to this paper.

Turney[3] was the first to consider both test and misclassification costs. Turney's approach presents the inexpensive classification with an expensive test (ICET) system. ICET uses a method that employs a hybrid approach, combining a greedy search heuristic (decision tree) with a genetic algorithm for building a decision tree that minimizes the total cost objective which is composed of test and misclassification

costs. Furthermore, ICET considers feature grouping. Although the ICET is robust, it is very time consuming as well. Chai et al. [4] offered csNB, a new cost-sensitive classifier based on a naïve Bayes algorithm. Several works, such as [8][10] use a cost-sensitive decision tree for the classification task. Ling et al.[5] proposed a decision tree algorithm which applies a new splitting criterion, minimal total cost, to training data instead of the well known minimum entropy measurement. In another paper, Sheng et al. [6] offered a framework where a decision tree is built for each new test case. Ling et al. [7]subsequently updated their strategy for building cost-sensitive decision trees by incorporating possible discounts when obtaining the values of a group of attributes with missing values in the tree building algorithm. Sheng et al. [9] suggested a hybrid cost-sensitive decision tree, DTNB that reduces the minimum total cost by integrating the advantages of a cost-sensitive decision tree and those of the cost-sensitive naïve Bayes. While it uses the cost-sensitive decision tree in order to decide which tests to choose, for the classification task it uses the cost-sensitive naïve Bayes. Freitas et al. [10] suggest a new splitting criterion in building decision trees that considers different aspects of test costs. They examine several cost-scale factors that regulate the influence of tests costs as it can make trees more sensitive to tests costs. They also suggest how to embed the risk cost for performing the test in the new cost-sensitive splitting criterion. The risk cost captures the change in the quality of life due to performing these tests on the patient. However, no experiments were carried out in regard to risk cost.

3 ProCASH- A Cost-Sensitive Algorithm

In this paper we introduce ProCASH, a cost-sensitive algorithm which takes as its starting point a similar preprocessing step as the CASH algorithm. However, unlike the CASH algorithm, ProCASH does not assume that all cost types have the same cost units; While CASH evaluation measurement is the summation result of the different costs, ProCASH evaluation metric is, the achieved average misclassification cost given a maximal budget of resource cost. Furthermore, in contrast to CASH which used the genetic algorithm as its search algorithm, ProCASH uses a new search algorithm.

CASH [11] is a cost-sensitive feature selection method which uses a new fitness function based on comparing histograms. This algorithm follows the filter approach. The CASH algorithm takes into account resource costs as well as feature grouping and misclassification costs. The

CASH algorithm consists of four main steps: preprocessing; creating an initial population of individuals; computing the fitness of each individual; and applying a genetic algorithm to the initial population. CASH assumes that all types of costs are given in the same scale and therefore, it uses as an evaluation metric the average total cost which is composed of the summation of the average misclassification and resource cost.

CASH's preprocessing step is composed of four sub-steps. In the first sub-step, CASH computes the average a priori cost which indicates when a features subset should not be obtained. That is to say, if the average a priori cost, computed according

to the distribution of classes in the training set, is lower than the average misclassification cost achieved by the features subset, CASH will not choose this subset. Then, in the second sub-step CASH computes histograms for each class value of each feature in the training dataset. In the third sub-step, CASH computes for each feature how it classifies the records in the training dataset. This computation is based on a cost- sensitive majority rule that classifies all the records in a certain bin to the class which minimizes the misclassification cost. Finally, in the fourth sub-step, CASH calculates for each feature the misclassification cost ratio it assigns to each record in the training dataset. The motivation for calculating the misclassification cost ratio of a certain feature is to supply the algorithm with the knowledge of whether or not the decision is sufficiently distinctive. That is to say, based on the distribution of the classes in a certain bin of a feature, the CASH algorithm tries to estimate what is the likelihood that the algorithm's classification was correct.

As opposed to CASH, ProCASH adds an alteration to the preprocessing step by changing the way that the classification decision of each feature to each record is carried out. For each feature and record, ProCASH checks if the misclassification cost ratio is larger than a certain threshold. If so, ProCASH add the record to a list of records that have not been correctly classified by that feature. Furthermore, in contrast to CASH which used the genetic algorithm as its search algorithm, ProCASH uses a new search algorithm. In the beginning of the search, ProCASH's search method first looks for an initial subset with which it starts the search by performing the two following steps. Firstly, it gathers each feature in the training set that: (1) has classified correctly more than a predefined threshold of the training set's records; (2) whose feature resource cost is no higher than the resource constraint; and (3) whose average misclassification cost of all the records in the dataset is lower than the average a priori cost Secondly, ProCASH builds the initial selected features subset by iteratively selecting features with the minimal average misclassification cost from the features that were gathered in the previous step. ProCASH continues to add features to the initial subset until the point is reached where the addition of an extra feature causes the initial chosen subset's resource cost to exceed the resource constraint.

Then, after the initial feature subset necessary to start the search has been chosen, ProCASH computes a list of all the records that have not been correctly classified by at least one of the chosen subset features. ProCASH then starts the search with the initial chosen subset by iteratively adding to the chosen subset one feature which holds the minimum misclassification cost. ProCASH continues to add features to the chosen subset as long as: (1) the number of records that have not been correctly classified by at least one feature in the chosen subset is larger than five percent of the records in the training set and (2) the chosen subset's features resource cost does not exceed the resource constraint. Finally, ProCASH returns as output the chosen subset of features.

4 Experiments

In this section we present and analyze empirical results obtained from evaluating a cost-sensitive malware detection framework designed for Android devices [12]. Our

goal was to explore malware detection when using ProCASH in comparison to several other cost-sensitive algorithms, all constrained by a specific CPU consumption budget cost.

ProCASH was implemented in Java. In order to evaluate the performance of ProCASH, we compared it to four algorithms: csDT_csf1 classifier[10], csDT_csf2 classifier[10], GA+META+CsId3[3] and GA+META_ICF[3]. The csDT_csf is a classifier which follows the embedded approach. csDT_csf employs a cost-sensitive decision tree to obtain a setting for the cost-scale factor (csf), that adjusts the strength of the bias towards lower cost attributes. The GA+META_GA+META_ICF is a wrapper algorithm which uses GA+META_ICF [13] as its fitness function and the genetic search as its search algorithm. The GA+META_CsId3 is a wrapper algorithm which uses the same heuristic function as in CSID3[14][15] algorithm and the genetic search as its search algorithm. For each feature selection algorithm (ProCASH, GA+META+ICF and GA+META+CSID3) based on the training set, the algorithm selects a feature subset. Then, features that were not selected were eliminated from the corresponding training and testing set. Afterward, a decision tree was induced on each of the training sets and its performance was evaluated on the corresponding test set. We used the J48 classifier, a Java implementation in WEKA[16] data mining applications of the C4.5 decision tree algorithm that Quinlan[17] introduced. In order to make our classifier cost-sensitive to misclassification costs, we used a meta-learner, implemented in Weka MetaCostClassifier[18].

The rest of this section is composed of the following subsections: In subsection 4.1 we describe the datasets we used; in subsection 4.2 we show the metrics and the statistical tests for measuring the performance of the algorithms and compare them; in subsection 4.3 we describe the experiment plan; in subsection 4.4 we describe the evaluation results followed by a discussion in subsection 4.5.

4.1 Datasets

There are several types of threats targeting mobile malware. In our research we focus on attacks against the phones themselves and not the service provider's infrastructure [34]. Four classes of attacks on a mobile device were identified [6]: unsolicited information, theft-of-service, information theft and denial-of-service (DoS). Since Android is a new platform and there are yet no known instances of Android malware, we developed four applications that perform denial-of-service and information theft.
The first malware we developed was the "Tip Calculator", a calculator which unobtrusively performs a DoS attack. When a user clicks the "calculate" button to calculate the tip, the application starts a background service that waits for a period of time and then launches several hundreds CPU-consuming threads. The attack almost absolutely paralyzes the device. The system becomes very unresponsive and the only effective choice is to shutdown the device (which also takes some time). An interesting observation is that the Android system often kills a CPU-consuming service but always keeps on re-launching it a few seconds later.

Also, we developed three malicious information theft applications. The first malware includes a set of two Android applications exploiting the Shared User ID feature. In Android, each application requests a set of permissions which is granted at

installation time. The Shared User ID feature enables multiple applications to share their permission sets, provided they are all signed with the same key and explicitly request the sharing. It is noteworthy that the sharing is done behind the scenes without informing the user or asking for approval, resulting in implicit granting of permissions. The first Android application is Schedule SMS, a truly benign application that enables one to send delayed SMS messages to people from a contact list for which the application requests necessary permissions. The second application, Lunar Lander, is a seemingly benign game that requests no permissions.

Once both applications are installed and the Lunar Lander obtains the capability to read the contacts and send SMS messages, it exhibits a Trojan-like behavior, leaking all of the contact names and phone numbers through SMS messages to a pre-defined number. This resembles RedBrowser - a Trojan masquerading as a browser that infects mobile phones running J2ME by obtaining and exploiting SMS permissions.

The second information theft application masquerades as a Snake game and misuses the device's camera to spy on the unsuspecting user. The Snake game requests Internet permission for uploading top scores and while depicting the game on the screen, the application is unobtrusively taking pictures and sending them to a remote server.

The third and last malicious information theft application we developed, HTTP Upload, also steals information from the device. It exploits the fact that access to the SD-card does not require any permission. Therefore, all applications can read and write to/from the SD-card. The application requires only Internet permission and in the background it accesses the SD card, steals its contents and sends it through the Internet to a predefined address.

The small number of malicious applications left us with a class imbalance problem. The class imbalance problem occurs when one of the classes is represented by a very small number of cases compared to the other classes. This problem has been recognized as a crucial problem in machine learning and data mining since it causes serious negative effects on the performance of standard learning methods (which assume a balanced distribution of the classes). Several solutions to the imbalance problem have been proposed: assigning distinct costs to the classification errors; internally biasing the discrimination-based process; re-sampling the original data set (either by over-sampling the minority class and/or under-sampling the majority class) until the classes are approximately equally represented, or by duplicate vectors from the minority class. We decided to cope with this problem by under-sampling the benign class, i.e., using only part of the benign applications that were generated for the first set of experiments and duplicating the vectors of malicious applications.

The four malicious applications were installed on two Android devices. A monitoring application, which continuously sampled various features on the device, was installed and activated on the devices. The conditions were regulated and measurements were logged on the SD-card. Three of the four malicious applications (Tip Calculator, Snake, and HTTP Uploader) were used for 10 minutes by each user, while in the background the application collected feature vectors every 2 seconds. The Lunar Lander game was not used for 10 minutes. The only malicious functionality of the Lunar Lander game was to send SMSs to a predefined destination. Therefore, when the Lunar Lander game was the only application used on the device, we expected that all of the vectors would be identical. Hence, we decided to sample this

application once during the short period of the attack and the sampled. These feature vectors were then duplicated and aligned with the number of vectors from the rest of the applications.

In addition to malicious applications, 20 benign tool applications from the Android framework and Android market were used. All the benign tool applications were verified to be virus-free before installation by manually exploring the permissions that the applications required and by using a static analysis of .dex files. Each of the two Android devices had one user who used each of the 20 applications for 10 minutes; in the background, the monitor application collected new feature vectors every 2 seconds. All the vectors were labeled with their true class: 'tool' or 'malicious'.

Then, for each device, five dataset were generated. To create the 5 datasets for each device, we divided randomly the 20 tool application that were used for the first sub experiment to 5 groups of size 4, while none of the tools overlapped across the different groups. For each device, the feature vectors of each group were added to a different dataset out of the 5 datasets of the device. The feature vectors for the malicious application were collected and added to each one of the 10 datasets (5 dataset for each user of the two users).The reason for choosing only 4 tool applications is to guarantee that the different classes are equally represented in each dataset. After the division we had 10 groups, 5 for each device.

Table 1 indicates the CPU power consumption costs, with and without group discount, of each feature when it is being monitored by the monitoring application. Costs were estimated using the CPU profiler which can be found in Android's SDK

The first column represents the different features that the agent extracted from the device. The extracted features are clustered into two primary categories: Application Framework and Linux Kernel. Features belonging to groups such as Messaging, Phone Calls and Applications belong to the Application Framework category and were extracted through APIs provided by the framework. Features from such groups as Keyboard, Touch Screen, Scheduling and Memory belong to the Linux Kernel category. Some Linux Kernel features can be extracted directly from Java. For example, Memory and Scheduling parameters can be accessed directly through a special filesystem-based interface of the Kernel and so can be extracted through the usage of ordinary Java I/O classes.

A total of 88 features were collected for each monitored application. The second column presents the feature group assignment indication. There are 15 unique groups. Tests carried out on a group are discounted in terms of CPU power consumption cost. The most common reason that features were in the same group is that they are read from the same file-like source. Therefore, the entire file must read even if only a single feature is needed. However, once we paid the reading cost for the first feature, we no longer need to pay the reading cost for a second feature on the same file. As an example, all the features in the group Binder were written on the same file and the agent only needed to open the file, which has a common CPU power consumption cost (18.25%) that is shared for all the features in that group. Then, in order to process each one of the features in that file (10 features in total), we only had to pay for each feature an additional CPU power consumption cost of 0.2. The other 14 groups are as follows: Sysfs, Scheduler_Statistics, Load_Average, Virtual_Machine_Statistics, Keyboard_Dynamics, System_configuration, Pressure_Dynamic, Process_Statistcs, Phone_Call, SMS, Logcat, Keyboard, Network and Misc.

Table 1. Feature CPU consumption cost before and after discount and feature grouping

Feature	Group	Before discount cost	After discount cost
Local_RX_Packets	Network	13.2	0.2
Local_RX_Bytes	Network	13.2	0.2
Local_RX_Packets	Network	13.2	0.2
Local_TX_Bytes	Network	13.2	0.2
WiFi_TX_Packets	Network	13.2	0.2
WiFi_TX_Bytes	Network	13.2	0.2
BC_Transaction	Binder	18.45	0.2
BC_Reply	Binder	18.45	0.2
BC_Acquire	Binder	18.45	0.2
BC_Release	Binder	18.45	0.2
Binder_Active_Nodes	Binder	18.45	0.2
Binder_Total_Nodes	Binder	18.45	0.2
Binder_Ref_Active	Binder	18.45	0.2
Binder_Ref_Total	Binder	18.45	0.2
Binder_Death_Active	Binder	18.45	0.2
Binder_Death_Total	Binder	18.45	0.2
Binder_Transaction_Active	Binder	18.45	0.2
Binder_Transaction_Total	Binder	18.45	0.2
Binder_Trns_Complete_Active	Binder	18.45	0.2
Binder_Trns_Complete_Total	Binder	18.45	0.2
Free_Pages	Virtual_Machine_Statistics	11.35	0.2
Inactive_Pages	Virtual_Machine_Statistics	11.35	0.2
Active_Pages	Virtual_Machine_Statistics	11.35	0.2
Anonymous_Pages	Virtual_Machine_Statistics	11.35	0.2
Mapped_Pages	Virtual_Machine_Statistics	11.35	0.2
File_Pages	Virtual_Machine_Statistics	11.35	0.2
Dirty_Pages	Virtual_Machine_Statistics	11.35	0.2
Writeback_Pages	Virtual_Machine_Statistics	11.35	0.2
DMA_Allocations	Virtual_Machine_Statistics	11.35	0.2
Page_Frees	Virtual_Machine_Statistics	11.35	0.2
Page_Activations	Virtual_Machine_Statistics	11.35	0.2
Page_Deactivations	Virtual_Machine_Statistics	11.35	0.2
Minor_Page_Faults	Virtual_Machine_Statistics	11.35	0.2
Batery_Voltage	Sysfs	1	0.95
Battery_Current	Sysfs	1	0.95
Battery_Temp	Sysfs	1	0.95
Button_Backlight	Sysfs	1	0.95
Keyboard_Backlight	Sysfs	1	0.95
LCD_Backlight	Sysfs	1	0.95

Table 1. (*continued*)

Battery_Level_Change	Sysfs	1	0.95
Avg_Key_Flight_Time	Keyboard_Dynamics	0.56	0.2
Del_Key_Use_Rate	Keyboard_Dynamics	0.56	0.2
Avg_Trans_To_U	Keyboard_Dynamics	0.56	0.2
Avg_Trans_L_To_R	Keyboard_Dynamics	0.56	0.2
Avg_Trans_R_To_L	Keyboard_Dynamics	0.56	0.2
Avg_Key_Dwell_Time	Keyboard_Dynamics	0.56	0.2
Yield_Calls	Scheduler_Statistics	4.5	0.2
Schedule_Calls	Scheduler_Statistics	4.5	0.2
Schedule_Idle	Scheduler_Statistics	4.5	0.2
Running_Jiffies	Scheduler_Statistics	4.5	0.2
Waiting_Jiffies	Scheduler_Statistics	4.5	0.2
Load_Avg_1_min	Load_Average	2.55	0.2
Load_Avg_5_mins	Load_Average	2.55	0.2
Load_Avg_15_mins	Load_Average	2.55	0.2
Runnable_Entities	Load_Average	2.55	0.2
Total_Entities	Load_Average	2.55	0.2
CPU_Usage	Process_Statistics	13.9	0.2
Running_Processes	Process_Statistics	13.9	0.2
Context_Switches	Process_Statistics	13.9	0.2
Processes_Created	Process_Statistics	13.9	0.2
Outgoing_SMS	Log_Cat	1.25	0.2
Garbage_Collections	Log_Cat	1.25	0.2
Camera	Log_Cat	1.25	0.2
Orientation_Changing	System_Configuration	0.75	0.75
Keyboard_Opening	Keyboard	0.75	0.2
Keyboard_Closing	Keyboard	0.75	0.2
Incoming_SMS	SMS	0.65	0.2
Package_Changing	Misc	0.65	0.2
Package_Restarting	Misc	0.65	0.2
Incoming_Calls	Phone_Calls	2.55	0.2
Outgoing_Calls	Phone_Calls	2.55	0.2
Missed_Calls	Phone_Calls	2.55	0.2
Avg_Touch_Pressure	Pressure_Dynamics	0.25	0.25
Avg_Touch_Area	Pressure_Dynamics	0.25	0.25

4.2 Experiment Plan: Detecting Android Malware in a Cost Sensitive Fashion

In this study, our two types of costs were not measured by the same unit cost. One cost, the CPU power consumption, was measured on a scale of the CPU percentage utilized when monitoring a certain feature from the device. The other cost, misclassification, was measured in dollar ($) terms. Therefore, when given a maximal budget of CPU power consumption cost, our evaluation metric was the achieved average misclassification cost. The lower the evaluation measure value was, the better the algorithm performed.

We used hypothesis tests in order to examine if ProCASH's average misclassification costs and execution times were statistically significant lower than the other algorithms'. We performed an Adjusted Friedman cost hypothesis test [19] with a 5% significance level. If the null hypothesis was rejected, we then conducted a Bonferroni-Dunn post Hoc[19] with a 5% significance level.

The purpose of the experiment was to evaluate the ability of the proposed methods to distinguish between malicious and benign applications given a specific CPU processing power budget. We used 10 datasets extracted from two different devices, evaluated five cost sensitive methods, four misclassification cost matrices and nine CPU processing power budgets. The division of the dataset into training and testing sets was performed in such a way that the benign and malicious applications in the training and testing set were different. For each one of the 10 datasets (generated by selecting 5 times 4 different benign applications for each device), each time a different malicious application and different benign application were not included in the training set but were included in the testing set.

4.3 Experiment Results

Table 2 presents the average misclassification cost obtained in all of the runs. The first column represents the CPU consumption power budget cost. The second column represents the algorithms that were compared. Then, each of the following columns showed the average misclassification cost on the different cost matrices. As can be seen from Table 2, the average misclassification cost of the ProCASH algorithm tends to be better than that of all the other algorithms. ProCASH outperforms csDT_csf2, GA+META_ICF and GA+META_CSID3 algorithms in all of the cost matrices and under all the CPU consumption budgets. Furthermore, ProCASH algorithm outperforms csDT_csf1 in almost all of the cases except three. Tables 3 and 4 present the results of adjusted Friedman and Bonferroni-Dunn statistical hypothesis tests respectively. From Table 3 we can see that the null-hypothesis, that all classifiers perform the same, was rejected using the adjusted Friedman test on all of the misclassification cost matrices. The adjusted Friedman test was conducted with a confidence level of 95%. Table 4 indicates that the ProCASH algorithm significantly outperforms all of the four algorithms at confidence levels of 95% in all of the 4 misclassification cost matrices. Additionally, Figures 1 and 2 represent the performance of each algorithm in each of the CPU budget costs, given a misclassification matrix cost when the FP cost and FN cost are equal to $10 and when the FP cost is equal to $10 and the FN cost is equal to $5 respectively. We can see in Figures 1 and 2 that the ProCASH algorithm outperformed the rest of the algorithms with significant differences in the majority of the datasets. Moreover, we can see that ProCASH performance improved as the budget cost increased, until a certain budget was reached where ProCASH performance stayed the same. This implies that at a certain point, ProCASH decided that sampling more features would cause overfitting to the training set and would not improve the classification performance on the testing set.

Table 2. Comparing cost sensitive algorithms: summary of experimental results

Budget cost	Algorithms	Misclassification cost matrices (FN cost_FP cost)			
		FN=$10 FP=$10	FN=$8 FP=$10	FN=$7.5 FP=$10	FN=$5 FP=$10
1	ProCASH	**$1.72**	**$1.66**	**$1.49**	**$1.50**
1	csDT_csf1	$4.27	$3.95	$2.96	$1.86
1	csDT_csf2	$4.67	$4.21	$3.34	$2.56
1	GA+META_ICF	$3.35	$3.18	$2.85	$1.87
1	GA+META_CSID3	$3.35	$3.16	$2.77	$2.26
5	ProCASH	$1.81	**$1.68**	$1.77	**$1.49**
5	csDT_csf1	**$1.61**	$1.87	**$1.73**	$2.10
5	csDT_csf2	$2.40	$2.46	$2.05	$1.86
5	GA+META_ICF	$2.24	$2.42	$2.34	$2.66
5	GA+META_CSID3	$2.70	$2.24	$2.50	$2.58
10	ProCASH	$1.81	**$1.68**	**$1.71**	**$1.49**
10	csDT_csf1	**$1.61**	$1.87	$1.74	$2.10
10	csDT_csf2	$2.41	$2.47	$2.06	$1.85
10	GA+META_ICF	$2.27	$2.00	$1.78	$2.63
10	GA+META_CSID3	$2.66	$2.54	$2.23	$2.46
15	ProCASH	**$1.21**	**$1.04**	**$0.99**	**$0.94**
15	csDT_csf1	$1.61	$1.87	$1.74	$2.10
15	csDT_csf2	$2.41	$2.47	$2.06	$1.85
15	GA+META_ICF	$2.95	$2.74	$2.55	$2.58
15	GA+META_CSID3	$2.44	$2.12	$2.35	$1.75
20	ProCASH	**$1.21**	**$1.04**	**$0.99**	**$0.94**
20	csDT_csf1	$1.61	$1.87	$1.74	$2.10
20	csDT_csf2	$2.41	$2.47	$2.06	$1.85
20	GA+META_ICF	$2.49	$2.36	$2.39	$2.63
20	GA+META_CSID3	$2.51	$2.99	$3.00	$2.96
25	ProCASH	**$1.21**	**$1.04**	**$0.99**	**$0.94**
25	csDT_csf1	$1.61	$1.87	$1.74	$2.10
25	csDT_csf2	$2.41	$2.47	$2.06	$1.85
25	GA+META_ICF	$2.13	$1.84	$1.79	$2.26
25	GA+META_CSID3	$2.59	$2.71	$2.52	$2.69
30	ProCASH	**$1.21**	**$1.04**	**$0.99**	**$0.94**
30	csDT_csf1	$1.61	$1.87	$1.74	$2.10
30	csDT_csf2	$2.41	$2.47	$2.06	$1.85
30	GA+META_ICF	$3.12	$3.07	$2.64	$3.65
30	GA+META_CSID3	$2.25	$2.06	$1.79	$3.00
35	ProCASH	**$1.21**	**$1.04**	**$0.99**	**$0.94**
35	csDT_csf1	$1.61	$1.87	$1.74	$2.10
35	csDT_csf2	$2.41	2$.47	$2.06	$1.85

Table 2. (*continued*)

35	GA+META_ICF	$2.64	$2.37	$2.32	$2.95
35	GA+META_CSID3	$2.48	$2.66	$2.03	$2.98
40	ProCASH	**$1.21**	**$1.04**	**$0.99**	**$0.94**
40	csDT_csf1	$1.61	$1.87	$1.74	$2.10
40	csDT_csf2	$2.41	$2.47	$2.06	$1.85
40	GA+META_ICF	$2.25	$1.82	$1.94	$2.12
40	GA+META_CSID3	$3.52	$3.19	$2.65	$2.83

Table 2. adjusted Friedman tests results for each one of the misclassification cost matrices

F(4,8) with Critical Value of 2.69				
Misclassification cost (FP-FN)	FN=$10 FP=$10	FN=$8 FP=$10	FN=$7.5 FP=$10	FN=$5 FP=$10
Friedman statistic value	13.6	17.92	18.55	21.72

Table 3. The superscript "+" indicates that the average total cost of ProCASH was significantly higher than the corresponding algorithm at a confidence level of 95%

Average Misclassification cost	FN=$10 FP=$10	FN=$8 FP=$10	FN=$7.5 FP=$10	FN=$5 FP=$10
csDT_csf1	+0.12	+1.61	+2.41	+2.25
csDT_csf2	$^{+}$0.17	$^{+}$0.38	$^{+}$0.28	+0.36
GA+META_ICF	$^{+}$0.12	$^{+}$0.32	$^{+}$3	$^{+}$2.77
GA+META_CSIC3	$^{+}$0.25	$^{+}$0.17	$^{+}$0.38	$^{+}$0.38

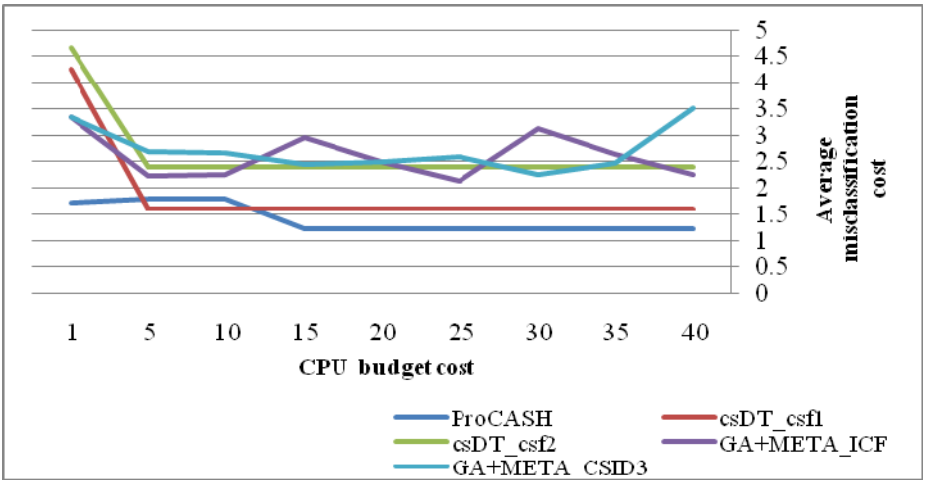

Fig. 1. The average misclassification cost that was obtained for each algorithm in each CPU consumption budget constraint, given a misclassification cost matrices of FP cost=10 and FN cost =$10

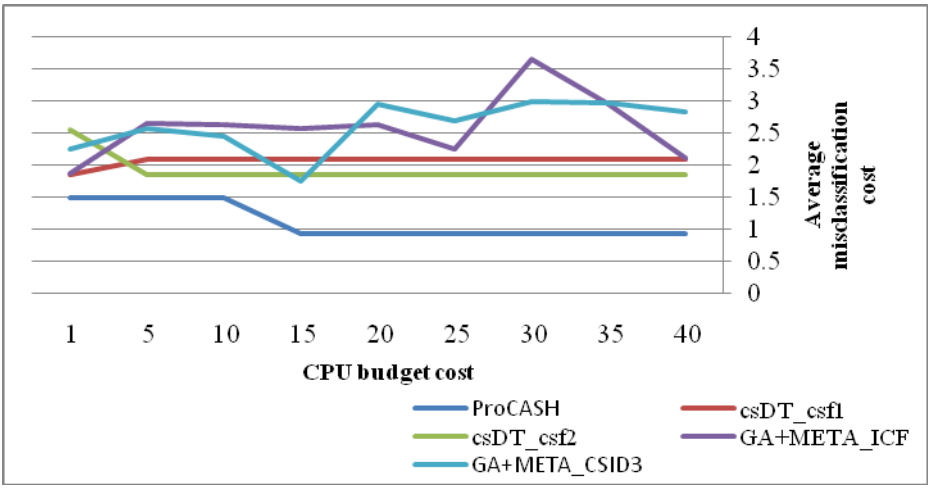

Fig. 2. The average misclassification cost that was obtained for each algorithm in each CPU consumption budget constraint, given a misclassification cost matrices of FP cost=FN cost =$5

4.4 Discussion

The advantages of the new ProCASH algorithm, as the experimental study indicates, can be summarized as follows:

1. When compared to state-of-the-art, cost sensitive algorithms, ProCASH performed better in distinguishing between malicious and benign applications on Android mobile devices under a wide range of CPU resource budget constraints and misclassification cost matrices.
2. Since ProCASH follows the filter approach, it can be used in conjunction with any classification algorithm and not only decision tree classifiers. Potentially, there might be domains in which using other classifiers will dramatically reduce the average misclassification cost.

5 Experiments

Host-based intrusion detection systems (HIDS) are based on continuously monitoring many parameters in the device such as CPU and memory consumption. Then, data mining techniques are applied to the collected data in order to detect abnormal states. Since the continuous monitoring of many parameters consumes considerable computational resources it is necessary to reduce consumption in order to used HIDS on mobile devices.

One way to achieve this is to collect less parameters by means of cost-sensitive feature selection techniques. One way to reduce computational resource consumption is to collect fewer parameters by means of cost-sensitive feature selection techniques. In this study, we evaluated ProCASH, a new cost-sensitive feature selection algorithm sensitive to resource and misclassification costs and feature grouping. ProCASH was evaluated on an Android mobile device. The data mining task was to distinguish between benign and malicious applications. The results indicate that ProCASH

outperforms other cost-sensitive algorithms in terms of average misclassification costs in all of the CPU resource budget constraints.

References

1. http://ems007.iconnect007.net/pages/zone.cgi?a=48033
2. Turney, P.D.: Types of Cost in Inductive Concept Learning. In: Proc. Workshop Cost-Sensitive Learning, 17th Int'l Conf. Machine Learning, pp. 15–21 (2000)
3. Turney, P.D.: Cost-Sensitive Classification: Empirical Evaluation of a Hybrid Genetic Decision Tree Induction Algorithm. J. Artificial Intelligence Research 2, 369–409 (1995)
4. Chai, X., Deng, L., Yang, Q., Ling, C.X.: Test-Cost Sensitive Naive Bayes Classification. In: Proc. 4th Int. Conf. Data Mining, pp. 51–58 (2004)
5. Ling, C.X., Yang, Q., Wang, J., Zhang, S.: Decision Trees with Minimal Costs. In: Proc. 21st Int. Conf. Machine Learning, p. 69 (2004)
6. Sheng, S., Ling, C.X., Yang, Q.: Simple Test Strategies for Cost-Sensitive Decision Trees. In: Proc. 16th European Conf. Machine Learning, pp. 365–376 (2005)
7. Ling, C.X., Sheng, V.S., Yang, Q.: Test Strategies for Cost-Sensitive Decision Trees. IEEE Transactions on Knowledge and Data Engineering 18(8), 1055–1067 (2006)
8. Sheng, V.S., Ling, C.X., Ni, A., Zhang, S.: Cost-Sensitive Test Strategies. In: Proc. 21st Nat'l Conf. Artificial Intelligence (2006)
9. Sheng, S., Ling, C.X.: Hybrid Cost-Sensitive Decision Tree. In: Jorge, A.M., Torgo, L., Brazdil, P.B., Camacho, R., Gama, J. (eds.) PKDD 2005. LNCS (LNAI), vol. 3721, pp. 274–284. Springer, Heidelberg (2005)
10. Freitas, A., Costa-Pereira, A., Brazdil, P.B.: Cost-Sensitive Decision Trees Applied to Medical Data. In: Song, I.-Y., Eder, J., Nguyen, T.M. (eds.) DaWaK 2007. LNCS, vol. 4654, pp. 303–312. Springer, Heidelberg (2007)
11. Weiss, Y., Elovici, Y., Rokach, L.: The CASH Algorithm-Cost-Sensitive Attribute Selection using Histograms. Lecture Notes in information system engineering. Ben-Gurion University (2010)
12. Shabtai, A., Weiss, Y., Kanonov, U., Elovici, Y., Glezer, C.: "Andromaly": An Anomaly Detection Framework for Android Devices. Lecture Notes in information system engineering Ben-Gurion University (2009)
13. Núñez, M.: The use of background knowledge in decision tree induction. Machine Learning 6, 231–250 (1991)
14. Tan, M., Schlimmer, J.: Cost-sensitive concept learning of sensor use in approach and recognition. In: Proceedings of the Sixth International Workshop on Machine Learning, ML 1989, pp. 392–395 (1989)
15. Tan, M.: Cost-sensitive learning of classification knowledge and its applications in robotics. Machine Learning 13, 7–33 (1993)
16. Frank, E., Hall, M.A., Holmes, G., Kirkby, R., Pfahringer, B., Witten, I.H.: Weka: A machine learning workbench for data mining. In: Maimon, O.Z., Rokach, L. (eds.) Data Mining and Knowledge Discovery Handbook. Springer, Heidelberg (2005)
17. Quinlan, J.: C4.5: Programs for machine learning. Morgan Kaufmann, San Francisco (1993)
18. Domingos, P.: MetaCost: A general method for making classifiers cost-sensitive. In: Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining, pp. 155–164 (1999)
19. Botha, R.A., Furnell, S.M., Clarke, N.L.: From desktop to mobile: Examining the security experience. Computer & Security 28, 130–137 (2009)
20. Demšar, J.: Statistical comparison of classifiers over multiple data sets. Journal of Machine Learning Research 7, 1–30 (2006)