Towards effective reproducible botnet detection methods through scientific workflow management systems

Frederico Tosta de Oliveira[1], **Maria Claudia Cavalcanti**[1], **Ronaldo Moreira Salles**[1]

[1]Instituto Militar de Engenharia (IME)
Praca General Tiburcio, 80 – 22290-270 – Rio de Janeiro – RJ – Brasil

{tosta,yoko,salles}@ime.eb.br

Abstract. *Even after nearly two decades of the creation of the first botnet, the detection and mitigation of their attacks remain one of the biggest challenges faced by researchers and cyber-security professionals. Although there are numerous studies related to botnet detection, estimating how much one method is better than another is still an open problem, mainly because of the difficulty in comparing and reproducing such methods. This work proposes an architecture, implemented with Spark as a high-performance data processing solution, together with VisTrails as a workflow management and data provenance solution, to address this comparability and reproducibility problem in a large-scale environment, as well as a tool, ProvTracker, to analyze and compare the methods results. Another contribution is on the way to couple these two technologies so that intermediary data is maintained available during several partial (re)executions of the experiments involved in each method, minimizing the impact on the analysis of the large amount of data involved.*

1. Introduction

In the past, the malware was limited to cause local impacts, but with the creation of the internet, a new opportunity raised for those malicious software, which spread to millions of computers around the world. At the same time, the "hijacking" of computers started, when computers are controlled remotely by a new family of malware called bot. This network of compromised machines, botnet, is then controlled by one or more people, botmasters [Hogben et al. 2015].

There are several methods and techniques to detect botnets [Silva et al. 2013, Garcia et al. 2014b]. Among them, we can highlight the methods based on passive network monitoring, where the most prevalent is the detection based on machine learning techniques, e.g. Decision Tree and Random Forest [Biglar Beigi et al. 2014]. In this technique, a set of features contained in the network flows [Hofstede et al. 2014] is handled by the learning algorithm to determine which flow can be considered malicious, i.e., results from the communication between the botmaster and bots.

Currently, experiments for evaluating botnet detection methods are done in an *ad-hoc* way, making it difficult to register valuable information of all the steps involved in the experiment, such as dataset, programs and algorithms used, as well as which parameters' values were tested, which features of the network flows were used, results achieved, metrics, etc.

Even though most of the activities involved in machine learning technique are commonly used in several approaches, as far as we investigated, there is no tool or environment able to support the entire life cycle of the botnet detection experiment, allowing the researchers to create, execute, and analyze their experiment.

This gap makes it almost impossible to reproduce and compare these experiments' results and, consequently, makes it difficult to select a detection method to deploy in a real production environment. This keeps the solutions restricted to the study environment of their creators, offering no real alternative to the botnets dismantling. In addition, a botnet experiment involves the processing of high volume of data.

There are generic approaches already proposed, which can be useful to support botnet detection experiments. For instance, there are initiatives on managing workflows [Shi Meilin et al. 1998] and provenance capture [Freire et al. 2008]. In addition, there are also initiatives on using high performance platform for provenance capture [Korolev and Joshi 2014, Akoush et al. 2013], which we will highlight in the following sections. However, they are not properly tailored to address the problems mentioned before.

This work specifies and implements an architecture to address botnet detection experiments. The main paper contribution is on the innovative way it combines the functionality of workflow management systems (WfMS), provenance capture and high performance solutions. Specifically, it provides an infrastructure to support the botnet experiments requirements: creation, execution and analysis. It is implemented using VisTrails WfMS and Spark, together with our tool, ProvTracker. Above all, researchers are able to share their experiments, allowing others to reproduce and compare results. After concluding the experiments, the best approach may be used in a production environment.

The remainder of this paper is structured as follows. Next section presents the related works. Section 3 introduces some concepts for better understanding the proposed architecture, which is presented in Section 4. Section 5 describes the architecture implementation and the technologies involved. Sections 6 and 7 present, respectively, some experiments' results and a conclusion pointing to future work.

2. Related Work

In [Aviv and Haeberlen 2011], the authors explore the challenges to carry out the evaluation of botnet detection systems. Many of the challenges are related to the difficulty in obtaining and sharing a real and diverse network traffic dataset. This is justified by the privacy of the data on the traffic, but ends up impacting directly on the inability to perform qualitative comparisons and repeat third party experiments. They also emphasize the lack of methodology to drive botnet detection experiments.

In [Garcia et al. 2014a], the authors present an empirical comparison of three different botnet detection methods. They point out that the results achieved by botnet detection experiments usually do not compare to other methods. Among the existing factors that prevent them to perform the comparison, they highlight the difficulty of obtaining or sharing datasets, lack of an appropriate description of the experiments, lack of comparison methods and error metric. They propose a new metric errors and conclude that a joint effort to create an execution and comparison platform to botnet detection methods can dramatically improve the results achieved in the area.

In [Garcia et al. 2014b], the authors present a review on botnet detection methods.

They compare and classify fourteen botnet detection methods and highlight that ten could not be reproduced by lack of information about the experiment and twelve could not be reproduced because the datasets were not published. They enumerate a list of desired properties in every publication concerning the detection of botnet, where we can highlight: reproducibility; experiment setup; metrics; and comparison of results. They also speak of the possibility of creating a hybrid detection method, and performing various algorithms simultaneously using a big data solution.

In works such as [Singh et al. 2014, Saad et al. 2011, Biglar Beigi et al. 2014, Zhao et al. 2012, Zhao et al. 2013], the authors implement botnet detection methods using various machine learning techniques. They talk about the importance of continuous training and parameters refinement of the learning algorithms to deal with the botnet mutable behavior in the network traffic. Something that requires a well-documented history of the experiments. [Singh et al. 2014] also raises the need of a big data solution to handle all the network traffic volume.

All the works presented exposed, in some way, the need for a methodology or tool to perform botnet detection experiments so they can be reproduced and compared as well as allow them to evolve their methods in a systematic manner, either by reconfiguration the detection algorithms, either by the selection of new network flows features. Moreover, some authors also mentioned that network flows are difficult to process given its real time and big volume characteristics.

In summary, for botnet detection experiments, it is necessary to address the following requirements: support for creation, execution and reproduction of the experiments, comparability and analysis of their results, and ability to deal with big volume data. However, none of those works propose a tool or approach in this direction. On the other hand, there are generic approaches, already proposed, which can be useful, if combined with each other. These initiatives are discussed in the next section.

3. Background

Typically, a large scale scientific experiment consists of three basic activities : *capture*, *curation* and *analysis*. The *capture* is the acquisition of data, which appear in various shapes and scales, from sensors, simulations, etc. The *curation* may involve activities such as filtering the data and finding the correct data structure to store them. The data *analysis* covers several tasks during the activities flow, such as the use of databases, the application of data mining algorithms, modeling and visualization [Hey et al. 2009].

These kinds of experiments are usually composed by chaining a set of tasks that handle a huge amount of data. Typically, these experiments are created manually by connecting programs inputs and outputs, producing an execution flow plan [Davidson and Freire 2008]. This plan is also known as a workflow. In addition, each plan is configured through specific parameters and input settings. Often, after its execution, the outputs are analyzed and, depending on its performance, this may incur on changes on the parameter settings, aiming at better results. According to [Hey et al. 2009], the workflow is becoming a paradigm to large scale science, through the management of data preparation (capture and curation) and their analysis.

Workflow Management Systems (WfMS), which are automated coordination engines, are used to specify, instantiate, execute, audit and develop a workflow [1]. The WfMS often provides [Taylor et al. 2007]: a visual scripting interface so that scientists can design these workflows (plans) without programming; a way to integrate and access computing platforms transparently; means to conduct analyzes (workflows executions) over various sets of data in a systematic and automatic manner; a way to capture the configuration of the programs involved in these executions (workflow configuration) so that the results can be reproduced and the methods can be reviewed, validated, repeated and adapted. Moreover, WfMS optimize the use of computing resources, allowing partial execution of workflows, as well as rerunning tasks that have failed without the need to rerun the entire workflow [Gil et al. 2007].

Several authors say that the artifacts (data and programs) involved in scientific experiments should contain information about their origins. This information should include items such as who and what process produced the artifact, which sources were used, which parameters and settings were used during the experiment, when the experiment or part of it was performed, among others. All this information related to the artifact origins is also known as *provenance*. Provenance is also important to workflows and scientific experiments [Freire et al. 2008, Moreau et al. 2011]. It can be used as an aid to evaluate the quality and reliability of experiments, since it allows to interpret and understand their results, to check the sequence of steps that led to these results, to identify the entries of these experiments and, if possible, to reproduce their results [Freire et al. 2008].

Some authors proposed approaches to handle provenance and reproducibility in a big data environment: In [Korolev and Joshi 2014], the authors propose a tool to track provenance and allow the reproduction of experiments involving Big Data workflows using an instrumented version of the PIG language. The proposed tool is based on Git, Git-Annex and Git2Prov. In order to maintain the provenance of the data, the system needs to connect to the repository and register the source through Git2Prov. Although the system provides a high degree of reproducibility of the experiment, the author does not mention the overhead introduced by successive synchronizations with Git. The need for synchronization before and after each workflow step will probably make it impossible to use this approach in systems requiring real-time execution, as in the case of botnet detection; In [Akoush et al. 2013], the authors propose a tool called HadoopProv, which was developed to capture the provenance of record-level data. It captures provenance from the intermediate keys and values created in the MapReduce process. The overhead added vary from 7% to 30%, on the map and reduce respectively. For the development of the tool it was necessary to instrument the Hadoop source code. **In summary**, these last two works use an instrumentation approach, and do not count on the functionality of WfMS for the creation and execution of workflows.

With respect to botnet detection experiments, the combination of a WfMS, provenance capture and a high performance platform would be a great contribution. Therefore, this work aims to combine the mentioned generic approaches in an innovative way, with special focus on addressing botnet experiments requirements. This proposal can address other domains, but, to the best of our knowledge, there are no initiatives reported in the botnet area. Next section describes the proposal in details.

[1]http://www.wfmc.org/

4. Architecture Overview

As in large scale scientific experiments, the botnet detection methods are characterized by being frequently executed and are constantly evolving, analyzing a large volume of data that is renewed constantly. The executions of these methods need to be carried out in a systematic way, so that it is possible to handle the highest amount of network flows as possible, minimizing the impacts of cyber-attacks. To achieve this goal, an infrastructure is required, of both hardware and software (*High Performance Infrastructure*), with high processing power, compatible with the volume of data processed, the volume of provenance data and the complexity of the activities involved in the methods.

It is necessary, as it is in large-scale scientific experiments, the development, adaptation and combination of models and tools to give support to all steps of the experiments involving botnet detection methods, i.e. creation, execution and analysis, allowing for greater collaboration between researchers, through reproduction and comparison of the experiments, and consequently increase gains in this area.

With this in mind, we created and implemented an architecture that combines and adapts scientific workflow management solutions, provenance capture and high performance data processing solutions to support such experiments. In this section, we present the architecture overview.

For better understanding of our architecture, we are considering a generic machine learning workflow to detect botnets which consists of three main steps which may involve many tasks. *Capture* is the first step, where a set of tasks may be accomplished to capture the network flow. *Curation* is the second, where the features are extracted from the flows, such as total packages, total bytes exchanged and flow duration. This step can also include tasks to clean the network flows, remove white listed domains, create indexes and store the flows in a data structure for further analysis. The last one is the *Analysis*, where one or more machine learning algorithm will be executed to identify, based on the selected flows features, suspicious machines that may belong to a botnet.

The use of a WfMS with support to provenance capture and to create and execute workflows similar to the workflow described previously would solve some of the problems presented in the previous section, such as: the experiment setup, through the workflow creation and its configuration (registry of the algorithms parameter values and input settings), the reproduction of the experiment and automation of the execution process of the method.

Moreover, another important characteristic of WfMS is that it is possible to create and maintain pre-built tasks for reuse or sharing, i.e., tasks that are ready to compose users' workflows. Therefore, in our architecture, we provide pre-built tasks that implement the most common machine leaning algorithms which are one of the most important tasks during the botnet detection process because they are directly associated to the performance in terms of time and precision of the flow classification. As we are dealing with a huge amount of data, another feature of paramount importance of WfMS that meets our needs is the possibility of the partial execution of a workflow. This will save time and resources and will be highlighted again in the following section.

But just the use of WfMS is not enough. There are issues that remain open, that WfMS alone are not able to address, such as the results of each tested workflow configuration, cross-experiment analysis, methods comparison and workflow efficiency evolution. Problems that we address with others components of our architecture.

To solve the remaining problems, we designed the architecture shown in Figure 1. This architecture is divided in two stages. In the first stage, *Traffic Curation and Analysis*, the main actors are the workflows to detect botnets (WF_1, WF_2, ..., WF_N). They are responsible for tasks like select and read a source of network flows, choose the features to be evaluated by the machine learning algorithm and execute the training of the selected algorithm with their parameters. Both the values of the parameters of each task and the results found on each execution are captured and stored in the provenance database. As the training of those algorithms is a time-consuming task, we aim to store the trained algorithm as a provenance artifact for future reuse. Storing provenance of the workflows executions over time will allow the researchers to monitor their experiment, and evaluate the quality of the results of the corresponding workflow over time and to continuous refine its methods (also known as workflow evolution), either by selecting new flows feature or by changing the machine learning algorithm.

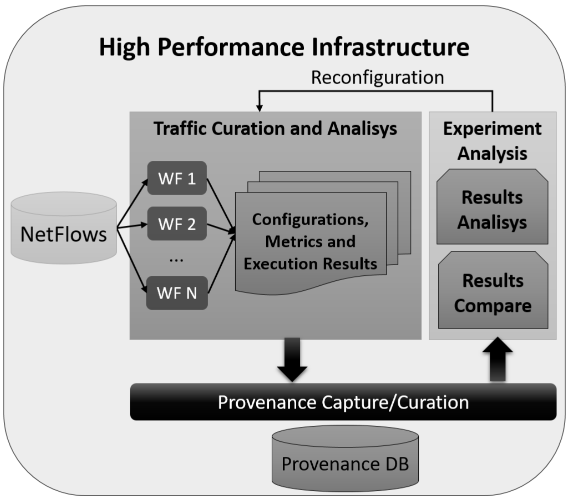

High Performance Infrastructure
Reconfiguration
Traffic Curation and Analisys
Experiment Analysis
NetFlows
WF 1
WF 2
...
WF N
Configurations, Metrics and Execution Results
Results Analisys
Results Compare
Provenance Capture/Curation
Provenance DB

Figure 1. Architecture Overview

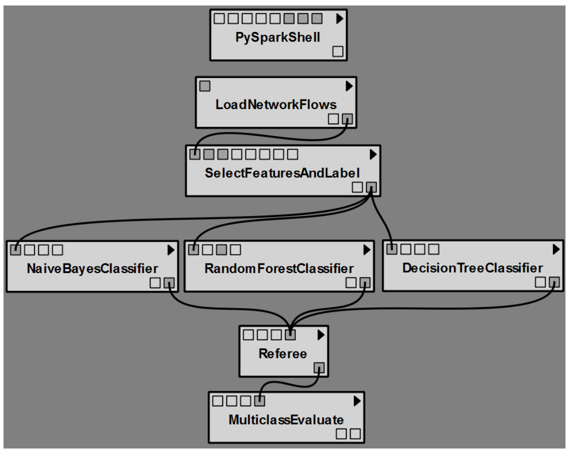


Figure 2. WfMS - VisTrails

At the second stage, *Experiment Analysis*, the researcher is able to analyze, compare and evaluate the results of different experiments. Each experiment is based on specific workflow and its set of configurations and corresponding executions. The main idea is to select the best learning algorithm to be used in a real botnet detection environment. Since all important information of the experiments are stored in the provenance database, as we will show in the next sections, we will be able to answer questions like: What is the fastest configuration of the Workflow $1(WF_1)$ for a given dataset? Which configuration is the most accurate? Which algorithm was used by WF_2 and with what parameter values? When the experiment was carried out and on which network flow database? A user interface is provided to facilitate the comparison of workflow executions and detection methods.

All the provenance captured, such as the workflows and all their configurations, parameter values and results, can be shared by researchers and, therefore, it is possible to repeat and compare the different methods. If sharing the dataset is not an option, it is not possible to reproduce the exact experiment, but with our architecture, the user is able to save and share the trained algorithm as a provenance asset, so that others can execute the experiment on their own datasets and still compare the results.

The implemented architecture, which is detailed in the next section, intends to solve most of the problems presented before and contribute significantly to the development of botnet detection methods and the experiments involving them.

5. Architecture Implementation

To implement the stage *Traffic Curation and Analysis*, we started by selecting the WfMS. It should support the workflow's creation, orchestrate the execution of the experiments and capture the provenance of the workflow's tasks. We decided to choose VisTrails [Bavoil et al. 2005] because it could provide most of the functionalities of our architecture. It is a WfMS with support to graphical workflow construction and provenance capture. In VisTrails, the tasks that compose a workflow are called *modules*, which have input ports (parameters or dataset inputs) and output ports (results). Figure 2 shows the VisTrails' canvas where the workflows are created.

In addition to designing workflows and storing the provenance of their executions, VisTrails allows the partial execution of these workflows, either by a failure in the execution of a module, or by a change in a module's attribute value. For each execution, it infers which modules were successfully executed and did not change any of their parameters and mark them as *cached*. Thus, they do not need to be executed again, optimizing the workflow execution. This functionality can significantly reduce the botnet experiment execution time when paired/coupled correctly with the execution environment, as we will show later on.

Moreover, VisTrails allows users to extend its functionality by creating new modules for reuse. These new modules can then be shared with other researchers, allowing the reassembly of their workflows, in order to perform new experiments.

VisTrails already offers a set of modules that implement machine learning algorithms using the Scikit-learn library [2], but these modules are not geared to handle large volumes of data. However, one of the main requirements of our architecture is to be able to run in a high-performance environment.

One may argue that there are solutions like Weka [Witten et al. 2011] that provides an environment for tracking experiments using and enchaining machine learning activities. But Weka's purpose is not workflow management, and thus it can not address important WfMS features, like provenance capture and workflow partial execution.

Therefore, we searched for a tool that, besides being able to be implemented as a VisTrails module, could provide support for running machine learning algorithms in a high performance environment. At the same time, it should be flexible enough to allow scientists to implement their various and complex detection algorithms, avoiding any kind of limitation when they are developing their methods. With that in mind, we chose Spark [Zaharia et al. 2010], a "fast and general engine for large-scale data processing". Spark, besides being a platform for running programs in clusters, provides a library for machine learning algorithms optimized for parallel execution (ML).

[2]http://scikit-learn.org/

Spark allows users to connect to its cluster through an *interactive shell*. This enables the process of knowledge discovery to be conducted in an interactive and extremely efficient way, because while the shell is operating, the data is kept in memory, optimizing the execution of multiple operations on the same dataset.

To integrate Spark functionality with VisTrails, the development of several modules was necessary. The mapping of its functionality into modules allowed the creation of workflows inside VisTrails and hence the orchestration of their execution by VisTrails using Spark resources. These modules represent activities often undertaken in botnet detection methods based on machine learning. For every Spark feature/function we wanted to use, we implemented it as a VisTrails module. Spark will also serve as the infrastructure to run the machine learning tasks.

The main module of our implementation is the *PySparkShell*. It allows us to take advantage of one of the main feature of both tools, VisTrails *partial execution* and Spark *interactive shell*, connecting them in an innovative way. It creates an SSH channel to the Spark cluster and instantiate a PySpark shell. This shell is the channel through which commands are submitted to the Spark Cluster. Each command is created and sent by the respective module using that channel. The return of the command is also captured by the module itself when pertinent, i.e., the *MulticlassEvaluate* module captures the metrics that will be displayed later in ProvTracker. This Spark shell is kept open even after the end of the workflow execution. Thus, Spark understands that the application continues to run and maintains all data and hardware resources available for the application, allowing VisTrails to reuse them. This enables and ensures proper partial execution of the workflow optimally, executing only the necessary modules, both in case of an error in any of the modules, as in case of a parameter change/tune. To disconnect from Spark and release the resources allocated, it is necessary to close VisTrail.

Some other useful and reusable modules were also implemented to date are: *LoadNetworkFlows*; *SelectFeaturesAndLabel*; *Decision Tree*; *Random Forest*; *Naive Bayes*; *Multilayer Perception*; *Logistic Regression*; and *MulticlassEvaluate*. These modules are detailed as follows.

The *LoadNetworkFlows* module is able to read the Network Flows formatted in a CSV style. Module *SelectFeaturesAndLabel* allows the user to select each flow features he wants to be analyzed. Moreover, modules that implement wrappers to Spark provided machine learning facilities were made available, e.g., *Decision Tree*, *Random Forest*, *Naive Bayes*, *Multilayer Perception*, and *Logistic Regression*. All the Spark functions parameters are available as input ports on the corresponding modules.

Another important and required module is the *MulticlassEvaluate* module. It evaluates the results using the metrics precision, recall, f-measure, false positive rate and true positive rate for each label and the whole experiment (weighted metrics). This module also captures the results so it can be shared and analyzed.

Figure 2 shows a complete botnet detection workflow similar to the methods described at [Singh et al. 2014, Saad et al. 2011, Biglar Beigi et al. 2014, Zhao et al. 2012, Zhao et al. 2013], but using three machine learning algorithms simultaneously, *Random Forest*, *Decision Tree* and *Naive Bayes*. Note that it initiates with the *PySparkShell* module and ends with the *MultiClassEvaluate* module. In order to benefit from the Spark facilities and to capture execution provenance, it is necessary to use these two modules. However, the user is free to compose his own workflow to address his specific needs, varying the algorithms, network load facilities, feature selection, etc.

It is worth to mention that using the Spark *interactive shell* through VisTrails (PySparkShell) it is possible to run more than one algorithm on the same dataset without the need to reload it and select its features again. As VisTrails only runs modules once per execution, and we do not close the connection to Spark, we ensure that all algorithms can access the dataset loaded in the beginning of the workflow.

All the trained models from the classifiers and *SelectFeaturesAndLabel* can also be captured and saved for later reuse or sharing, e.g. the trained algorithm can be used to classify new datasets without the need to train the algorithm every time a new dataset is available, or, as we mentioned in previous section, the trained algorithm can be shared between researchers so they can compare their results without the need to share their datasets.

At the next stage of the architecture shown in Figure 1, stage *Experiment Analysis*, subsidies are given to the scientists to analyze the experiment and results found by their methods. VisTrails allows the workflow and their provenance to be saved in a file, which can be shared with other scientists. It tracks each workflow modification, such as parametrization variations and allow the user to save it as a version into the file. For each workflow version (configuration) execution, VisTrails appends a set of provenance data to that file, such as: execution time of each module; if a module was cached; if the execution was successful; etc. To better visualize this data, we developed ProvTracker[3], a web tool that reads those files and show them in a friendly manner to the user.

Figure 3(a) shows the main page of ProvTracker. This page has a table with information about each experiment for which a *Workflow* was designed, its *Configurations*, one for each different parametrization used for its execution, and some metrics of the corresponding configuration executions: the average accuracy of the executions results; the average time spent to execute the complete workflow; the total number of executions; etc. In order to view more details about each workflow configuration, such as the parametrization values, the user has to click on the corresponding line. In the example of Figure 3(a), we can see the details of the workflow configuration *DefaultValues* from the Workflow *RandomForest*. Here we can see, for example, that the algorithm has been trained with *maxDepth* property (maximum depth of the tree) equals to 5.

ProvTracker also allows the scientist to view information about each workflow configuration execution. In the example of Figure 3 (b), ProvTracker displays information about the execution of the workflow configuration *ReducedFeatures* from the Workflow *RandomForest*. It informs: the *user* who performed the execution, the *VisTrails version* used, the *start* and *end* times of the workflow execution, if the execution was *complete*, the *total execution time* and the *accuracy metrics* of the corresponding execution results, e.g. true and false positive, precision, recall, f1, etc. It is also possible to see more details of each executed workflow module, like *execution time* and if the module was *cached* during the execution of the workflow.

It is worth to highlight that with ProvTracker, it is possible to analyze the workflow

[3]Available at http://ftoliveira.github.io/provtracker/

performance across its different configurations, and to identify the best ones. In addition, once it is possible to share the VisTrails file with the workflow configurations, other scientists can also use the tool to load all the experiment data, choose a workflow configuration and run a similar experiment on his own data. This allows scientists to compare results of different but similar experiments. Moreover, if the original input datasets are also made available, it is possible to other users to reproduce the whole experiment. Finally, the monitoring of the executions of a particular workflow over time also allows the scientist to check the effectiveness of the methodology. If its accuracy decays, he/she may review the workflow design and evolve it, in order to gain efficiency and detect botnets. It is also worth to mention that this implementation can be reproduced, since it uses open, free and mature software.

6. Results

We developed four test scenarios to explore the architecture's features and show its usability. We used our developed modules along with the CTU-13 dataset [Garcia et al. 2014a], a botnet traffic dataset captured in 2011 at the CTU University in the Czech Republic. It contains data of real botnets traffic mixed with normal traffic (reliable machines) and background traffic (untrusted machines). The CTU-13 dataset consists of thirteen captures of seven different botnet samples. Each capture was downloaded as a NetFlow file (generated with Argus) including the labels, in a CSV like style.

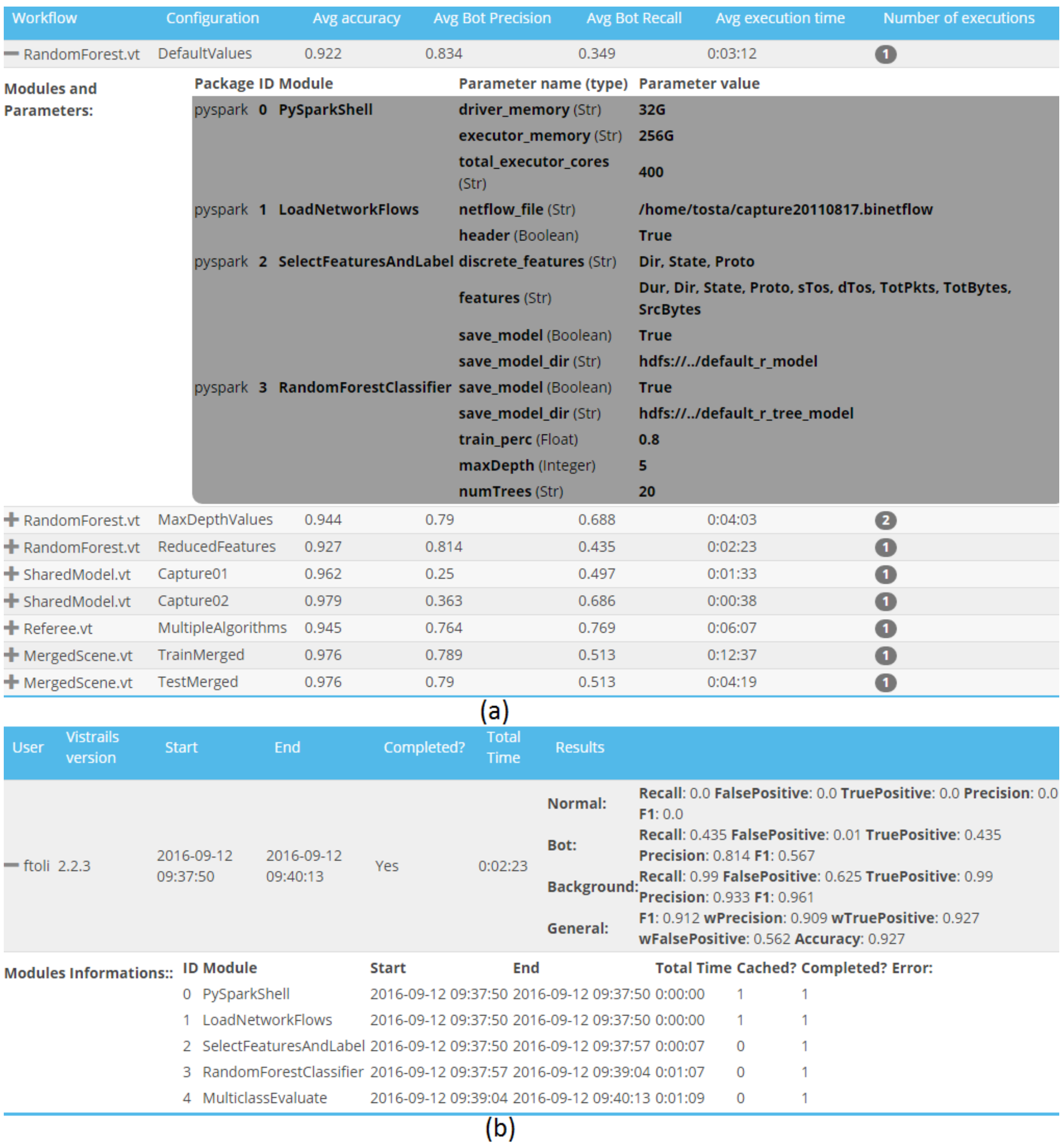
Among the information contained in each capture file, we highlight nine network flow features that we used to test our tool: Dir, State, Proto, Dur, sTos, dTos, TotPkts, TotBytes and SrcBytes. The first three of them are categorical (discrete) and the others contain continuous values. There is also the information about the category (label) of each flow, *botnet*, *normal* and *background*.

The experiment environment is composed by five machines, each one has 80 cores and 512GB of RAM. The amount of resources requested to the Spark Cluster is passed through the parameters of the *PySparkShell* module (Fig. 3(a)).

For the first scenario, we created a workflow in VisTrails called *RandomForest*, similar to the workflow shown in the Figure 2, but only with the Random Forest Classifier. The CTU-13's Capture 09 dataset (*Neris* bot) was used as input, since it has the highest number of botnet flows of all captures. We split this capture in 80% for training and 20% for testing. Internally, the module developed gets the correct % of each label.

Figure 3(a) shows that the *RandomForest* was executed with 3 different configurations. The first is called *DefaultValues*, and uses the Spark default values for the Random Forest Classifier (e.g. $numTrees = 20$, $maxDepth = 5$, etc.). For the *MaxDepthValues* and *ReducedFeatures* configurations, the maximum tree depth parameter value was changed to 30 and 5, respectively. The first two configurations used all the nine features available, while the third one used only Dur, State, Proto, TotPkts, TotBytes and SrcBytes.

Analyzing the results of the executions for the different configurations (Fig. 3(a)), the *ReducedFeatures* configuration was the fastest, while the *MaxDepthValues* was the slowest, but with the highest accuracy. Figure 3(b) shows information about the execution of the configuration *ReducedFeatures* where the modules *PySparkShell* and *LoadNetworkFlows* were cached. It happened because this configuration was executed right



Workflow	Configuration	Avg accuracy	Avg Bot Precision	Avg Bot Recall	Avg execution time	Number of executions
— RandomForest.vt	DefaultValues	0.922	0.834	0.349	0:03:12	1

Modules and Parameters:

Package	ID	Module	Parameter name (type)	Parameter value
pyspark	0	PySparkShell	driver_memory (Str)	32G
			executor_memory (Str)	256G
			total_executor_cores (Str)	400
pyspark	1	LoadNetworkFlows	netflow_file (Str)	/home/tosta/capture20110817.binetflow
			header (Boolean)	True
pyspark	2	SelectFeaturesAndLabel	discrete_features (Str)	Dir, State, Proto
			features (Str)	Dur, Dir, State, Proto, sTos, dTos, TotPkts, TotBytes, SrcBytes
			save_model (Boolean)	True
			save_model_dir (Str)	hdfs://../default_r_model
pyspark	3	RandomForestClassifier	save_model (Boolean)	True
			save_model_dir (Str)	hdfs://../default_r_tree_model
			train_perc (Float)	0.8
			maxDepth (Integer)	5
			numTrees (Str)	20

Workflow	Configuration	Avg accuracy	Avg Bot Precision	Avg Bot Recall	Avg execution time	Number of executions
+ RandomForest.vt	MaxDepthValues	0.944	0.79	0.688	0:04:03	2
+ RandomForest.vt	ReducedFeatures	0.927	0.814	0.435	0:02:23	1
+ SharedModel.vt	Capture01	0.962	0.25	0.497	0:01:33	1
+ SharedModel.vt	Capture02	0.979	0.363	0.686	0:00:38	1
+ Referee.vt	MultipleAlgorithms	0.945	0.764	0.769	0:06:07	1
+ MergedScene.vt	TrainMerged	0.976	0.789	0.513	0:12:37	1
+ MergedScene.vt	TestMerged	0.976	0.79	0.513	0:04:19	1

(a)

User	Vistrails version	Start	End	Completed?	Total Time	Results	
— ftoli	2.2.3	2016-09-12 09:37:50	2016-09-12 09:40:13	Yes	0:02:23	Normal:	**Recall**: 0.0 **FalsePositive**: 0.0 **TruePositive**: 0.0 **Precision**: 0.0 **F1**: 0.0
						Bot:	**Recall**: 0.435 **FalsePositive**: 0.01 **TruePositive**: 0.435 **Precision**: 0.814 **F1**: 0.567
						Background:	**Recall**: 0.99 **FalsePositive**: 0.625 **TruePositive**: 0.99 **Precision**: 0.933 **F1**: 0.961
						General:	**F1**: 0.912 **wPrecision**: 0.909 **wTruePositive**: 0.927 **wFalsePositive**: 0.562 **Accuracy**: 0.927

Modules Informations::

ID	Module	Start	End	Total Time	Cached?	Completed?	Error:
0	PySparkShell	2016-09-12 09:37:50	2016-09-12 09:37:50	0:00:00	1	1	
1	LoadNetworkFlows	2016-09-12 09:37:50	2016-09-12 09:37:50	0:00:00	1	1	
2	SelectFeaturesAndLabel	2016-09-12 09:37:50	2016-09-12 09:37:57	0:00:07	0	1	
3	RandomForestClassifier	2016-09-12 09:37:57	2016-09-12 09:39:04	0:01:07	0	1	
4	MulticlassEvaluate	2016-09-12 09:39:04	2016-09-12 09:40:13	0:01:09	0	1	

(b)

Figure 3. ProvTracker interface. (a) Main page (b) Executions

after the configuration *MaxDepthValues*, and the parameters of those two modules did not change. Therefore, they were not executed again, saving processing time. We can also see more detailed metrics information, e.g. for the Bot label we have a *Precision* of 0.814.

The second scenario illustrates the sharing facility of the implemented architecture. Assuming that the input dataset could not be shared, the idea was to share the generated models. Note that the models generated by the *RandomForestClassifier* and *SelectFeaturesAndLabel* modules were saved during the execution of the *DefaultValues* configuration (save-model=true). Then, the Random Forest model was used as a classifier in the execution of the *SharedModel*, which is very similar to the *RandomForest*. This new workflow was configured to use two different CTU's captures datasets as inputs, *Capture01* and *Capture02*, respectively, and to load previously saved models.

Analyzing the execution results of the *SharedModel* (Fig. 3(a)), we can see that the average accuracy was still high and the time spent to execute it was 2-6 times faster than the original workflow (the one that generated the models). This is because the *SelectFeaturesAndLabel* and *RandomForestCassifier* modules used previously saved models, and then, since it was not necessary to create new models, they executed much faster.

For the third scenario, we tested the use of a more sophisticated workflow named *Referee*, just like the one in Figure 2, which embeds three different ML algorithms. For its configuration named *MultipleAlgorithms*, we used *maxDepth*=20 and *numTrees*= 10 for the *RandomForestClassifier* module, *maxDepth*=30 for the *DecisionTreeClassifier* module and the default values for the *NaiveBayersClassifier* module. To select the final label, we got the modal (most common) label from the three classifiers. Despite the long training time, it showed better results with respect to the precision and recall of bots.

For the fourth scenario, we wanted to test the architecture with a bigger dataset. To accomplish that, we merged all thirteen datasets into a bigger one, which has approximately 20 million flows on a 3GB file. We trained a Decision Tree Algorithm with maximum depth of 20 and named the workflow as *MergedScene*. The first configuration, named *TrainMerged*, was used to train and save the classifier model while the second one, *TestMerged*, was used to classify yet another dataset, created by concatenating this bigger dataset four times, reaching 80 millions of rows, using the saved model from the previous configuration and no cached module.

The idea was to evaluate the application of a saved model in a bigger dataset, to see if the architecture could scale and handle larger volumes of data, as in real environments, where we first train the algorithm and them use the corresponding model to classify new flows. The training time, for the *TrainMerged* configuration, as we expected, took a bit longer than the others, 12:37 minutes, 10:07 to finish the module *DecisionTreeClassifier* and 1:37 to finish the module *MulticlassEvaluate*. However, the *TestMerged* configuration execution took 4:19 minutes where 1:17 was used to evaluate the classifier and only 2:18 seconds to classify the whole dataset, almost 80 million flows!

These four scenarios showed that our architecture could address all the botnet experiments requirements: creation, execution, results analysis, experiments reproducibility and cross experiments' results comparability. In addition, it is worth to note that it was capable to scale up and handle a large amount of data.

7. Conclusion

Although there is a large number of techniques for botnet detection, they remain a threat to users and institutions, requiring the development of new detection techniques. The studies analyzed showed that the experiments performed to evaluate those techniques are still precarious, not allowing their reproduction and comparison, limiting the studies in this area. In this work, we raised the importance of a methodology to drive the execution of botnet detection experiments and the great need for frameworks and tools to support these experiments.

We specified and implemented an architecture that addresses most of the issues raised in the literature. The tests showed that it is feasible to implement and that it will benefit the experiments in this area, allowing scientists to create, execute, analyze, reproduce and compare their experiments, even if the input dataset is not shared. The implementation also addresses the big volume of data involved in the experiments.Moreover, the implementation contributes with a new solution for integration of VisTrails WfMS and Spark, so we can capture the provenance of the tasks involved in the experiments as well as the results achieved.

Future works include to improve the *Results Compare* module so we can facilitate the analysis of the experiment and answer questions such as: show me all workflows that used a particular input dataset. We also aim to use the architecture to capture and classify the netflows in real-time, similar to the one proposed by [Singh et al. 2014], but with provenance capture and all other benefits of our architecture. Because of the dynamism of botnets, the learning algorithms need to be trained constantly so they can evolve systemically. Upon detection of a new malicious flow, it needs to be validated by a specialist and reinserted into the dataset for retraining the algorithm. This task needs to be done systematically. As botnets flows are rare classes when compared to legitimate flows, this is feasible to be done.

References

Akoush, S., Sohan, R., and Hopper, A. (2013). HadoopProv: Towards Provenance as a First Class Citizen in MapReduce. In *Presented as part of the 5th USENIX Workshop on the Theory and Practice of Provenance*, Berkeley, CA. USENIX.

Aviv, A. J. and Haeberlen, A. (2011). Challenges in Experimenting with Botnet Detection Systems. In *Proceedings of the 4th Conference on Cyber Security Experimentation and Test*, pages 6–6, Berkeley, CA, USA. USENIX Association.

Bavoil, L., Callahan, S. P., Crossno, P. J., Freire, J., Scheidegger, C. E., Silva, C. T., and Vo, H. T. (2005). Vistrails: Enabling interactive multiple-view visualizations. In *Visualization, 2005. VIS 05. IEEE*, pages 135–142. IEEE.

Biglar Beigi, E., Hadian Jazi, H., Stakhanova, N., and Ghorbani, A. A. (2014). Towards effective feature selection in machine learning-based botnet detection approaches. pages 247–255. IEEE.

Davidson, S. B. and Freire, J. (2008). Provenance and Scientific Workflows: Challenges and Opportunities. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1345–1350, New York, NY, USA. ACM.

Freire, J., Koop, D., Santos, E., and Silva, C. (2008). Provenance for Computational Tasks: A Survey. *Computing in Science & Engineering*, 10(3):11–21.

Garcia, S., Grill, M., Stiborek, J., and Zunino, A. (2014a). An empirical comparison of botnet detection methods. *Computers & Security*, 45:100–123.

Garcia, S., Zunino, A., and Campo, M. (2014b). Survey on network-based botnet detection methods: Survey botnet detection methods. *Security and Communication Networks*, 7(5):878–903.

Gil, Y., Deelman, E., Ellisman, M., Fahringer, T., Fox, G., Gannon, D., Goble, C., Livny, M., Moreau, L., and Myers, J. (2007). Examining the Challenges of Scientific Workflows. *IEEE Computer*, 40(12):26–34.

Hey, T., Tansley, S., and Tolle, K., editors (2009). *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, Redmond, Washington, 1 edition edition.

Hofstede, R., Celeda, P., Trammell, B., Drago, I., Sadre, R., Sperotto, A., and Pras, A. (2014). Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX. *IEEE Communications Surveys & Tutorials*, 16(4):2037–2064.

Hogben, G., Plohmann, D., Gerhards-Padilla, E., and Leder, F. (2015). Botnets: Measurement, Detection, Disinfection and Defence - ENISA.

Korolev, V. and Joshi, A. (2014). PROB: A tool for Tracking Provenance and Reproducibility of Big Data Experiments. In *Reproduce '14. HPCA 2014*.

Moreau, L., Clifford, B., Freire, J., Futrelle, J., Gil, Y., Groth, P., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., Plale, B., Simmhan, Y., Stephan, E., and den Bussche, J. V. (2011). The Open Provenance Model core specification (v1.1). *Future Generation Computer Systems*, 27(6):743–756.

Saad, S., Traore, I., Ghorbani, A., Sayed, B., Zhao, D., Wei Lu, Felix, J., and Hakimian, P. (2011). Detecting P2p botnets through network behavior analysis and machine learning. pages 174–180. IEEE.

Shi Meilin, Yang Guangxin, Xiang Yong, and Wu Shangguang (1998). Workflow management systems: a survey. volume vol.2, page 6. Publising House of Constr. Mater.

Silva, S. S., Silva, R. M., Pinto, R. C., and Salles, R. M. (2013). Botnets: A survey. *Computer Networks*, 57(2):378–403.

Singh, K., Guntuku, S. C., Thakur, A., and Hota, C. (2014). Big Data Analytics framework for Peer-to-Peer Botnet detection using Random Forests. *Information Sciences*, 278:488–497.

Taylor, I. J., Deelman, E., Gannon, D. B., and Shields, M., editors (2007). *Workflows for e-Science*. Springer London, London.

Witten, I. H., Frank, E., and Hall, M. A. (2011). *Data mining: practical machine learning tools and techniques*. Morgan Kaufmann series in data management systems. Morgan Kaufmann, Burlington, MA, 3rd ed edition. OCLC: ocn262433473.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10–10, Berkeley, CA, USA. USENIX Association.

Zhao, D., Traore, I., Ghorbani, A., Sayed, B., Saad, S., and Lu, W. (2012). Peer to Peer Botnet Detection Based on Flow Intervals. In Gritzalis, D., Furnell, S., and Theoharidou, M., editors, *Information Security and Privacy Research*, volume 376, pages 87–102. Springer Berlin Heidelberg, Berlin, Heidelberg.

Zhao, D., Traore, I., Sayed, B., Lu, W., Saad, S., Ghorbani, A., and Garant, D. (2013). Botnet detection based on traffic behavior analysis and flow intervals. *Computers & Security*, 39, Part A(0):2 – 16.