Using machine learning techniques for traffic classification and preliminary surveying of an attacker's profile

P. Frühwirt[1], S. Schrittwieser[2], E. R. Weippl[1]
[1] SBA Research, Vienna
[2] St. Pölten University of Applied Sciences
pfruehwirt@sba-research, sebastian.schrittwieser@fhstp.ac.at, eweippl@sba-research.org

Abstract

The increasing complexity of systems brings up new attack vectors and it is therefore easier to compromise systems. A defender of a system is often forced to quickly assess the situation and develop an appropriate defense strategy. The most common way to protect their own networks are Intrusion Detection Systems (IDS). IDS detect attacks either using predefined, static signatures, or based on the behavior of users. Existing systems are inflexible due to static and often stale signatures and thus can be easily bypassed by attackers.

This paper on the one hand presents the theoretical concepts of detection of attackers and extends existing attack mitigation approaches by machine learning mechanisms, which can be used during security exercise/challenges like the UCSB International Capture The Flag (iCTF). By improving and combining static signatures with machine learning approaches, a new technique of attack detection called "Classification Voting" was developed, which reduces the number of false positive alerts in an production environment. Our approach allows the generation of signatures without dedicated domain knowledge by guided manual classification of detected network packets. Based on machine learning, new packets can be further classified using the generated model. At the same time the existing models of other signatures can be improved by adding new packets.

1 Introduction

Due to the growing complexity of systems in use, the number of attack vectors and backdoors raises. As a defender it is often necessary to assess the state of the system as fast as possible to prevent further damage [15]. An attacker on the other side has much more time for preparation. Hence the defender is in the weaker position concerning time. Due to the growing amount of traffic and data, good automated or at least semi-automated tool support is essential.

Existing solutions like intrusion detection systems only gather information about the defender and the attacked system itself. However, it gets more and more important to gather knowledge about the attacker: Hence, the reaction towards a more professional hacker differs from a script kiddie, uses ready-made exploits from the internet.

Intrusion detection systems follow in most cases two different approaches how to detect an intruder: *signature-based* or *behaviour-based* detection. On the one hand behaviour-driven detection uses a profile of unsuspicious activity and compares new traffic to this profile and has therefore a good detection rate for external attacks, but it is really hard to define a regular network activity, because systems are getting more and more complex and flexible. On the other hand there are signature-based detection systems that are easy to implement, however a set of signatures has to be created (or bought). If these signatures are under sampled an attacker can easily bypass the definition just by modification of the attack signature. On the other hand if the signature is over sampled you may block regular traffic. Nevertheless a security professional with lots of expertise has to define appropriate signatures.

Our approach uses a combination of these two techniques. It enables the user to create signatures just by selecting sample traffic and deciding whether a request is unobtrusive or suspicious. In the evaluation part is covered by a machine-learning algorithm to create a definition of the pattern. The challenge of this approach is to make the machine learning process as transparent as possible and in spite of obtaining satisfying results. To avoid aberration we introduce a new technique called *Classification voting* that is described in Section 4.6.

Based on state of the art of network-based defence mechanism (Section 2) and existing machine learning approaches we developed theoretical concepts to improve machine learning techniques used in intrusion detection systems. We have evaluated existing concepts and assessed the differences between these concepts differs from our approach.

In order to evaluate our theoretical concepts we developed a prototype. We used the official traffic dumps of the international Capture the Flag (iCTF) contest [24] of 2011 [25]. Our prototype is used to automatically process network packets in order to create machine-learning models to classify the traffic. We trained our prototype to detect penetration-testing tools, which have a relatively static signature in general. Furthermore we trained our system to detect the scorebot, which is designed to be obfuscated because teams should not detected it, in order to filter all non-scorebot traffic during contest e.g. attacks of other teams. After creating the models we measured the classification error rate by counting wrong classified traffic (false positives).

During development and evaluation of the prototype we

gained first results and improved the classification error rate by using state of the art machine learning methods and applying these concepts to our theoretical models. Further we tried to use our concept on other research fields in which we investigate more effort in future research.

The remainder of this paper is structured as follows: In Section 2 we take a look of state of the art network-based defence mechanism. Based on these concepts we introduce in Section 3 theoretical concepts for improving intrusion detection systems and offer system design suggestions. Further we developed a prototype to evaluate our theoretical concepts and present our evaluation results in Section 5. Finally, we outline our results and provide an outlook of our future research in Section 6.

2 Network-Based Defence Mechanisms

Most security experts would agree on the statement that it is impossible to build a completely secure system. Therefore it is important to have good defence mechanisms to stay alert for attacks. In this section we discuss state-of-the-art approaches and give an overview of possibilities to defend a network. Network-based defence mechanisms are an important countermeasure, because they add an additional layer of security to the network and therefore can minimize the risk of attacks.

Intrusion detection systems are considered as an appropriate solution for network protection. These systems are designed to actively detect and possibly prevent intrusions by monitoring different sources like especially the network traffic. There are different approaches how intrusions are detected. Some systems use more or less static signatures, others detect deviations from normal traffic (abnormality detection). Over the past years many new detection techniques have been developed. One of these new approaches is the use of machine learning algorithms and statistical modeling to detection intrusions.

2.1 Intrusion Detection Systems

Intrusion Detection Systems (IDS) are devices or software components which monitor computer systems or network traffic and look for malicious activities. In case of a policy violation they generate reports for the management. Intrusion detection systems are primarily monitoring systems. Therefore it is not necessarily guaranteed that these systems stop or prevent intrusions [11].

Intrusion detection is based on the assumption that an intruder behaves differently compared to an unsuspicious user in a way that can be qualified. One cannot assume these two behaviours do not overlap. The main challenge for an intrusion detection system is to differ between these two types of behaviour. If the system uses a loose interpretation (under sampling) of the intruder's behaviour it may increase in *false positives* and the intrusion isn't detected any more. If the system is too strict (over sampling) it may identifies normal user traffic as intrusion and produces *false negatives*.

2.2 Classification

Intrusion Detection Systems can be classified as *Host-Based*, *Network-Based* and *Hybrid Intrusion Detection System*. Host-Based Intrusion Detection Systems monitors the local activity of a computer system such as processes and log files and therefore have deeper information about the system itself. On the other hand the Network-Based Intrusion Detection Systems have only access to the network. Hybrid Intrusion Detection Systems are a combination of both approaches, which can achieve higher detection rates because of more and more detailed information about the network and the individual components.

2.3 Machine Learning approaches

Machine learning offers flexibility to react to new situations and retrain to battle new attacks or intrusions. In classic implementations of intrusion detection systems static rules and signatures are used [21, 22]. However this approach often fails and can easily be tricked. Therefore often machine learning approaches are used to gain better results and more flexibility [1, 13, 17, 18, 26, 28]. Machine learning has a high potential and in theory it could solve many problems of Internet Service Providers if they can classify traffic in real-time. Thuy T.T. Nguyen et al. made a survey [19] where they evaluated 18 significant works from 2004 until early 2007 which tries to uses machine learning techniques for real-time classification. It showed that classifiers like decision trees or NaiveBayes can achieve high accuracy for different types of network application traffic. The survey demonstrated that most approaches build their classification models on sample data during setup. However it is still an open question how choose an appropriate test set. They suggested a combination of classification models to avoid false positives. In our approach we are improving this suggestion by using a new technique called *classification voting* (Section 4.6).

Many concepts use different protocol independent features like packet length statistics, flow size and duration, inter-arrival time statistics [27, 6, 10]. These techniques have the advantage of being independent of the used application, therefore can react to unknown protocols and not for each application an individual model and feature set have to be developed. However it is not possible to train for application specific weaknesses and therefore a general feature set is maybe to fuzzy.

David Endler is using machine learning to process events of the basic security module auditing tool of Sun's Solaris operating system [7]. He showed that intrusion detection systems can be improved by using machine learning on the generated events of a IDS and therefore reduce false positives. This approach is limited to the functionality of the intrusion detection system.

Chris Sinclair et al. presented an approach of using machine learning in context of network intrusion detection [23]. His approach is using decision trees and genetic algorithms to generate rules for an intrusion detection system. This technique generates a model without verification against another data set, therefore it is hard evaluate the generated rules. Further, this approach is very susceptible to wrong

classified instances, because the decision is limited to a single classifier, as argued in the survey of Thuy T.T. Nguyen et al. before.

Casey Cipriano et al. use the developed prototype Nexat a different type of machine learning for intrusion detection [5]. They make use of the sequence of triggered alerts of existing IDSes for training. Nexat splits up the different alerts and into single events and connects the alerts into attack sessions. Basically they try to connect these alerts using the source and destination address and a time window to put together an attack session. After session collecting the system learns what attacks follow after a sequence of alerts. After training they use these models to predict the next actions of an attacker.

Marco Barreno et al. [2] developed a framework for evaluating the security of machine learning based intrusion detection systems and asked the question "Can Machine Learning be Secure?". They claimed that an attacker may misuse machine learning approaches to "mis-train" the system and in that way confuse the IDS.

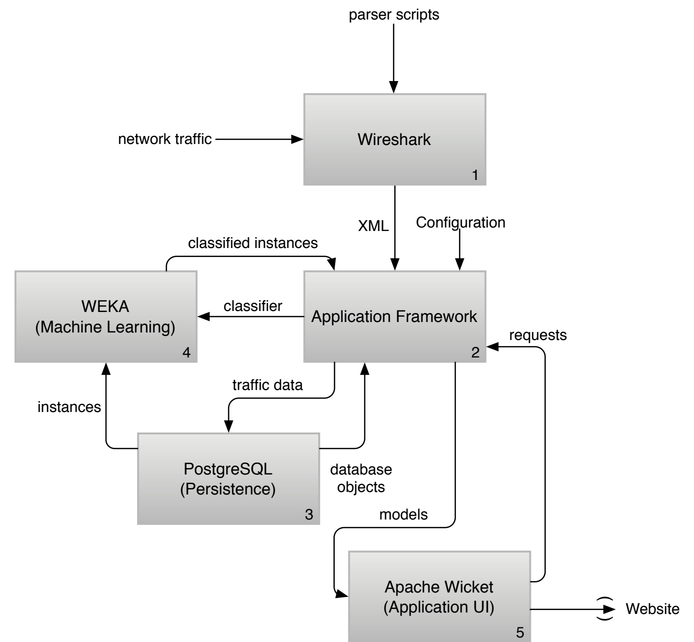

Figure 1: System overview

3 System Design and Implementation

As we have seen in Section 2, different techniques and solutions of network-based defence mechanisms exist. In many cases these systems are very complex and needs deeper knowledge and understanding, e.g. an administrator has to know how to create rules within an IDS. Therefore system administrators are required to understand how to create rules for detection of intrusions. However using signature-based detection techniques for the system is very static and may not be flexible enough to react to new intrusions.

Where there exists machine-learning processes, which try to simplify this static process by using anomaly-based detection. In our approach we try to create a new technique to generate some kind of dynamic rules by using machine learning. The user should not notice that he uses machine-learning, because it is quite a complicated process, and therefore we make the selection of the machine learning algorithm, data harvesting, learning and other important factors of the machine learning process to the user transparent. As mentioned in Section 2.3 there already exist some machine learning approaches for traffic classification. We go a step further and optimize existing algorithms with a new technique called classification voting (Section 4.6).

In order to evaluate our research (Section 5) we developed a prototype, which implements the new designed workflow. Figure 1 illustrate a schematic overview of our workflow in IDF0 notation. As input we use network traffic, which we want to classify. They input maybe a live traffic stream or a stored network dump.

To gain a defined context of the network data we parse the network stream with Wireshark in order to gather aggregated data stored in XML files. Our framework uses different parsing scripts for different kinds of network protocols and applications e.g. HTTP. Every network parsing script is implemented as Lua-script, which can add listener to `tshark`, a console base implementation of Wireshark. We can automate and manipulate Wireshark processes in order to generate this kind of XML output [12]. The application framework validates and parses the network traffic XML input and stores all network packages in the database. Each network traffic is connected to its input package therefore if something went wrong the framework restores the original state and marks the input file as damaged. This mechanism ensures integrity of the data. After successful procedure of new instances, the framework will start a data harvesting process. We will describe this process in detail in Section 4.

After these steps we have a set of instances, which will be processed by the machine learning unit. In our implementation we used WEKA [20]. WEKA loads all available instances with its own *JDBC*-Database connection and starts the classification procedure. We will describe this process in detail in Section 4.3.

The flow of the workflow is controlled by the framework and in especial by the user input over the Application User Interface. The WebUI triggers activities of the framework and request data from it. The framework will, after successful execution, send the data to the WebUI component.

4 Data harvesting

The data harvesting process is quite complex and involves many different components. Figure 2 shows a schematic overview of the harvesting process. Due to the fact that every network connection is different we have to transform into a common data model. Therefore we parse the network traffic with wireshark and some parsing scripts written in Lua. Wireshark offers a console-based version named `tshark` which offers the possibility to add own code to manipulate the processing of the traffic. We created a lua

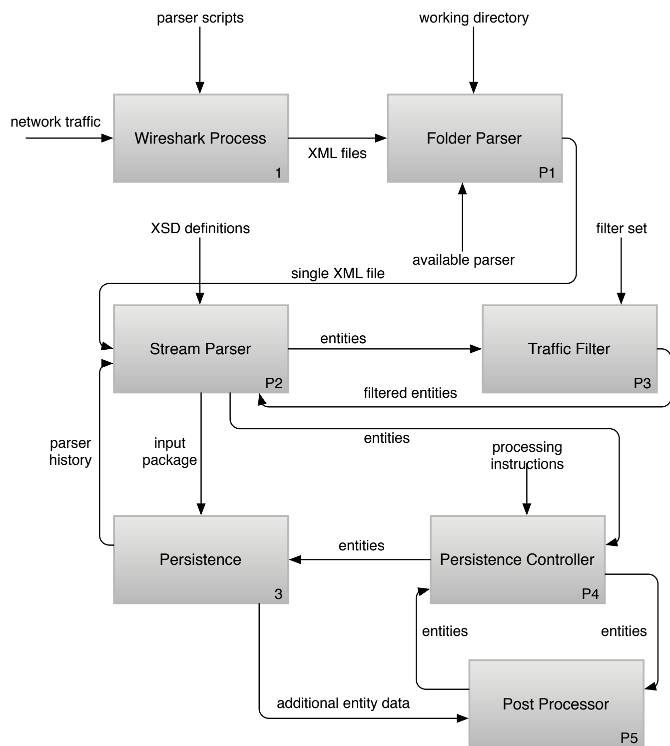

Figure 2: Data harvesting process

script, which listens to specific TCP packages, e.g. for HTTP traffic we used the pattern `tcp && http.request && !tcp.analysis.retransmission` for HTTP requests and `tcp && http.response && !tcp.analysis.retransmission` for HTTP responses. The lua scripts produces a XML file for each protocol and traffic package. It stores all generated files in a directory.

The folder parser scans the working directory for XML files of `tshark` and forwards all valid files to the concrete stream parser. Each of the following steps is encapsulated within a transaction. If there is an uncovered state, the framework will automatically restore every action of the package and will mark it as damaged. This action guaranties the integrity of the imported data. The stream parser stores all important information in the database and will use these data in further runs, e.g. if the script runs again it will skip already successfully imported packages.

In the first step the steam parser validates the input XML file with a XSD definition file to avoid errors caused by `tshark` and the Lua parsing script. The stream parse creates for each XML entity an entity object that is analysed by some traffic filters. Depending on the configuration the traffic filter will drop some packages like network noise. The persistence controller will further process the filtered entities.

The persistence controller is an abstraction of the underlying database management system. It stores the entities to the concrete SQL tables. Further it will do some post processing, for example each HTTP conversation contains one HTTP request and one HTTP response. Each part (request and response) is stored as a single XML entity due the fact that is not possible to create a container with both parts in lua. The post processor connects these two entities by finding the correct HTTP request for a certain HTTP response with the help of the database management system by using the TCP Session ID.

During the data-harvesting phase we try to collect as many attributes as possible by different techniques: *Collection*, *Abstraction* and *Extraction*.

Collection The collection method is one of the simplest methods to gain attributes: It simply gathers different properties of a connection and uses these properties as attribute. For example it is very easy to gain the source IP address of a connection and use this property as attribute for further operations.

Abstraction In many cases a simple collection of properties is to precise for machine learning (e.g. the concrete user agent string) therefore it is important to use an abstract model of the raw attribute data. In case of the user agent string it is e.g. in most cases it is sufficient to create an attribute "firefox" with the nominal values "yes" and "no" instead of the concrete user agent string with OS and browser version information.

Extraction In some cases you can gain further information out of the combination of attributes. The extraction technique uses some attributes as input and generates further properties. For example we use the URI, user agent string, connection protocol and destination port attributes to gain a new attribute "tool usage" by mapping these properties to a list of known penetration testing tools. Further we use this technique to gain the predecessor of an http request with the source IP, destination IP, destination port and connection protocol.

4.1 Attributes

General Connection attributes We used the following general attributes for every type of connection, regardless from the used protocol: Source IP, Source Port, Destination IP, Destination Port, Connection Type (TCP, UDP or ICMP), Connection Time, Connection close Time, Frame ID

HTTP specific In addition to the general connection attributes, it is possible to extract more attributes based on the http protocol. We only used attributes that improve the training result: URI and Full URI, HTTP Host, HTTP Method, HTTP Version, User Agent String, User Agent Abstraction (e.g. "uses firefox":yes/no), HTTP Cookie and HTTP Cookie abstraction ("uses cookie":yes/no), Is known Browser (Abstraction of user agent string; yes/no), Is known Tool (e.g. automatic security testing tool; yes/no), Uses path Traversal (yes/no), Uses special Characters (yes/no), Normalized URI (URI without parameter values, e.g.
news.php?id=20 → news.php?id=[?]), HTTP Predecessor, Levenshtein distance of URI and predecessor URI

We use the predecessor of each HTTP request as additional attributes. This attribute helps the machine learning process to generate knowledge about historical requests, e.g. a tool that iterates over a web site will have a similar predecessor request and a low levenshtein distance [14]. Based on these attributes it is possible to generate better models, i.e. request history based decisions.

4.2 Attribute Storage

Due to the fact that we don't know which protocol we are using and which attributes are needed, we implemented a dynamic database schema. Basically every type of connection creates an entry in the table *Connection*. However in most cases it is more efficient if we create a base table for each protocol that inherits each column of the table *Connection*. We implemented a table *HttpRequest* for the HTTP protocol with all basic attributes gained with the harvesting method *Collection* (e.g. HTTP method, host, HTTP version, etc.).

In order to add dynamically new attributes that we gain from *Abstraction* and *Extraction*, we created a helping table *ConnectionAttributeObject* that contains all available attributes and their properties.

We define a default value for each attribute, e.g. "no". If we collect a different attribute value then the default value we create a new entry in the *ConnectionAttribute* table. During the creation of the SQL view with all available attributes we use LEFT JOINs and replace all *null*-values with the default value. Therefore there is no need to have to save all attributes of an entity in the database.

4.3 WEKA Integration and Data Classification

After the data harvesting process, all raw data is processed and stored in a database management system. Each instance with its attributes is stored in the database, therefore they can be used for learning and classification. Figure 3 shows an overview of the machine-learning process.

Before we can classify new traffic, we have to use the existing data to learn the different attacks and signatures. During the learning phase the classification component loads all available instances. The user can create signatures by classifying existing instances with the UI of the prototype. These instances will be used for training and evaluation. Every classified instance will be used as a training set. However the user has to classify some instances to gain good results.

After manual classification by the user the framework uses this training set to build the models. The framework uses all available classifiers offered by the machine learning framework WEKA [20] to create the models. Afterwards each model is evaluated by 10-fold cross-validation. During the optimization phase these evaluation results and their performance are used to calculate a set of good choices for classifier. These sets of classifiers will be used for classification as models for the classification voting, which we will describe later in Section 4.6.

We gained different models during the training phase. These models are used to classify new traffic. In productive environments the framework will generate reports of the network activity. These reports contain all positive instances. For the further classification process the user can validate the correctness of these instances. Our framework can therefore adapt its models and create better results.

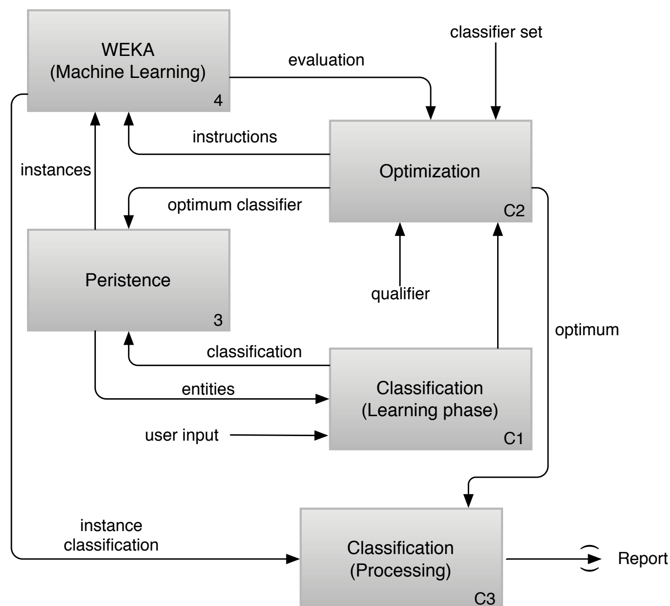

Figure 3: Data Classification process

4.4 Learning and adaption

Learning and adapting behaviour is a very important characteristic and ability of machine learning. Our framework supports learning of behaviour during its training phase. However these instances have to be classified by hand and therefore in most cases a small feature set is used as training set which triggers more classification errors and a worse data quality.

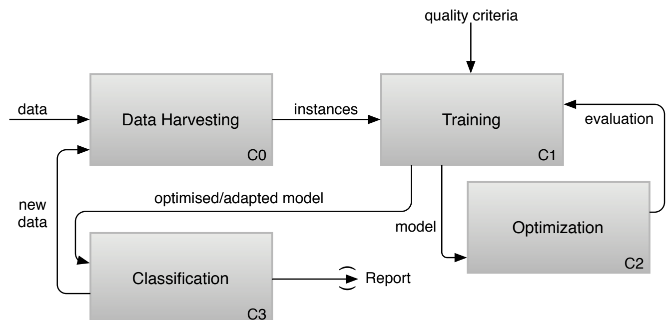

Figure 4: Learning and adaption process

To reduce false positives we used a new method called classification voting. To reduce even more classification errors we try to adapt the used training models by new traffic. Figure 4 shows the process of learning and adopting behaviour. After the data harvesting procedure the framework is trained based on some quality criteria like performance and efficiency. Each classifier offered by WEKA has a cer-

tain time frame to compute a result. If the classifier can't finish within this time frame, the framework will drop this classifier. In our implementation we used the time frame of five seconds.

4.5 Optimization

Each classifier will validate its model with 10-fold cross-validation and measure its performance p and classification error rate e. These attributes are used for the optimization phase to find a set of classifier to gain good results. The framework determines the best n classifiers by calculating an internal ranking value $r(c)$ of each classifier c by the following method:

$$r(c) = \frac{ln(p * 100)}{(1-e)^{10}}$$

This rating function will penalize algorithms with bad performance or a high error rate. Therefore it is possible that algorithms with a bad result and good performance will be chosen instead of a slow, good classifier. However due to the high exponent, classifier with higher error rate will be dropped even if they are very fast.

We use a new method called *classification voting* to avoid false positives therefore it is better if we have more classifier with a good performance instead of one optimal model with a bad performance.

4.6 Classification Voting

In our first evaluation we used only one optimized classifier for classification similar to the implementation by Wei Li in 2007 [16], that we gained through optimization (see Section 4.5, Optimization). We noticed that we have either to choose between a fast algorithm with a high error rate or a slow classifier with good results. During our optimization phase we used a classifiers with moderate error rates and a good performance. However it is better to use many bad and fast classifiers instead of one good and slow one, like the *Random Forest* algorithm [4, 8, 9] in most cases. Figure 5 illustrates the voting process. Due to performance reasons we execute all classifier evaluations asynchronous and synchronize them after all classifier are executed.

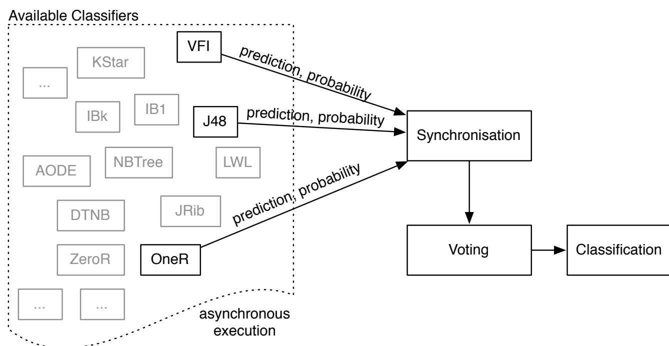

Figure 5: Classification voting process

In the first step our algorithm classifies a new instance with all classifiers, which are used for this decision, and calculates their probability. The user can decide, based on the primary 10-folds Cross-validation evaluation results of the training set, which classifiers are used. Each probability and prediction results of the chosen classifier are used by the voting system to gain a cumulative prediction. The voting system is a mathematical algorithm. in the first implementation iteration we used the majority prediction as a result. In case of a draw we used the class with the higher mean probability. However further tests showed that this naive method sometimes produces wrong voting results. For example we have 5 classifier with following predictions for a new instance: yes: 51%, no: 90%, yes: 53%, no: 95%, yes: 55%. We see that the majority decides for "yes" but the minority is quite sure that the classification correct is "no". Therefore we made further improvements of the voting function.

Let $x_{n,c}$ denote the probability of classification of class $c \in C$ of classifier $n \in N$. $A \subseteq M$ denotes the subset of classifier M which classifies an instance I with class c. We calculate the probability $p(c)$ of class c of instance I by the following algorithm:

$$p(c) = \frac{\sum_{a \in A} x_{a,c} + \sum_{b \notin A}(1 - \sum_{c' \in C \wedge c' \neq c} x_{b,c'})}{|N|}$$

The function $p(c)$ calculates a cumulative probability for class c taking the averages of the probability of c into account. However it uses the counter probability for all not matching classifier. If you have only two classes like "yes" and "no" you can use this simplified function $p(c)$.

$$p(c) = \frac{\sum_{a \in A} x_{a,c} + \sum_{b \notin A \wedge c' \neq c}(1 - x_{b,c'})}{|N|}$$

If we use this improved voting function to calculate the predictions of the example above we can calculate the probability for "yes" by

$$p(yes) = \frac{0,51 + (1-0,9) + 0,53 + (1-0,95) + 0,55}{5} = 0,348$$

and accordantly

$$p(no) = \frac{(1-0,51) + 0,9 + (1-0,53) + 0,95 + (1-0,55)}{5} = 0,652$$

We see that our improved voting system now decides for "no" against the majority result with a probability of 65,2%.

If you use only one classifier you gain performance, on the other hand you might have problems with false positives especially if you use fast classifier with a high error rate. These false positives will cause wrong reports e.g. single network packages can cause alarms. During our evaluation we eliminated all false positives by using multiple classifiers and a voting function. Note that critical decisions will most likely have different predictions of the classifiers and therefore will have a relatively low probability especially in context of other observations.

5 Evaluation

For evaluation purposes we used the network dumps [25] of the UCSB iCTF provided by the UCSB iCTF commit-

tee. The UCSB International Capture The Flag is a distributed security challenge, which goal it is to test the security skills of the participants [24]. Every team controls their own server with different services. Each game round (approximately every two minutes) the scorebot distributes a fixed length string (called flag) to each service of every team by using the service. Later the scorebot tries to collect the distributed flags. During this period of time, the other teams try to steal these flags by abusing and hacking the services of the opponent teams. The UCSB International Capture The Flag network dumps provide a good sample for testing data because they contain a great amount of different types of attacks mixed with regular traffic, e.g. the scorebot. Furthermore, the system has a limited range of well-defined services an it is therefore easier to determine the system status. By searching the traffic for flags, it is possible to determine the success rate of an attack; An attack is considered as successful if it contains a flag of the opponent team.

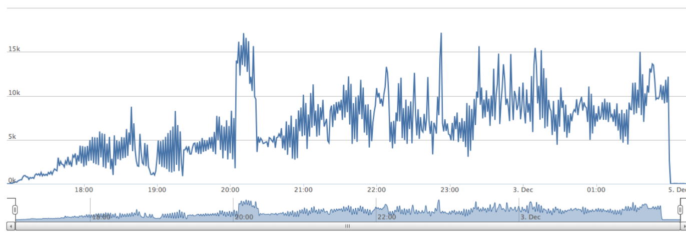

Figure 6: HTTP Network activity during iCTF 2011

Out of the 241 GB traffic dumps we collected information of over 6,000,000 TCP connections that we used to evaluate our approach. Figure 6 shows the network activity based on the HTTP protocol over the time. After parsing, validating and harvesting the data, we received a database of about 15 GB of data sets. Due to compression, abstraction and data deduplication of the raw data, the size of the database used in further steps was reduced to approximately 90%.

5.1 Prediction performance

In case of intrusion detection systems, a system classifies the traffic based on static rules and the number of detected events. However these systems are very limited in flexibility and as many attack patterns or methods are hard to model by/with static rules.

Based on our system design from Section 3 we developed a prototype for evaluation purpose. Our goal was to develop an (semi-)automatic mechanism to create models of attack patterns to detect those in the future. As mentioned in the previous sections, we use machine-learning techniques to create models and classify traffic.

We measure the prediction performance by counting all wrong classifications and classifications that produced an error during the prediction phase. To gain comparable results, we normalize all results by dividing the classification error count with the amount of all instances.

For evaluation, we looked for a sample, which is non-trivial and cannot be implemented with a static rule set: the scorebot. The scorebot is an automated script that distributes the flags during the contest. To avoid cheating by only allowing traffic of the scorebot and blocking all other teams, the scorebot changes its IP-address about every 100 requests. It uses an arbitrary IP-address of a randomly chosen team subnet-space. Further, it tries to obfuscate properties like the user agent string. However, it is an automatic procedure, which has the same behaviour in every round. Due to the properties of the scorebot it is hard to define a static rule:

- The Source IP-address may change during a session
- The User Agent String is non static
  - Mozilla/4.0 (compatible; MSIE 6.0; MSIE 5.5; Windows NT 5.1) Opera 7.02 [en]
  - Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.1) Gecko/2008071615 Fedora/3.0.1-1.fc9 Firefox/3.0.1
  - curl/7.8 (i686-pc-linux-gnu) libcurl 7.8 (OpenSSL 0.9.6)
  - Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/525.13 (KHTML, like Gecko) Chrome/0.A.B.C Safari/525.13
  - Mozilla/5.0 (Windows; U; Windows NT 5.1; en-GB; rv:1.8.1.6) Gecko/20070725 Firefox/2.0.0.6
  - Links (0.9x; Linux 2.4.7-10 i686)
  - random character string: [a-zA-Z]10,20
- Recurring requests to the same services (maybe with different parameters)

5.1.1 Size of the training-set

One of the important measures to evaluate machine learning algorithms is the size of the training set. If the training set is too small, one may receive many false positives and an inappropriate model. On the other hand, if the training model is too large it may take a while to generate a model and one has o manually classify many instances for training.

Our experiments show that the training size depends on the complexity of the signature of the model. Figure 7 shows the classification error rate depending on the training set. We use 2000 HTTP requests two hours after the start of the iCTF contest. We classified the traffic to determine if scoring bot generated the traffic.

We used a wide spread range of classifiers. We observed a drastic improvement of performance over time. We observed that the classifiers greatly differ regarding their error rate. Therefore our classification voting method gains a better prediction performance due to the usage of the majority of the classification. This reduces the rate of aberrations. As shown in Figure 8, our classification voting method will have at least the performance as the best classification algorithm.

Figure 7 showed how classification voting can improve the classification performance by using the strength of classification algorithms and compensate their weaknesses with the help of other classification algorithms. In some cases it is not possible to create such a classification diversity.

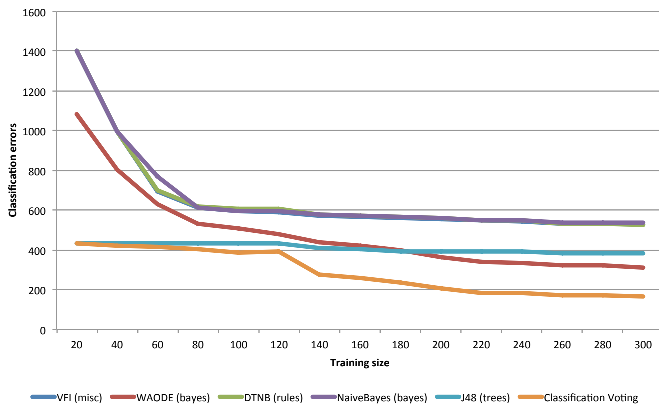

Figure 7: Classification errors depending on the size of the training sets

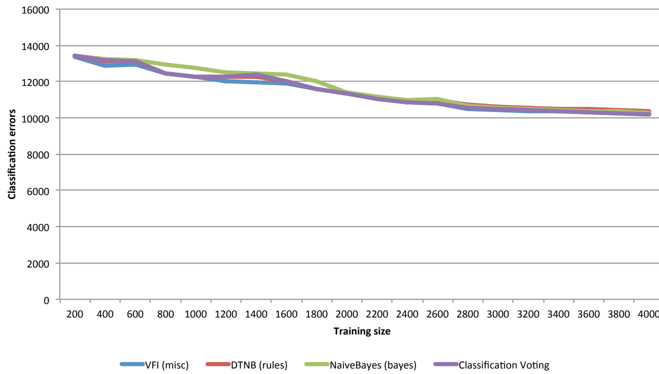

Figure 8: Classification voting with similar classifier depending on the size of the training sets

Figure 8 demonstrates such a case. We tried to identify the traffic of the penetration testing tool DirBuster. DirBuster is a tool that brute forces directories of web servers to find hidden applications or insecure configurations, which provide another attack vector. DirBuster is very simple to detect, because it uses the same URI pattern every time and generates many "404 Page not Found" responses. Due to the relatively low complexity of the signature of DirBuster, most machine learning classification methods will generate similar results. Our results suggest, the improvements of the classification voting mechanism are very limited, since the different classification algorithms may generate similar models. However, our experiments showed that classification voting is at least as good as the best of the used algorithms, by exploiting the advantages of several classification results combined into one model as described in Section 4.6.

5.1.2 Classification algorithms

According to our system design in Section 4.5 we evaluate all available classifiers. Therefore, we generate a model for each classifier and evaluate it with a 10-fold cross-validation and measure its performance. If a classifier does not deliver a result within a time frame of five seconds, it is ignored. As in the example described above, we try to detect the scorebot with the help of machine learning. We have used all available classifiers offered by WEKA with a training set of 2000 instances containing over 1500 positive instances of the scorebot. Figure 9 shows the distribution of the different classifiers.

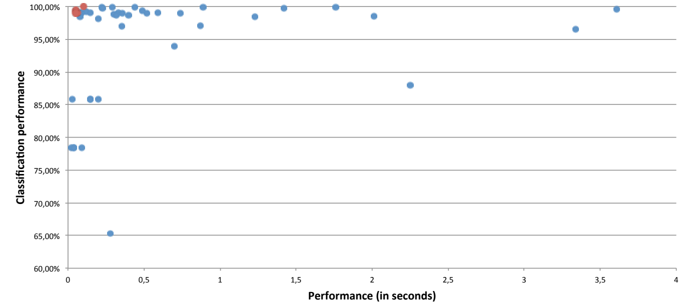

Figure 9: Performance and evaluation of different classification algorithms

We observed that many classifiers built a good model with a classification performance of 95% and above. However, we use only five or less algorithms for classification voting. In our example, we marked the chosen algorithms in red.

The results showed that many classification algorithms learn very quickly and gain good results by evaluation with the training set. However, it is important that these classifiers produce good and reliable results during classification of new traffic. Therefore, it is better not to use the best classifier, but to choose faster algorithms. Instead we have chosen faster algorithms with good results and reduce aberrations by voting.

5.1.3 Detection of penetration testing tools

To simulate real-life scenarios, we train our prototype to detect the most popular automated penetration testing tools. We choose a mix of general web scanner like Nikto and tools that are specialized to certain vulnerabilities (e.g. sqlmap). Table 1 shows a list of the tested tools, their classification results and the needed size of the training set to guarantee an adequate classification performance.

OWASP DirBuster Project DirBuster is a Java application which is designed to brute force directories and unsecurd files (e.g. configuration files) on web servers. It tries to guess the URIs of files or directories by using common file names. The detection of this tool is quite simple, because it uses a static user agent string containing the version of the tool (DirBuster-1.0-RC1 [...]).

Burp Suite Burp is a penetration tool for testing of web applications. Burp offers different components like a proxy, spider, scanner, intruder tool, repeater tool and sequencers. The user has full control over the software and can define its own test cases and parameter. Burp uses plain TCP streams without any unnecessary headers or information (basic configuration), because it tries to be as stealthy as possible. Due to this fact, it is very hard to generate a classification model to detect this tool, because you do not have a defined set of

test cases and no additional identifier like a user agent string.

Nessus Nessus is a commercial vulnerability scanner. It can monitor different hosts and scan them for security flaws e.g. malware and outdated software. It further can scan a web server for security flaws like Cross-Site Scripting or SQL Injections. The detection of this tool is very complex, because it used not a static signature like other tools. However one can identify Nessus by its payload, because every test contains somewhere the keyword *nessus*. If the test set is large enough, it is still possible to train the algorithms to detect this tool by learning all its test cases.

sqlmap sqlmap is an open source penetration testing framework that scans a web site automatically for different database vulnerabilities e.g. SQL injection flaws. It uses a wide range of patterns and techniques to detect security flaws. It supports all popular database management systems. The detection of this framework is quite simple, because it uses a static user agent string (sqlmap/1.0-dev (http://sqlmap.org)).

Nikto Web Scanner Nikto is an Open Source web scanner that tests a web server for multiple items, like outdated version or common configuration flaws. This tool was not designed to be stealthy, therefore it is quite simple to identify the requests by the user agent string (Mozilla/5.00 (Nikto/2.1.5) (Evasions:None) (Test:XXXXXX)). However Nikto adds the number of the test to the user agent string, therefore the classifier have to learn each test or use an abstract model of the user agent string which does not contain the test number.

Tool	*Complexity (Signature)*	*Training size*	*Success rate*
DirBuster	low	10-50	high (99%)
Burp Suite	none (plain TCP)	5000+	low (40%)
Nessus	complex	5000+	medium (85%)
sqlmap	low	10-50	high (99%)
Nikto	low/medium	10-50	high (95%)

Table 1: Detection of different penetration testing tools

The quality of the generated model strongly depends on the attribute selection and the provided information. The Burp Suite doesn't offer any additional information like HTTP Header, because it is using plain TCP connection, therefore it is very hard to generate an adequate model. In contrast Nessus uses different predefined test cases, which can be used to create some kind of a behavior model. If one run the tool on a testing system to capture the traffic, it is possible to create a model of all included test cases. This model can detect all used test cases in a production environment therefore the results are quite good even if the signature is not trivial. This example demonstrates the advantages of our approach, because it is very hard to create a signature for Nessus without a simple characteristic like the user agent string (e.g. like DirBuster).

Figure 10 visualizes the classification performance of the different classifier during training phase. Penetration tools

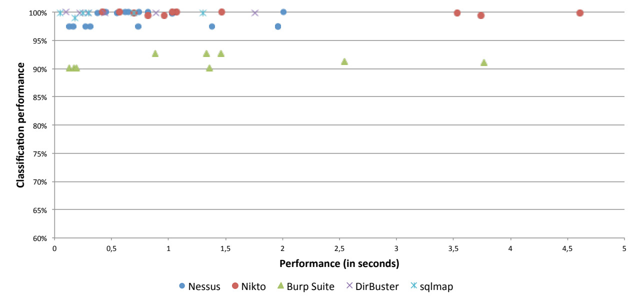

Figure 10: Classification performance on different penetration testing tools

with a simple signature e.g. a fixed user agent string are easy to learn, therefore fast and good classifiers are easy to find. Nevertheless there are tools like Burp that obfuscate their traffic. Due to their obfuscation, it is very hard to create an adequate model during the training phase. Therefore the models are fuzzy and can be improved with classification voting to avoid false positives.

5.2 Prototype performance

Performance is an important factor of implementing intrusion detection systems. If the system is too slow it may cause connection drops or bad latency. Therefore, it is important that such a system makes quick decisions. In our test setup, we used 10.000 new connections. As before, we tried to detect the scorebot with five different classifiers using classification voting. After optimization, the system automatically picked five algorithms (see Section 4.5): VFI, HyperPipes, NaiveBayesUpdateable, OneR and J48.

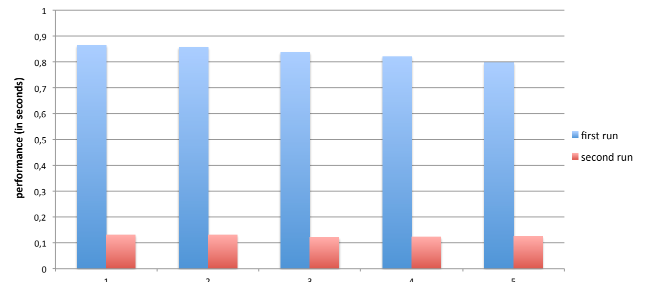

Figure 11: Classification performance of the framework

Figure 11 shows the performance of this set of classifiers. Our result suggest that the slowest part of classification is the creation of the model, which has to be done for all five classifiers. The system has to learn from the training set in order to classify the new traffic. This step has to be done once and can be reused later. Therefore, our prototype stores the generated model in a cache in the file system. Once the system learned the behaviour of the scorebot, the classification is very fast, even if we apply five different classifiers instead of a single one. Note that the performance of the learning phase strongly depends on the certain classifier. Therefore it is very important to pick faster classifiers in order to guarantee a good performance.

5.2.1 Configuration Set

In our evaluation section we have evaluated the traffic dumps of the iCTF contest of 2011 with a total size of 241 GB traffic with over 6,000,000 TCP connections during a time span of 9 hours. The traffic was captured by the game organizers using the eth interface. During the contest there were different peaks with about 18k HTTP request per minute that have been analyzed.

These traffic data was analysed by a MacBook Pro "Core i7" 2.2 15" Early 2011 with 8 GB RAM. After generating the test set the traffic was evaluated in real-time with an average delay of about 15ms. The main bottle neck was the generation of the test set that may take a long time depending on the size of the test set.

5.3 Limitations

Machine Learning is quite depending on the training set and the chosen algorithm. We propose a new approach that is independent of the knowledge of the system and the concrete machine learning algorithms. This enables a fast classification without expert knowledge. However it is a quite limited factor to generation of the test set. The trained model is evaluated against the a test set, but if the system changes over the time the rate of false positives may increase and the system has be trained again. The proposed method is quite useful in a static environment like a capture the flag contest where the system is limited and static. In these environments it is often important to have some kind of heuristics that overs a fast was way of create a preliminary profile of an attacker's profile and use this survey in a later step for suitable defence strategy.

Moreover with a small training set the generated model is often not suitable to the real system. Therefore a larger training set is mandatory for a reasonable result. While it is quite obvious that a larger training set may generate a better model, however you have to classify the traffic by hand which take a while in larger environments and slows down the initial results.

6 Conclusion and Perspectives

This paper provides an overview of existing network-based defence mechanisms and explains how machine learning can be used to improve existing techniques to achieve better detection rate. This paper explains the importance of machine learning in the area of intrusion detection systems and sums up the current research in this field.

6.1 Contributions

The purpose of this study is to evaluate the applicability of machine learning in network traffic classification. We introduced a new approach for signature-based detection in combination with behaviour-based detection. However our implementation reduces the complexity and the required knowledge to generate new signatures based on decisions on sample traffic. We used state of the art machine learning algorithms for packet classification. This process is transparent to the user, therefore no special knowledge about data mining is needed. To reduce false positives we introduced a new technique called classification voting. By using different classifiers and a solid majority voting mechanism we gain significant better results than using just one classifier. We use an automatic learning and optimization process in order to generate models for classifications of new network packets. The results of this evaluation show that it is feasible to learn patterns and use these patterns for classification of new traffic.

6.2 Future Research

We have shown that it is possible to classify new traffic by using machine learning techniques. In many cases it is often very helpful as defender to know what an attacker will do in his next steps. Machine learning approaches offers many meta information which can be used for *predictions* of the next steps of an attacker.

Other research by Steven M. Bellovin showed that is possible to count the NATed-Hosts [3]. Our approach doesn't necessarily use IP addresses in order to determine a host. We want to extend our existing approach to *cluster* traffic to split up traffic based on different attributes. Due to the fact that we are using different attributes than the IP address it would be possible to count the hosts behind a NAT with machine learning.

By using existing information gained by machine learning we want to create attack profiles based on traffic packets of one host. These profiles can be used to support the administrator with his decision about the next defence steps against intruders.

References

[1] I. Androutsopoulos, J. Koutsias, K. V. Chandrinos, G. Paliouras, and C. D. Spyropoulos. An evaluation of naive bayesian anti-spam filtering. *arXiv preprint cs/0006013*, 2000.

[2] M. Barreno, B. Nelson, R. Sears, A. D. Joseph, and J. Tygar. Can machine learning be secure? In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 16–25. ACM, 2006.

[3] S. M. Bellovin. A technique for counting natted hosts. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 267 – 272. ACM, 2002.

[4] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, October 2001.

[5] C. Cipriano, A. Zand, A. Houmansadr, C. Kruegel, and G. Vigna. Nexat: a history-based approach to predict attacker actions. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 383–392. ACM, 2011.

[6] D. Endler. Intrusion detection. applying machine learning to solaris audit data. In *Computer Security Applications Conference, 1998. Proceedings. 14th Annual*, pages 268–279. IEEE, 1998.

[7] J. Erman, A. Mahanti, M. Arlitt, I. Cohen, and C. Williamson. Offline/realtime traffic classification using semi-supervised learning. *Performance Evaluation*, 64(9–12):1194–1213, October 2007.

[8] T. K. Ho. Random decision forests. In *Proceedings of the Third International Conference on Document Analysis and Recognition*, volume 1, pages 278–282, August 1995.

[9] T. K. Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, August 1998.

[10] M.-Y. Huang, R. J. Jasper, and T. M. Wicks. Large-scale intrusion detection framework based on attack strategy analysis. In *Computer Networks: The International Journal of Computer and Telecommunications Networking*, pages 2465–2475, December 1999.

[11] R. A. Kemmerer and G. Vigna. Intrusion detection: a brief history and overview. *Computer*, 35(4):27–30, 2002.

[12] U. Lamping, R. Sharpe, and E. Warnicke. *Wireshark User's Guide for Wireshark 1.9*, 2004–2012.

[13] A. Lazarevic, L. Ertoz, V. Kumar, A. Ozgur, and J. Srivastava. A comparative study of anomaly detection schemes in network intrusion detection. In *Proceedings of the third SIAM international conference on data mining*, volume 3, pages 25–36. Siam, 2003.

[14] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, volume 10, page 707, 1966.

[15] G. Levitin. Optimal defense strategy against intentional attacks. *Transactions on Reliability, IEEE*, 56:148–157, March 2007.

[16] W. Li and A. W. Moore. A machine learning approach for efficient traffic classification. In *Proceedings of the 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 310–317, 2007.

[17] Y. Liao and V. R. Vemuri. Using text categorization techniques for intrusion detection. In *Proceedings of the 11th USENIX Security Symposium*, pages 51–59. USENIX Association, 2002.

[18] S. Mukkamala, G. Janoski, and A. Sung. Intrusion detection using neural networks and support vector machines. In *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on*, volume 2, pages 1702–1707. IEEE, 2002.

[19] T. Nguyen and G. J. Armitage. A survey of techniques for internet traffic classification using machine learning. *Communications Surveys & Tutorials, IEEE*, 10(4):56–76, 2008.

[20] U. of Waikato. *Weka 3: Data Mining Software in Java*, Sept. 2012.

[21] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435 – 2463, 1999.

[22] S. D. Shanklin, T. E. Bernhard, and G. S. Lathem. Intrusion detection signature analysis using regular expressions and logical operators, Nov. 26 2002. US Patent 6,487,666.

[23] C. Sinclair, L. Pierce, and S. Matzner. An application of machine learning to network intrusion detection. In *15th Annual Proceedings Computer Security Applications Conference*, pages 371–377. IEEE, 1999.

[24] G. Vigna. The 2010 international capture the flag competition. *Security & Privacy, IEEE*, 9(1):12–14, Jan.-Feb. 2011.

[25] G. Vigna. ictf 2011 traffic capture. http://ictf.cs.ucsb.edu/data/ictf2011/pcaps/, September 2011. [Online; accessed 05.04.2013].

[26] L. Wehenkel. Machine learning approaches to power-system security assessment. *IEEE Expert*, 12(5):60–72, 1997.

[27] N. Williams, S. Zander, and G. Armitage. A preliminary performance comparison of five machine learning algorithms for practical ip traffic flow classification. *ACM SIGCOMM Computer Communication Review*, 36(5):5–16, Oktober 2006.

[28] D.-Y. Yeung and C. Chow. Parzen-window network intrusion detectors. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 4, pages 385–388. IEEE, 2002.