Bro Covert Channel Detection (BroCCaDe) Framework: Design and Implementation

Hendra Gunadi (Hendra.Gunadi@murdoch.edu.au),
Sebastian Zander (s.zander@murdoch.edu.au)

17 November 2017

Abstract

In this report we present the design for the Bro Covert Channel Detection (BroCCaDe). We propose the framework structure and describe the different components in detail. Then we describe how the components interact with each other, followed by a description of the implementation of the covert channels to test our infrastructure. Finally, we provide an example script to give an example on how to use our infrastructure.

1 Introduction

Based upon technical report [9], we describe the design of the Bro Covert Channel Detection (BroCCaDe) Framework. Section 2 explains the design of our covert channel detection framework and Section 2.2 provides an overview of how the components interact within the framework. In Section 3 we describe how we implemented the covert channels used to test BroCCaDe. Finally, Section 4 provides an example how BroCCaDe works before Section 5 concludes the report.

2 Design

In this section we will explain in detail the design and of BroCCaDe. We start by describing the overall structure and then discuss the details of each component. We will further go into detail of the implicit .bif files which are contained in each of the plugins. The final part of the section describes the most complex plugin in our infrastructure, i.e., the Analysis Plugin. Before we continue, we explain the important concepts of window size and step size which underpin how our whole infrastructure works (both of which operate on a unidirectional basis in line with our definition of unidirectional flows, see Section 3 technical report [9]):

- **Window Size:** the number of packets that will be used in one run of the analysis. Currently, window size only applies to the KS test and Autocorrelation analysis. For Entropy and Multi Modal analysis, the data is stored in a histogram, hence, it is not limited to the window size. Similarly for CCE the tree stores the count of patterns since Bro (with the plugin) started (and is not limited to the window size). For the Regularity analysis the window size is used, but it is not strictly the number of packets that is used in the analysis since the analysis spans across several windows.

- **Step Size:** a number which indicates the interval (in number of packets) before the metric calculation is triggered. The count of packets is per unidirectional flow. Figure 1 illustrates the difference between window size and step size. In the figure, the step size is set to 50, i.e., the analysis is triggered every 50 packets in a unidirectional flow. In Figure 1(a), the window size is smaller than the step size so there are some packets that are not included in the analysis. In Figure 1(b), the window size is larger than the step size so the windows are overlapping.

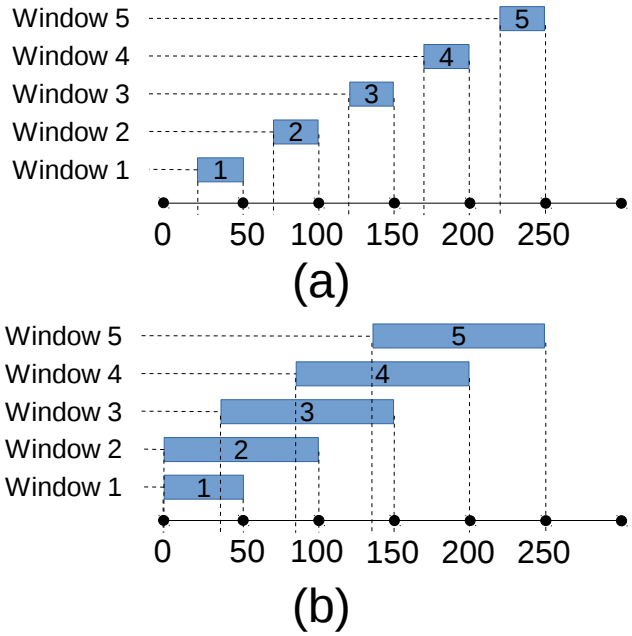

Figure 1: Illustration of Step Size and Window Size: (a) Window Size is smaller than Step Size (b) Window Size is larger than Step Size

2.1 Components

The BroCCaDe extension does not require any modifications of Bro's internals, as can be seen from the greyed boxes in Figure 2 which indicate the modifications required for BroCCaDe. The design of BroCCaDe takes advantage of the flexibility of Bro's plugin design. In particular, Bro plugins are designed to extend Bro script(s) with various functionalities akin to native functions in Java. In Bro events provide another way of communicating results back to the script level rather than just returning values. A plugin consists of a lot of files, but we are only interested in the Bro script, C++ source code, and the .bif file. The .bif file contains a list of functions and events definitions implemented by the plugin, and it works as the interface between Bro scripts and the Bro engine.

BroCCaDe basically consists of five components, but the plugin used for training is not shown in Figure 2 due to its special nature:

1. Custom Bro scripts which handle events raised by various plugins and bind together the functionalities of different types of plugins described below.

2. The Feature Extraction Plugin which extracts feature values from incoming packets. The plugin will then pass back the feature values to the script level as an argument of an event. After a feature value is extracted by the Feature Extraction Plugin, the information of interest is then passed to the script via an event.

3. The Analysis Plugin takes the features extracted by the Feature Extraction Plugin, and based on the feature values calculates a detection metric. This plugin will pass the metric values to the script layer via an event. The script can compare the metric value against a simple threshold to determine whether a

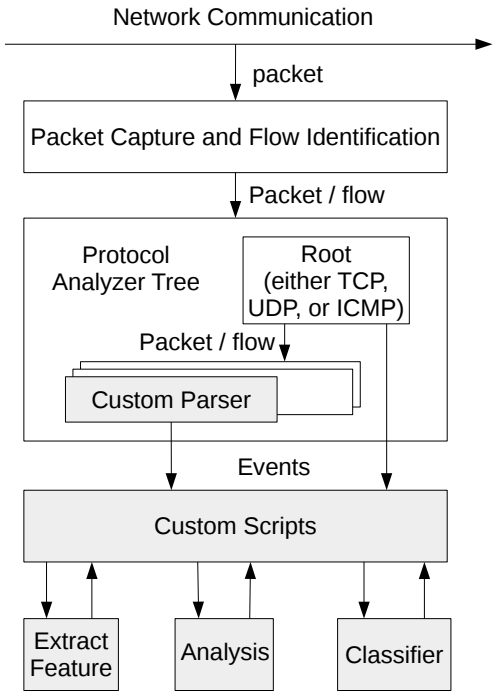

Figure 2: Internal Bro Communication (BroCCaDe modifications highlighted in grey)

packet flow contains a covert channel. Alternatively, the script can pass the metric value to the classifier plugin (see below) to classify a flow based on one or more metrics.

4. The Classifier Plugin takes metric values computed by one or more Analysis Plugins and determines whether a packet flow contains a covert channel. There are two ways that this plugin can communicate the outcome of the classification: it can either return the class directly as a return value or raise an event which contains the classification result. If a covert channel is detected, then appropriate actions can be taken as defined in the script, such as logging the event or even alerting the network administrator via email. Currently we only implemented one classifier plugin which uses a C4.5 decision tree classifier.

5. The Training Plugin is used for training a classifier or gathering auxiliary data for an analysis plugin and is not involved in the normal execution of BroCCaDe. Currently, there are two plugins which fall into this category:

   (a) Plugin used to train models for the Classifier Plugin: The role of this plugin is to gather analysis results, making use of the Feature Extraction Plugin and the Analysis Plugin, and then output the data in a format that can be used to train a classifier. Our current plugin outputs the data in ARFF format [1] which can then be used to train a C4.5 tree classifier in WEKA [6] (for more details see Section 2.2.2). Note that the main focus of this plugin is to generate the necessary output (analysis metric values) that can be used as input to train classifiers. In case another classification technique requires a different input format the output format can be changed easily by creating a new variant of this plugin.

   (b) Plugin for gathering auxiliary data for the analysis plugin: Depending on the analysis involved, there can be several of this kind of plugin. In BroCCaDe, there is a plugin which gathers feature value (such as packet lengths) and produces two types of information: the normal data for the Kolmogorov-Smirnov test and the intervals for the equiprobable bins for histogram and CCE. The plugin needs to be applied on the training data and makes use of feature extraction plugin(s) as

required to gather feature values. To compute the normal data for the KS test, the plugin stores feature values for each unidirectional flow and then outputs the feature values when a flow ends. The plugin also stores the feature value of the whole training data set to compute the equiprobable bin intervals.

The underlying concept behind BroCCaDe's design is that the custom script has the necessary event handlers, which will call functions from the different plugins as required. The script is the component that ties the whole infrastructure together. We decided on three different types of plugins because we wanted to make the architecture as modular as possible without making it more complex than necessary. We need the Feature Extraction Plugin to extract the features from the network protocol data. The Analysis Plugin is required because there are a lot of ways in which we can analyse the features from the Feature Extraction Plugin. The Classifier Plugin is used to compare metrics against a simple threshold or extracted header fields extracted against a specific value, or it can use a machine learning classification technique to determine whether the metric values indicate the existence of a covert channel.

For each type of plugin there can be many plugins with different functionality, e.g. in our current setting, we have three feature extraction plugins: a protocol parser plugin to extract the TCP urgent pointer, a plugin to extract the first 4 bytes of the ICMP payload, and a plugin to calculate the inter-arrival times between packets of one flow. However, currently there is only one analysis plugin which implements the different analysis methods.

2.1.1 Feature Extraction Plugin

The feature extraction plugin is our first point of interaction with the main part of Bro. For the purpose of BroCCaDe, we loosely define the feature extraction as an extension which returns a feature. This means that the feature extraction is not strictly in the form of a plugin or a script. Generally there are three ways to extract feature values:

1. Use the script directly because Bro has already provided the feature value for us. For example, if packet length is the required featured, the default Bro event *new_packet* already provides IP header length. This means, this plugin is optional and only needed to extract non-standard features. It is not required if only standard features are needed, e.g., since IP header length is already provided, no additional plugin is needed to extract this feature value.

2. Make use of the information from the custom script. Since Bro already provided so many events, we might as well make use of this information. We have implemented two feature extraction plugins, one extracts inter-arrival times and the other extracts some bytes from the payload of each incoming packet, e.g., a plugin which calculate inter-arrival time feature based on the connection duration; or

3. Parse the packet. We create a plugin which parse a particular network protocol of interest in case Bro does not have support for it already. For example, the URG flag is already provided by Bro but to get the URG pointer we have to parse the TCP packet. In other words, if we are interested in the protocols and events that Bro already provides, then we do not need this plugin. We also may need to extend existing protocol parser if there is a parser but it cannot deliver the event we need.

The more detailed description of our current feature extraction modules will shed some light on this:

URG_flag and URG_ptr: Even though Bro extracts the TCP URG flag, it does not extract the TCP URG pointer. Therefore, this particular feature extraction was implemented in the form of a protocol parser which will raise the feature event (containing URG_flag and URG_ptr) to be handled by the analysis plugin.

Packet Length: Since Bro already extracts the packet length, the packet length feature extraction is simply a script. We piggyback on Bro's new_packet event which provides the packet header for each incoming packet, and use an event handler which takes just the packet length and passes it to the analysis plugin.

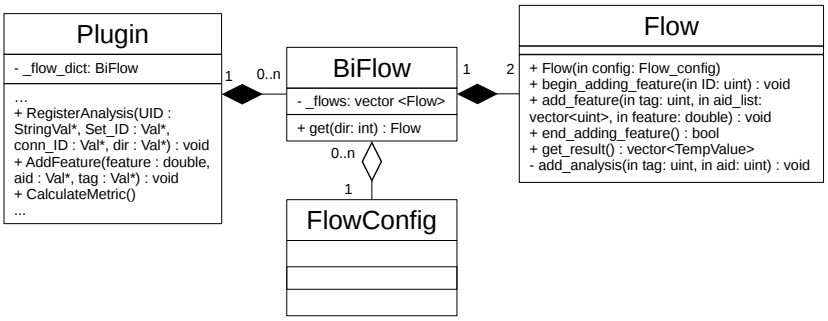

Figure 3: Class Diagram for Analysis Plugin

Inter Arrival Time: Unfortunately Bro does not maintain the packet inter-arrival times. Nevertheless, Bro keeps track of the duration of each flow or connection, so we still can utilise the new_packet event with a little bit of processing. We keep the previous duration from when the last packet of a particular flow arrived. Whenever a new packet arrives, we calculate the difference of the current duration and the previous duration which is the inter-arrival time between the current and the previous packet. The feature extraction for inter-arrival time is a combination of script and plugin, where the script gets the value of the connection duration and the plugin calculates the time difference.

Ping Tunnel: The way to detect the PingTunnel is by checking whether the magic number appears at the start of the ICMP payload (see Section 5.3 of technical report 20171108A [9]). Therefore we can piggyback on Bro's events ICMP_echo_request and ICMP_echo_reply which give us the ICMP payload, and from that extract the first four bytes with the help of a plugin. Feature extraction for PingTunnel is also a combination of script and plugin, where the script obtains the ICMP payload and the plugin extracts the first 4 bytes of the payload.

TTL: For each incoming packet, Bro already provides the TTL value of the packet.

2.1.2 Analysis Plugin Infrastructure

Figure 3 shows the structure of the Analysis plugin. From the figure we can see that mainly the plugin is used to provide functionalities to the script level, and to hold references to the bidirectional flows which are identified using Bro's unique connection UID. Subsequently, we will refer to these references as flow dictionary. Although Bro's definition of flow is bidirectional, we have to inspect each direction separately to determine whether it contains a covert message. So, at the heart of our analysis plugin is the Flow class which holds feature values for each unidirectional flow (each direction).

Other functions of the plugin involve setting up default values which will be stored by FlowConfig. FlowConfig is a global configuration for all of the unidirectional flows which is initialised with default values when the plugin is started, but the values can be changed later on. Since the Plugin class only holds a flow dictionary of bidirectional flows, the reference to the FlowConfig class is passed into BiFlow, which in turns will be passed on to the Flow class during initialisation. FlowConfig holds, among which, various parameters used for the analysis methods implemented, e.g., the number of autocorrelation lags, the step size, window size, etc. It also contains mapping between analysis methods and data container, and the Analysis ID which is defined in the .bif file. The concept of both data container and Analysis ID will be explained further in the section.

We shall look in closer detail how the Flow class works. Figure 4 details the class diagram for classes used in the Analysis plugin. In particular, it expands on the Flow class and its associated classes. Flow holds one or more data containers (Data_Container) and feature analysers (FeatureAnalyzer).The information in FlowConfig is used by Flow as the parameters to initialise the data containers and the feature analysers. When the metric

calculation is triggered but the result is not collected yet, the Flow object will have some outstanding metric values. These values are stored as a list of TempValue.

The Data_Container is a generic class which provides a layer of abstraction to add features into a data container. Specific data containers derived from Data_Container have their own way of managing the data and interfacing with the analysis objects. Some of the data containers, such as Histogram and Pattern_Data, rely on the Bin_Strategy to determine how to assign incoming features a bin number. Bin strategy, to put simply, is akin to a black box which will return a bin number when given a feature value as an input. Currently we have two bin strategies in place, *Bin_Strategy_Null* and *Bin_Strategy_Interval. Bin_Strategy_Null* uses the value directly as the bin number, e.g., a feature of value *5* will also have *5* as its bin number. *Bin_Strategy_Interval* will check whether the feature value falls into a certain interval, e.g., if we have *3* intervals $\{[0,3),[4,5),[6,10)\}$, then a value of *5* will be given a bin number of *1* (the index of the bin starts from *0*). As a side note, the red line in the figure bears no significance. We just colour it differently to make it easier to trace the lines.

In the following we briefly describe all existing containers:

Null_Data is the simplest type of data container which just returns the current value that it holds.

Raw_Data is also simple, it just stores the data as a list of feature values, limited by the window size.

Regularity_Data is a data container for the regularity metric. For the calculation of the regularity metric, where we require the current window and standard deviations of the previous n windows where specified n is the number of windows specified as a parameter to the analysis (see technical report 20171108A [9] Section 6.6). We decided to separate this implementation from the Raw_Data so that we do not have to store all of the data points. The container is based on the rapid calculation of standard deviation $\sigma = \sqrt{\frac{N \cdot s_2 - s_1^2}{N}}$, where $s_j = \sum_{k=1}^{N} x_k^j$, $j = \{1,2\}$, N is the number of features, and x_k is the feature at index k. With this rapid calculation, we just have to maintain the values of the sum, the square sum, and the number of data, as opposed to calculating the standard deviation over and over again based on all feature values.

Histogram is a data container which counts the number of occurrences for each bin. Inherently, this class requires that we have a bin strategy in place. For each incoming feature value, the histogram object will identify the bin number from the bin strategy and increase the count for that bin number.

Pattern_Data is a data container which contains a tree to count pattern length (as described in Section 6.3 of technical report 20171108A [9]). This data container also requires a bin strategy to classify incoming feature values.

Note that wherever possible, i.e. in all cases where objects do not require a local copy to be maintained, the objects are not duplicated when passed – only references are passed. For example, Flow_Config is only initialised once and a reference to it is passed to each Flow object for each unique flow. Similarly, the data containers are not copied, but when analysis objects need the data, a reference to the required data container is passed to them. Our design is driven by the fact that different feature analysers may share the same data containers, and by working with references rather than concrete objects we can avoid redundant copies.

The FeatureAnalyzer is also a generic class which is further specialised for specific analysis. Figure 5 shows the relationship between the data containers and feature analysers, as each feature analyser only uses one specific type of data container, e.g. the Kolmogorov-Smirnov analysis uses the raw data container, the Entropy analysis and Multi-Modalilty analysis use the histogram, and so on. The details for each of the feature analysers are presented in Section 3.2.

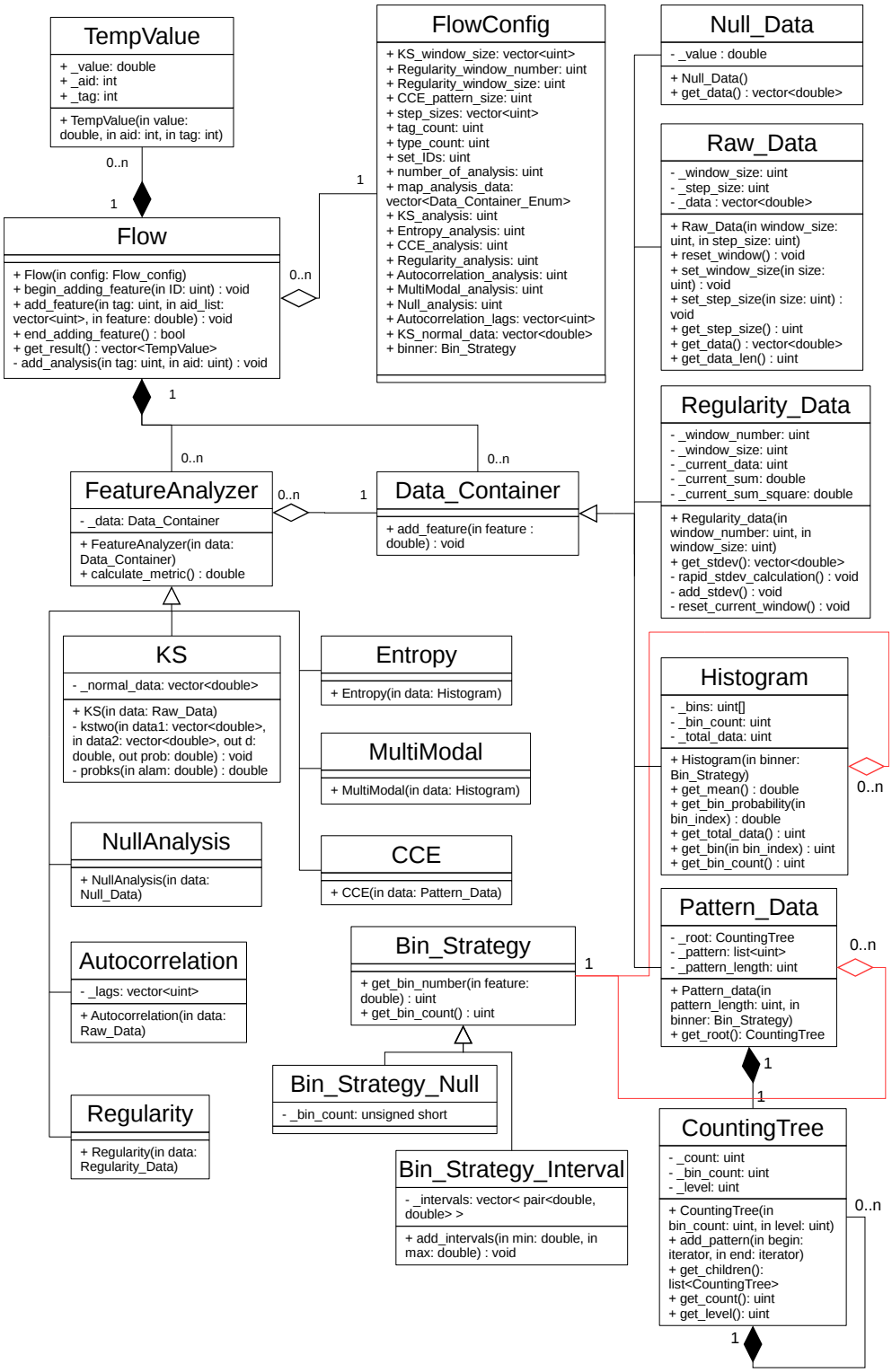

Figure 4: Class Diagram for the Classes Used in the Analysis Plugin

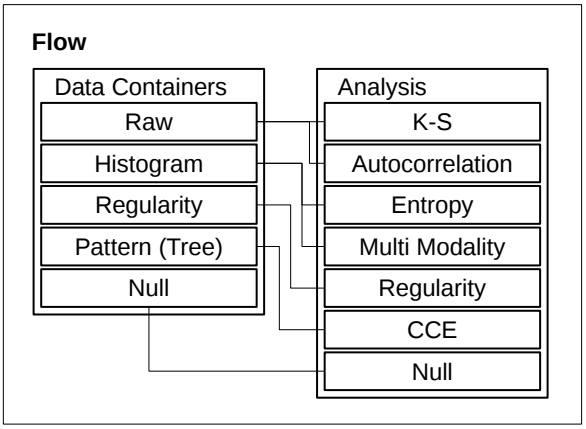

Figure 5: Right: Internal of Flow Class

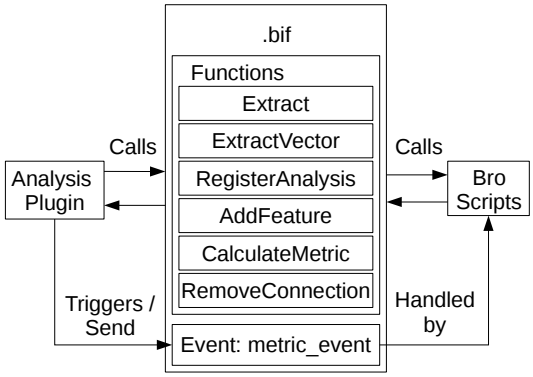

Figure 6: Plugin Structure

2.1.3 .bif File Interface

For every plugin we have to specify an interface between the plugin and the Bro script using a .bif file (see Figure 6 for the structure of the Analysis Plugin). We could bypass the .bif interface as long as we register methods and events properly so that they are visible from both the plugin and the rest of Bro. However, this would mean we would redefine something that Bro already provides. The .bif file contains method definitions that the plugin is defining and events that can be raised. From the script level, we can invoke the methods defined in a .bif file, and we can define an event handler to catch any events defined in the .bif file. The .bif file is written in a script language which is not the same as Bro script. The script in .bif files is compiled into C/C++ code and then this code is compiled along with the rest of the plugin. While technically it is possible to put code directly into a .bif file, we decided to use the .bif file purely as interface. This way we can focus on familiar C++ code at the plugin level rather than having to deal with code in another scripting language.

Each event handler for a particular feature will contain the information about the Bro unique UID (Bro has a globally unique identifier attached to each connection), 5-tuple ID, direction and feature value(s). Further identification must be provided when passing feature values to the Analysis Plugin using the following identifiers:

- **Feature ID:** This is a unique ID that identifies a feature, such as packet length, URG_flag, inter-arrival time. It is used to specify features that are analysed and group them into feature sets.

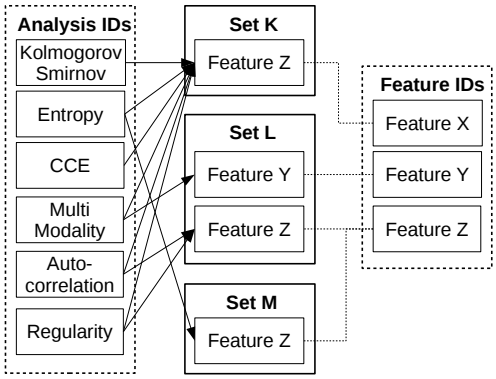

Figure 7: Relationship between Feature ID, Analysis ID and Feature Set ID

- **Analysis ID:** This is a unique ID for each type of analysis implemented (see technical report 20171108A [9]). It is used to specify the types of analysis that should be carried out for a particular feature in a particular feature set.
- **Feature Set ID:** This is a unique ID for a set of features (identified by their feature IDs). Also, for each feature the feature set defines the analysis types applied to the feature (identified by the Analysis IDs). In other words, the Feature Set ID is used to identify a particular set of analysis applied to a particular set of features (feature sets must be specified prior to starting an analysis). This construct enables the application of various analysis metrics to various features. For example, we might apply only entropy analysis to packet length in one feature set and choose to apply only autocorrelation analysis to packet length in another feature set.

Figure 7 illustrates the relationships between these identifiers. In the figure, we have the following analysis: Kolmogorov-Smirnov, Entropy, CCE, Multi-Modality, Autocorrelation, and Regularity analysis. We extract three types of feature from incoming packets, identified as feature *X*, *Y*, and *Z*. We also have four types of covert channels we are interested in detecting, hence the four feature set *K, L,* and *M*. Put it simply, a feature set can contain an arbitrary number of features and we can use a combination of analysis to analyse each feature. There are several things that we want to highlight from the figure:

- A particular feature may appear in many feature sets, in our example feature *Z* appears in sets *L* and *M*. This may be useful in cases where we can detect several covert channels based on one particular feature, e.g., we can analyse both rate channels and inter-packet gap channels based on packet inter-arrival time.
- A particular feature set may have an arbitrary number of features, for example set *L* has two features while the other feature sets only have one feature.
- Different combinations of analysis may be applied to different features in one feature set. In Figure 7, we can see that for set *L* only multi-modality analysis is applied to feature *Y* while autocorrelation and regularity analysis are applied to feature Z. Furthermore, different combinations of analysis may be applied to the same feature in different feature sets. For example, entropy is applied to feature *Z* in set *M* while autocorrelation and regularity are applied to feature *Z* in set *L*.

The feature ID is defined in the custom script, while the analysis ID is defined in the .bif file of the Analysis Plugin. We implemented the feature ID at the script level because the Analysis plugin does not have to know the type of feature or where it comes from, as long as it is able to distinguish between different features (essentially for the Analysis Plugin the Feature ID is unique opaque ID). The analysis ID is different in that the Analysis plugin needs to map it to an actual analysis object (and corresponding data container). Hence, the Analysis IDs are defined in the plugin .bif file which is visible both to the script and the plugin.

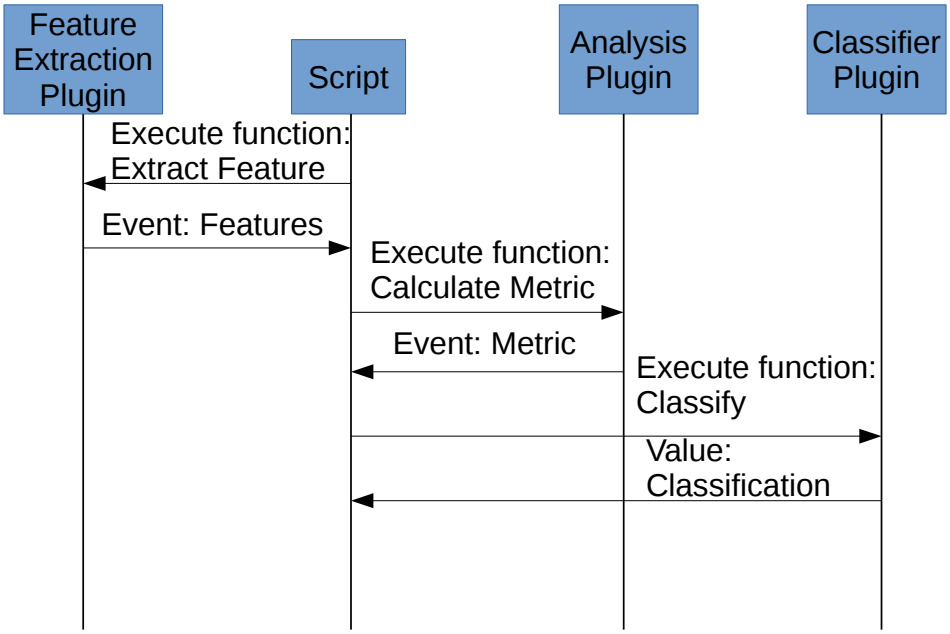

Figure 8: Communication flows between different plugin types within the framework

2.2 Flow of Information

Figure 8 shows the communication between the components when a packet arrives. Whenever Bro captures a packet from the network, it does flow identification (we have not modified Bro's policies to identify a packet as part of a flow). Bro identifies a flow based on the 5-tuple of source and destination address, protocol, source and destination port, also a protocol specific criteria, e.g. when a TCP flow reaches the finished state. The flow is then passed along with the packet to the analyser tree, which will inspect the packet and raise events of interest, e. g. new_packet for every captured packet, ICMP_echo_request when the captured packet is an ICMP echo request, and so on. Since we can hook into this analyser tree, it gives us the flexibility to extract values from packets or flows that Bro does not provide by default.

The execution then continues with the Feature Extraction Plugin extracting the feature from the packet if the feature of interest is not provided by Bro. In the case where we need to parse the packet directly, the Feature Extraction Plugin with a parser (a specialised version of the Feature Extraction Plugin) parses the packet and extracts the feature value(s). The Feature Extraction Plugin then queues a corresponding new event with the corresponding feature value(s), such as the urgent flag and urgent pointer. The event will fire the script's event handler, which will then pass feature value(s) to the Analysis Plugin.

Since Bro is event-driven, we need to clearly specify the event handler for each set of features. The details of this will be explained in Section 4. After the handler passes the feature value(s) to the Analysis plugin, the Analysis Plugin will do the analysis of the feature value(s) and may raise an event containing the analysis results. Upon receiving the (vector of) metric values from the Analysis plugin, one can define how to interpret the result. For example, one may want to tag flows for which the entropy metric has high values. Currently there are two ways in which we can do this. The vector of metric values can be interpreted in the script level or it can be passed to the Classifier Plugin (C4.5 decision tree classifier).

2.2.1 Communicating with the Analysis Plugin

To pass the feature values to the analysis plugin, the script has to start by notifying the plugin to begin the transaction by calling

FeatureAnalysis::RegisterAnalysis(UID : string, set_ID : Analysis_ID, id : conn_id);

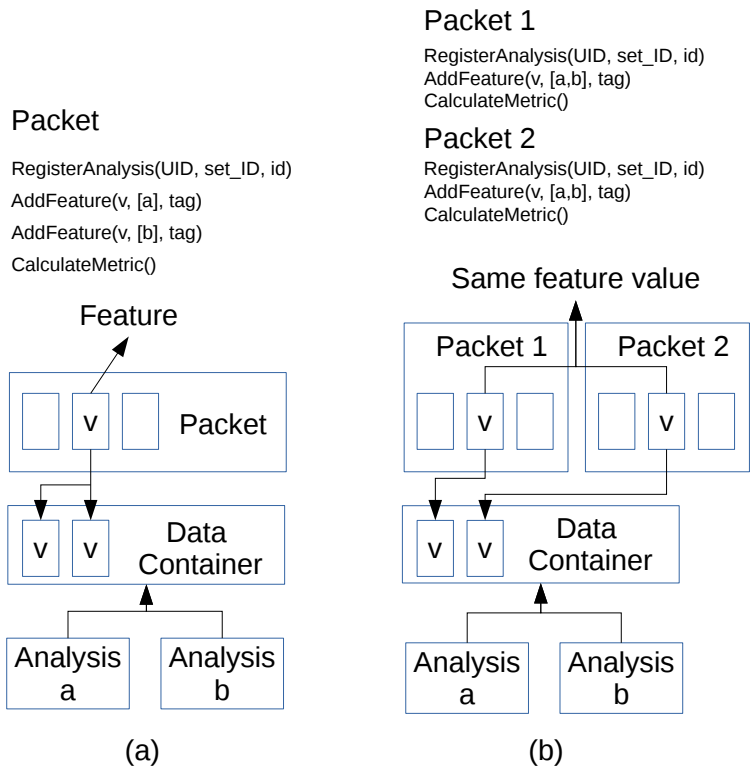

Figure 9: Adding a feature value: (a) adding a duplicate value (b) adding the same value for multiple features

where "UID" is the Bro unique identifier, "set_ID" is the analysis ID, and "id" is the 5-tuple id. Upon calling this function, the Analysis Plugin will fetch or create a flow object depending on whether the flow object exists or not. We can then pass feature values to the Analysis Plugin via

FeatureAnalysis::AddFeature(value : double, aid : analysis_vector, tag : analysis_tag);

where "value" is the feature value and "aid" is the list of the analysis ID we want to apply for this particular feature. Note that each feature set may contain more than one feature type, hence multiple calls to the AddFeature() function may be required. Each call to AddFeature() function will add the feature values to the flow's data containers, and the metric calculation may be triggered, depending on whether is calculation is due given the configured step size. The feature passing is finished by calling the function

FeatureAnalysis::CalculateMetric();

If the metric calculation is triggered, CalculateMetric() will raise an event with the list of metric values calculated. Note that each call to AddFeature() will result in adding a feature value into the data container even if it is a duplicate of the same feature value (Figure 9(a)). This is different from the case where we have the same feature value because we get the same value multiple times (Figure 9(b)). For example, suppose we call AddFeature() twice for the same value v, once for analysis ID a and once for analysis ID b, and both analysis use the same data container d, then there will be a duplicate insertion of value v to d.

The rest of the methods (see Figure 10) are for administrative purposes. RemoveConnection() removes a flow object from the flow dictionary whenever Bro removes a flow or connection from the main flow table. We can use the event connection_state_remove Bro provides to give us information about the flow or connection that is removed. The methods Extract() and ExtractVector() extract a feature (and a feature vector) from the event that is raised by the Analysis plugin. For Extract(), we have to specify the Feature ID and the Analysis ID,

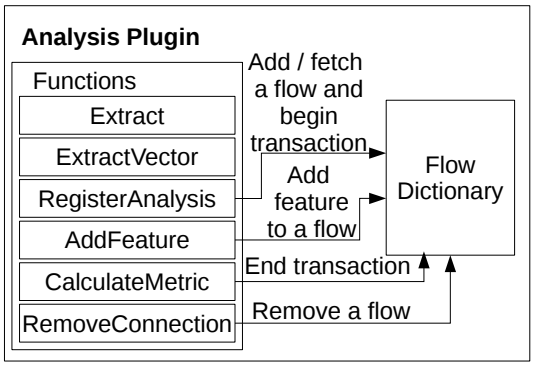

Figure 10: Internals of Analysis Plugin

whereas ExtractVector() returns all the feature values for all Feature and Analysis IDs. For example, if we have a metric result of $r = (\{3, a, x\}, \{5, b, y\}, \{7, c, z\})$, where a, b, c are the Feature IDs and x, y, z are the Analysis IDs, then calling Extract(r, z, c) will give us *7,* while calling ExtractVector(r) will give us $(3, 5, 7)$.

2.2.2 C4.5 Decision Tree Classifier

Another option to interpret the vector of metric values is by passing the vector to the C4.5 decision tree classifier. The tree consists of nodes that point to other nodes or a class (leaf), and each node is a comparison against a particular value in the input vector. Since for each input vector the tree traverse its nodes according to the corresponding feature values in the input vector, the computational overhead of the classifier is small. The implementation of the tree classifier is based on the implementation of the classifier in DIFFUSE [4], adapted to the C++ language. Currently, the supported comparisons are

binomial: whether a value compared against is equal or not to the value contained in the node; and

rational: whether the value compared against is less equal ($\leq$) or greater than ($>$) the value contained in the node.

Classifier models for BroCCaDe can be created with a modified version of the Waikato Environment for Knowledge Analysis (WEKA) [6]. By default WEKA saves classification models produced during training as Java serialised objects. This format is relatively complicated and no reliable free open-source C/C++ parsers exist. We have extended WEKA with a command line switch (`-y`) which allows saving WEKA models in an ASCII format that is easily readable for BroCCaDe.[1]

The format of the model file is as follows. A model file first lists the class names and feature/attribute names. The lines following these lists represent nodes of the tree and the associated tests. For each line the first parameter specifies the name of the node (`n_X`) while the second parameter specifies the name of the feature/attribute (`a_X`), where `X` corresponds to the numeric index of a feature in the feature list or the number of the node in the tree (starting with zero). The next parameter specifies whether the feature is nominal or real. The next parameter specifies the resulting class (`c_X`) if the feature is undefined (missing), where `X` corresponds to the numeric index of a class in the class list (starting with zero). Then the feature test is specified. There are three different cases:

- Nominal features with non-binary splits: a list of pairs of values and class/node names. Each value specifies a feature value and is followed by either a class name or node name.
- Nominal features with binary splits: a value followed by the class name or node name for equal feature values and the class name or node name for non-equal feature values.

[1] This WEKA extension was originally developed as part of DIFFUSE [4] and has been updated to a much more recent version of WEKA for BroCCaDe.

```
classes Iris-setosa Iris-versicolor Iris-virginica
attributes sepallength sepalwidth petallength petalwidth
n_0 a_3 r c_0 0.6 c_0 n_1
n_1 a_3 r c_1 1.7 n_2 c_2
n_2 a_2 r c_1 4.9 c_1 n_3
n_3 a_3 r c_2 1.5 c_2 c_1
```

Figure 11: Example C4.5 model

- Real features: a real split value followed by the class or node name for lesser equal feature values and the class or node name for greater feature values.

Figure 11 shows an example of a C4.5 classifier model generated for WEKA's Iris dataset [6].

3 Implementation

In this section we discuss how we implemented the feature extraction, the covert channel patterns, and the analysis methods we discussed in technical report 20171108A [1]. We also note the features of interest that need to be extracted and analysed for each covert channel.

3.1 Covert Channels

Recall that we are interested in five patterns: size modulation, value modulation, reserved / unused, inter-arrival time, and rate patterns.

Size Modulation Pattern: For this particular pattern, we selected two of the existing techniques (as described in technical report 20171108A [9]) and implemented a custom tool to create the covert channels in a trace file:

NTNCC: We implemented a Python script which takes as input single flows and then simulates NT-NCC's behaviour. During the initialisation, the script will read n packets and check if there are at least 2^wnumber of distinct packet lengths amongst the n packets, where w is the number of covert bits per packet. If there are not enough distinct packets lengths, the tool will read another n packets from the trace and check again. The program will do this until either there are enough distinct packet lengths or there are no more packets to be read. In the case where there are no more packets to be read, the program will insert packets with different packet lengths so that there are 2^wnumber of distinct packet lengths. The execution will then proceed as described in technical report [9] Section 5.1 except that we do not send the *Reference* packets. This is because we are only interested in the covert channel itself, not the setup of the covert channel.

Simple: For the simple packet length covert channel, the Python script takes as input the 5-tuple ID and the number of packets to send. The program will then send packets with random payload of length *n,* where n will differ according to the covert bit. If the covert bit is a logical *0,* the tool sends a packet with length *L,* otherwise it sends a packet with length *H.* Both H and L can be any distinct numbers, but in our experiments we used *20* and *30* by default.

Detecting both of the covert channels will require the capability to inspect the value of packet length. In the simple packet length scenario, the covert channel is easy to detect as it the flow will only have two modes of packet lengths, therefore the multi-modality or entropy metrics should work well. For NTNCC, autocorrelation is a good analysis metric because the covert channel destroys the autocorrelation. Multi-modality also works well especially in the case where there are not too many distinct number of packet lengths in each bin.

Value Modulation Pattern: We used CCHEF [3] to create TTL covert channels in a trace file. CCHEF implements the encoding specified in Section 5.2 of technical report 20171108A [9]. Detecting this TTL covert channel will require our infrastructure to check the TTL value of each inspected packet which fortunately is already provided by Bro. Similarly to the simple packet length channel, the TTL covert channel is relatively easy to detect as it the flow will have two modes for TTL, therefore the multi-modality or entropy metrics should work well. Note that legitimate flows with two modes may exist, but we expect them to be rare.

Reserved / Unused Pattern: We did not implement Ping Tunnel because there is already an implementation available online [5]. The distinguishing feature of Ping Tunnel is its magic number at the start of the ICMP payload. Therefore, in order to detect ping tunnel communication, we need to extract the first four bytes of ICMP payload of echo request and echo reply (See Section2.1.1). If the first four bytes match the magic number, then we can classify the traffic as Ping Tunnel.

Inter-arrival Time Pattern: For both of the covert channel techniques we have chosen as the representative for this pattern, we use CCHEF to create the covert channels.

modulo: CCHEF sees the inter-packet gap of the current packet and the last packet, and then add necessary delay to encode the covert bit as per the encoding rule described in technical report 20171108A [9] and Jitterbug [11].

simple: CCHEF sets the inter-packet gap according to the value of the covert bit to send, e.g., if the symbol to encode a logical *1* is *75ms* and the inter-packet gap of the current packet and the last packet is *30ms*, then CCHEF will delay the sending of the current packet by *45ms*. A caveat of CCHEF's approach to encode into existing overt traffic is that it needs to build up a queue of packets to send in order to encode the covert channel properly. If the queue runs empty packets CCHEF may not be able to produce the required inter-packet gaps, which means the covert channel would not be properly encoded. To solve this issue we selected flows where the average inter-packet time is smaller than the average inter-packet gap of the covert channel and so there is always a queue.

For the simple inter-packet gap channel, the inter-arrival times only have two modes. Hence, similar to the simple packet length covert channel, multi-modality analysis and entropy analysis should be able to effectively detect the existence of the covert channel. According to [8], entropy analysis is also a good approach to deal with the modulo timing channel (Jitterbug).

Rate Pattern: It is difficult to detect this channel by measuring the rate if the detector is unaware of the size of the time intervals in which the covert bits are sent, which is likely in the real-world as the covert sender and receiver can choose an arbitrary time interval. So rather than measuring the rate, we also measure inter-packet times to detect this channel. Cabuk et al. [7] proposed to use regularity analysis to detect this type of channel.

3.2 Analysis Metrics

Generally, the problem in implementing the analysis methods is in how we store the data itself (See Section 10) and not in how we implement the calculation, e.g., the calculation of Entropy and Multi-Modality analysis involves iterating over all of the histogram's buckets and apply the corresponding formula.

KS: To calculate the metric for the Kolmogorov-Smirnov (KS) metric, we need to load the normal data used as reference. We use the KS calculation from Numerical Recipes in C [10] adapted to C++. We made two modifications to the original Numerical Recipes code: (1) the data containers are C++ vectors instead of C arrays and (2) we assume the input data sets are already sorted a priori. We sort the data before the metric calculation because we do not want to sort the vectors of normal data every time we calculate the metric, i.e. the vectors of normal data are sorted after they are loaded during the initialisation.

CCE: We follow the implementation of Gianvecchio and Wang by using equiprobable bins and we store the patterns as a tree with the depth equal to the pattern length. For bin training, we first gather enough data and then sort and partition the data into bins with (roughly) equal number of data points. The boundaries of the bins become the intervals for the equiprobable bins used in calculating CCE, except the smallest and largest boundary becomes the smallest possible number and largest possible number respectively. More concretely, suppose that we have N data points and M number of bins, then each of the bins will have $floor(\frac{N}{M})$ number of data points. If the sorted data is $v_1, \cdots, v_M, v_{M+1}, \cdots, v_{2\cdot M}, v_{2\cdot M+1}, \cdots, v_N$, then the bin allocations will be $\{v_1, \cdots, v_M\}\{v_{M+1}, \cdots, v_{2\cdot M}\}\cdots\{v_{N-M+1}, \cdots, v_N\}$, and the intervals will be $DBL_{MIN} \leq i_1 < (\frac{v_M+v_{M+1}}{2})$, $(\frac{v_M+v_{M+1}}{2}) \leq i_2 < (\frac{v_{2\cdot M}+v_{2\cdot M+1}}{2})$, $\cdots$, $(\frac{v_{N-M}+v_{N-M+1}}{2}) \leq i_M < DBL_{MAX}$, where $v_j, 1 \leq j < N$ is the data point at index j, DBL_{MIN} and DBL_{MAX} are the smallest and the largest possible number respectively, and $i_{k,} 1 \leq k < M$ is the intervals for bin k.

We modified the implementation so that we do not have to traverse the tree every time we calculate the metric. As opposed to storing it in a tree data structure, we store it in an array of vectors where each vector in an array that corresponds to the depth of the tree. This way, the metric calculation only need to iterate over the vector as opposed to traversing the pattern tree. However, each element in the vector still acts like the node in the pattern tree, i.e., it has the count of the current pattern and the reference to the children nodes in subsequent array. Currently, we only take input that is long enough, i.e. it has at least L data points. We put incoming data into the corresponding bin, and remember the last L bin numbers which we treat as the input pattern for CCE.

Autocorrelation: Calculating the autocorrelation involves iterating over the raw data to calculate the mean, variance, and the autocorrelation value based on a lag. We use the autocorrelation calculation code from [2].

Regularity: Using the list of standard deviations of the windows, we calculate the standard deviation of these values. We use the rapid calculation technique for standard deviation as described in Section 2.1.2.

4 Example Scripts

Now we present some examples that demonstrate how to use BroCCaDe, ranging from a simple pure Bro script to a complex example which involves a protocol parser / feature extraction plugin and the decision tree classifier. For the script that is using the decision tree classifier, we assume that we already have a C4.5 tree model. In reality, we can use the training plugin to produce this model for us.

4.1 Basic Script

The simplest example is for analysing packet length where we interpret the metric result directly in the Bro script as opposed to passing it to the decision tree. In this scenario, suppose we are only interested in the KS, Entropy, CCE, and regularity analysis, and let the thresholds that indicate a covert channel for these metrics be 1.5, 1, and 0.4 respectively (note that these values are arbitrary). If all metrics satisfy the threshold conditions, then the script detects a covert channel and prints "possible covert channel".

```
global aid : vector of FeatureAnalysis::Analysis_ID;

event bro_init()
{
  aid[0] = FeatureAnalysis::CCE_ANALYSIS;
  aid[1] = FeatureAnalysis::REGULARITY_ANALYSIS;
  aid[2] = FeatureAnalysis::ENTROPY_ANALYSIS;

  FeatureAnalysis::ConfigureInternalType();

  local aid_CCE : vector of FeatureAnalysis::Analysis_ID;
  aid_CCE[0] = FeatureAnalysis::CCE_ANALYSIS;
  local aid_EN : vector of FeatureAnalysis::Analysis_ID;
```

```
14    aid_EN[0] = FeatureAnalysis::ENTROPY_ANALYSIS;
15    FeatureAnalysis::LoadInterval(PACKET_LENGTH, aid_CCE, "INTERVAL_PACKET_LENGTH");
16    FeatureAnalysis::SetBinNull(PACKET_LENGTH, aid_EN_MM, 65536);
17  }
18
19  event PacketLength_feature_event(UID:string, id:conn_id,
20    direction:FeatureAnalysis::Direction, value: double)
21  {
22    FeatureAnalysis::RegisterAnalysis(UID, FeatureAnalysis::PACKET_LENGTH_SET, id, direction);
23    FeatureAnalysis::AddFeature(value, aid, PACKET_LENGTH);
24    FeatureAnalysis::CalculateMetric();
25  }
26
27  event new_packet (c: connection, p: pkt_hdr)
28  {
29    if ( p ?$ ip ) {
30      event PacketLength_feature_event(c$uid, c$id,
31        FeatureAnalysis::GetDirection(c$id$orig_h, p$ip$src), p$ip$len);
32    }
33  }
34
35  event FeatureAnalysis::metric_event(id : FeatureAnalysis::set_ID,
36    direction:FeatureAnalysis::Direction, v : result_vector, conn_ID: conn_id)
37  {
38    local CCE_value = Extract(v, FeatureAnalysis::CCE_ANALYSIS, PACKET_LENGTH);
39    local EN_value = Extract(v, FeatureAnalysis::ENTROPY_ANALYSIS, PACKET_LENGTH);
40    local Regularity_value = Extract(v, FeatureAnalysis::REGULARITY_ANALYSIS, PACKET_LENGTH);
41    if ((EN_value > 1.5) && (CCE_value < 1) && (Regularity_value > 0.4))
42        print("possible covert channel");
43  }
```

We now explain the above script in detail:

- **Analysis set (line 1-7):** One of the most important things when writing a script for analysis is determining the type of analysis that we want to apply to the feature values. This is done via constructing a vector of analysis IDs (in the example it is the global variable *aid*), and populating the vector with the analysis IDs (line 5-7) during Bro's initialisation phase. If different features would use the same set of metrics, then *aid* can be reused. Otherwise we may need to define many different vectors of analysis IDs. Note that it is better to define the vector of analysis IDs globally rather than locally right before calculating the metrics, e.g., before line 22, because there is a significant overhead associated with Bro allocating and deallocating objects many times.

- **Various parameters for the analysis (line 11-16):** Currently we have several functions to modify the parameters for various analysis methods. In this example we can define the binning interval for packet length (used by CCE analysis) and configure the histogram data structure (used by entropy analysis) to use the feature values as bin number. Aside from these two functions, we may also add a vector of normal data for KS analysis, and set the maximum lag for autocorrelation analysis.

- **Analysis plugin (line 19-25):** After we determined the type of analysis we want, we notify the analysis plugin that we want to begin the process by calling the function *RegisterAnalysis()* with the parameters connection UID, feature set identifier, and the 5-tuple flow identifier. We then add each value and the analysis associated with it with *AddFeature()*, and notify the analysis plugin that we have finished with this particular feature (*CalculateMetric()*). The analysis plugin will then raise a metric event containing the analysis result when the metric calculation is triggered.

- **Feature extraction (line 27-33):** Firstly, we have to extract a feature, which in this case is packet length. Fortunately, this value is already provided in the IPv4 header. We currently only consider IPv4 and ignore IPv6, which in the script it is done via the instruction "if (p ?$ ip)". In this scenario, we do the analysis for all incoming packets, hence we use the *new_packet* event handler which is already provided in Bro. The *new_packet* event handler will raise a feature event to be handled by the analysis plugin. This feature

event is parameterised by the connection UID, 5-tuple flow identifier and the feature value, i.e. the packet length.

- **Interpreting the metric (line 35-43):** Here the metric values calculated by the analysis plugin are compared against the threshold values. BroCCaDe provides a way to access metric values in the vector delivered by the event with the function Extract() parametrised by the Feature ID and Analysis ID. The values are then compared against static (user-defined) threshold values, and depending on the policy we might send an e-mail, log an alert, etc. In this simple example we simply print a message on the screen.

Note that Bro type is defined as an object in C++. We found that looking up a type, e.g., feature_vector which we define as a vector of double, incurs a significant computational overhead. Therefore in line 9 we tell the plugin to cache the lookup result for Bro types internally.

4.2 Decision Tree Classifier

This example is a bit more complex in that we also use the decision tree classifier. To do this, we have to initialise the decision tree by loading the model (line 1-4), and then pass the metric values to the decision tree to classify. The feature generation and the analysis are the same as in the simple example, we just change the event handler which deals with the result of the analysis. As opposed to comparing the metric values against thresholds, we now pass the vector of metric values to the decision tree for classification.

```
1  global aid : vector of FeatureAnalysis::Analysis_ID;
2
3  event bro_init()
4  {
5    aid[0] = FeatureAnalysis::REGULARITY_ANALYSIS;
6    aid[1] = FeatureAnalysis::AUTOCORRELATION_ANALYSIS;
7
8    FeatureAnalysis::ConfigureInternalType();
9
10   DecisionTree::LoadModel(FeatureAnalysis::PACKET_LENGTH_SET, "TreeModel-PacketLength");
11 }
12
13 event PacketLength_feature_event(UID:string, id:conn_id,
14   direction:FeatureAnalysis::Direction, value: double)
15 {
16   FeatureAnalysis::RegisterAnalysis(UID, FeatureAnalysis::PACKET_LENGTH_SET, id, direction);
17   FeatureAnalysis::AddFeature(value, aid, PACKET_LENGTH);
18   FeatureAnalysis::CalculateMetric();
19 }
20
21 event new_packet (c: connection, p: pkt_hdr)
22 {
23   if ( p ?$ ip ) {
24     event PacketLength_feature_event(c$uid, c$id,
25       FeatureAnalysis::GetDirection(c$id$orig_h, p$ip$src), p$ip$len);
26   }
27 }
28
29 event FeatureAnalysis::metric_event(id : FeatureAnalysis::set_ID, v : result_vector)
30 {
31   if (DecisionTree::Classify(id, FeatureAnalysis::Extract_vector(v)) == 0)
32     print("possible covert channel");
33 }
```

The decision tree is initialised by assigning a set ID with a model. There are several ways to pass the analysis result into the decision tree:

- Pass a raw vector of double values, which is shown in the example script above. In this case, **the order of the metric values in the vector has to match the order of feature in the tree model**, e.g., *[1, 2, 3]*; or

- Pass a vector of double values and a corresponding vector of strings indicating the feature names. The decision tree will then use the feature values based on the feature names specified in the model loaded, e.g. *values=[1, 2, 3], names=["analysisA", "analysisB", "analysisC"].* In this scenario, the number of features and the length of the vector have to match. The vector of feature names also has to contain at least the feature names specified in the model. If at least one of the restriction is violated, the plugin will not proceed with the classification process and return an error, but Bro does not terminate; or

- Pass a record of metric values where the field name is the feature name, e.g., *{analysisA=1, analysisB=2, analysisC=3}* corresponds to analysis result for *analysisA, analysisB,* and *analysisC*. When using this type of input for the decision tree classifier, the record has to contain at least the feature names specified in the model, otherwise Bro will exit with an error of not finding a field in the record.

Examples of the last two methods of passing the arguments to the decision tree are shown below.

```
event FeatureAnalysis::metric_event(id : FeatureAnalysis::set_ID, v : result_vector)
{
  local vec : FeatureAnalysis::feature_vector = FeatureAnalysis::Extract_vector(v);
  local name_vec : vector of string;
  name_vec[|name_vec|] = "CCE_value";
  name_vec[|name_vec|] = "Regularity_value";
  DecisionTree::Classify_with_strings(
    FeatureAnalysis::PACKET_LENGTH_SET, vec, name_vec);
}

type decision_tree_args : record { CCE_value : double; Regularity_value : double; };

event FeatureAnalysis::metric_event(id : FeatureAnalysis::set_ID, v : result_vector)
{
  local CCE_value = Extract(v, FeatureAnalysis::CCE_ANALYSIS, PACKET_LENGTH);
  local Regularity_value = Extract(
    v, FeatureAnalysis::Regularity_ANALYSIS, PACKET_LENGTH);
  local r : FeatureAnalysis::decision_tree_args =
    [$CCE_value=CCE_value, $Regularity_value=Regularity_value];
  DecisionTree::Classify_record(FeatureAnalysis::PACKET_LENGTH_SET, r);
}
```

4.3 All Features

For simple features like the packet length, it is sufficient to modify the Bro script of the analysis plugin. For more complicated features like the TCP URG flag and URG pointer, which Bro does not provide, we have to write additional plugins to extract these features. See Section 2.1.1 for more information. Putting everything together, here is a sample script that does analysis on four different feature sets. All of the analysis pass metric values directly into the decision tree.

```
@load /opt/bro/lib/bro/plugins/FeatureExtraction/lib/bif
@load /opt/bro/lib/bro/plugins/FeatureExtraction_URG/lib/bif
@load /opt/bro/lib/bro/plugins/FeatureExtraction_PTunnel/lib/bif

global aid : vector of FeatureAnalysis::Analysis_ID;
global aid_null : vector of FeatureAnalysis::Analysis_ID;

event connection_state_remove (c: connection)
{
  FeatureAnalysis::RemoveConn(c$uid);
}

event PacketLength_feature_event(UID:string, id:conn_id,
  direction:FeatureAnalysis::Direction, value: double)
{
  FeatureAnalysis::RegisterAnalysis(UID, FeatureAnalysis::PACKET_LENGTH_SET, id, direction);
  FeatureAnalysis::AddFeature(value, aid, PACKET_LENGTH);
```

```
  FeatureAnalysis::CalculateMetric();
}

event IAT::feature_event(UID:string, id:conn_id,
  direction:FeatureAnalysis::Direction, value: double)
{
  FeatureAnalysis::RegisterAnalysis(UID, FeatureAnalysis::IAT_SET, id, direction);
  FeatureAnalysis::AddFeature(value, aid, INTERARRIVAL_TIME);
  FeatureAnalysis::CalculateMetric();
}

event TTL_feature_event(UID:string, id:conn_id,
  direction:FeatureAnalysis::Direction, value: double)
{
  FeatureAnalysis::RegisterAnalysis(UID, FeatureAnalysis::TTL_SET, id, direction);
  FeatureAnalysis::AddFeature(value, aid, TTL);
  FeatureAnalysis::CalculateMetric();
}

event icmp_echo_reply(c: connection, icmp: icmp_conn, id:
  count, seq: count, payload: string)
{ FeatureExtraction::ExtractHeaderFeature(c$uid, c$id,
    FeatureAnalysis::BACKWARD, payload, 0, 4); }

event icmp_echo_request(c: connection, icmp: icmp_conn,
  id: count, seq: count, payload: string)
{ FeatureExtraction::ExtractHeaderFeature(c$uid, c$id,
    FeatureAnalysis::FORWARD, payload, 0, 4); }

event new_packet (c: connection, p: pkt_hdr)
{
  if ( p ?$ ip ) {
    IAT::ExtractFeature(c$uid, c$id,
      FeatureAnalysis::GetDirection(c$id$orig_h, p$ip$src), c$duration);
    event TTL_feature_event(c$uid, c$id,
      FeatureAnalysis::GetDirection(c$id$orig_h, p$ip$src), p$ip$ttl);
    event PacketLength_feature_event(c$uid, c$id,
      FeatureAnalysis::GetDirection(c$id$orig_h, p$ip$src), p$ip$len);
  }
}

event bro_init() {
  DecisionTree::LoadModel(FeatureAnalysis::PTUNNEL_SET, "TreeModel-PingTunnel");
  DecisionTree::LoadModel(FeatureAnalysis::URGENT_SET, "TreeModel-URG");
  DecisionTree::LoadModel(FeatureAnalysis::IAT_SET, "TreeModel-IAT");
  DecisionTree::LoadModel(FeatureAnalysis::PACKET_LENGTH_SET, "TreeModel-PacketLength");

  FeatureAnalysis::ConfigureInternalType();
  FeatureAnalysis::AddNormalData(INTERARRIVAL_TIME, "KS_NORMAL_IAT");
  FeatureAnalysis::AddNormalData(PACKET_LENGTH, "KS_NORMAL_PACKET_LENGTH");
  FeatureAnalysis::AddNormalData(TTL, "KS_NORMAL_TTL");

  aid[0] = FeatureAnalysis::KS_ANALYSIS;
  aid[1] = FeatureAnalysis::ENTROPY_ANALYSIS;
  aid[2] = FeatureAnalysis::CCE_ANALYSIS;
  aid[3] = FeatureAnalysis::MULTIMODAL_ANALYSIS;
  aid[4] = FeatureAnalysis::AUTOCORRELATION_ANALYSIS;
  aid[5] = FeatureAnalysis::REGULARITY_ANALYSIS;
  aid_null[0] = FeatureAnalysis::NULL_ANALYSIS;

  local aid_CCE : vector of FeatureAnalysis::Analysis_ID;
  aid_CCE[0] = FeatureAnalysis::CCE_ANALYSIS;
  local aid_EN_MM : vector of FeatureAnalysis::Analysis_ID;
  aid_EN_MM[0] = FeatureAnalysis::ENTROPY_ANALYSIS;
  aid_EN_MM[1] = FeatureAnalysis::MULTIMODAL_ANALYSIS;
```

```
83    FeatureAnalysis::LoadInterval(TTL, aid_CCE, "INTERVAL_TTL");
84    FeatureAnalysis::SetBinNull(TTL, aid_EN_MM, 256);
85    FeatureAnalysis::LoadInterval(INTERARRIVAL_TIME, aid_CCE, "INTERVAL_IAT_COARSE");
86    FeatureAnalysis::LoadInterval(INTERARRIVAL_TIME, aid_EN_MM, "INTERVAL_IAT_FINE");
87    FeatureAnalysis::LoadInterval(PACKET_LENGTH, aid_CCE, "INTERVAL_PACKET_LENGTH");
88    FeatureAnalysis::SetBinNull(PACKET_LENGTH, aid_EN_MM, 65536);
89  }
90
91  event FeatureExtraction::URG_feature_event(UID : string,
92    id : conn_id, direction:FeatureAnalysis::Direction,
93    URG_flag : count, URG_ptr : count)
94  {
95    FeatureAnalysis::RegisterAnalysis(UID, FeatureAnalysis::URGENT_SET, id, direction);
96    FeatureAnalysis::AddFeature(URG_flag, aid_null, URG_FLAG);
97    FeatureAnalysis::AddFeature(URG_ptr, aid_null, URG_POINTER);
98    FeatureAnalysis::CalculateMetric();
99  }
100
101 event FeatureExtraction::PTunnel_feature_event(UID : string,
102   id : conn_id, direction:FeatureAnalysis::Direction,
103   features : FeatureAnalysis::feature_vector)
104 {
105   FeatureAnalysis::RegisterAnalysis(UID, FeatureAnalysis::PTUNNEL_SET, id);
106   for (idx in features) FeatureAnalysis::AddFeature(features[idx],
107     aid_null, ICMP_PAYLOAD_4_BYTES);
108   FeatureAnalysis::CalculateMetric();
109 }
110
111 event FeatureAnalysis::metric_event(id : FeatureAnalysis::set_ID,
112   direction:FeatureAnalysis::Direction, v : result_vector, conn_ID: conn_id)
113 {
114   if (DecisionTree::Classify(id, FeatureAnalysis::Extract_vector(v)) == 0)
115   {
116     print("Covert Channel detected");
117     print conn_ID;
118   }
119 }
```

- **TTL:** The TTL is extracted from the packet header (lines 52–53) and then the value is passed to the analysis plugin (lines 29–35).
- **Packet Length:** The feature itself is extracted (lines 54–55) and then the value is passed to the analysis plugin (lines 13–20).
- **Inter Arrival Time:** The packet inter-arrival time values are extracted by the feature extraction plugin, which raises the "IAT::feature_event" (lines 21–27).
- **URG:** The feature value is extracted by the protocol parser plugin, and the value is passed through the Null analysis to produce a metric value (lines 91–99).
- **Ping Tunnel:** The feature value extracted by the plugin is the first four bytes of the ICMP payload. The payload itself is already provided by Bro. This feature value is then passed through the Null analysis (lines 101–109).

Aside from the feature specific extraction and analysis, one needs to make sure that each connection is cleaned up (lines 8–11) as otherwise connections will never be removed from the dictionary and will continue to use memory. It also needs to be ensured that all necessary data types defined in the additional plugins are present (lines 1–3).

5 Conclusion

We described the design of the Bro Covert Channel Detection (BroCCaDe) framework which uses Bro's scripting and plugin mechanism. We designed a framework that consists of a custom Bro script and three separate plugin classes: Protocol Parser Plugin, Feature Extraction Plugin, and Analysis Plugin. The script coordinates the functions of the different plugins and effectively ties them together while the plugins implement complex and performance-critical functionality as native code, such as parsing packets, extracting features, computing analysis metrics and ML classification. We also discussed important implementation details of BroCCaDe and based on several use cases with actual code examples demonstrated how BroCCaDe can be used.

Acknowledgements

This work was supported by a grant from the Comcast Innovation Fund.

References

[1] Attribute-Relation File Format (ARFF). https://www.cs.waikato.ac.nz/ml/weka/arff.html.

[2] auto.c. `www.csee.usf.edu/~kchriste/tools/autoc.c`, Accessed: 12 June 2017.

[3] Covert Channels Evaluation Framework (CCHEF). https://sourceforge.net/projects/cchef/.

[4] DIstributed Firewall and Flow-shaper Using Statistical Evidence (DIFFUSE). `http://caia.swin.edu.au/urp/diffuse/`, Accessed: 12 June 2017.

[5] Ping Tunnel - For those times when everything else is blocked. `http://www.cs.uit.no/~daniels/PingTunnel/`, Accessed: 6 April 2017.

[6] Weka 3: Data Mining Software in Java. `http://www.cs.waikato.ac.nz/ml/weka/`, Accessed: 12 June 2017.

[7] Serdar Cabuk, Carla E Brodley, and Clay Shields. IP covert timing channels: design and detection. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 178–187. ACM, 2004.

[8] Steven Gianvecchio and Haining Wang. An entropy-based approach to detecting covert timing channels. *IEEE Transactions on Dependable and Secure Computing*, 8(6):785–797, 2011.

[9] Hendra Gunadi and Sebastian Zander. Bro Covert Channel Detection (BroCCaDe) Framework: Scope and Background. Technical Report 20171117A, Bro Covert Channel Detection (BroCCaDe) Framework: Scope and Background, 2017.

[10] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes in C*, volume 2. Cambridge Univ Press, 1982.

[11] Gaurav Shah, Andres Molina, Matt Blaze, et al. Keyboards and covert channels. In *USENIX Security Symposium*, volume 15, 2006.