Performance Evaluation of the Bro Covert Channel Detection (BroCCaDe) Framework

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

27 April 2018

Abstract

The Bro Covert Channel Detection (BroCCaDe) framework is a Bro extension for detecting covert channels in network protocols. This report describes a number of experiments we have carried out with BroCCaDe to measure its performance in terms of classification accuracy and performance (CPU and memory overhead). We tested BroCCaDe with a number of different traffic types and covert channels embedded in the IP TTL field, packet length, inter-packet time and packet rate. Our results show that BroCCaDe can identify these channels with a high accuracy (true positive rate of 98% and false negative rate of generally less than 1%). Our performance analysis reveals that BroCCaDe requires a small to moderate additional amount of RAM and CPU time. The overhead in terms of CPU time is generally less than 50% and the overhead in terms of memory is generally a few Megabytes, except for the entropy rate metric. Notably, a substantial proportion of the memory overhead is due to storing the feature values. Most of the CPU overhead is a result of the metric computation and feature extraction while the classification of flows requires very little CPU time. Our analysis also reveals which detection metrics are most useful for the detection of particular covert channels.

1 Introduction

The Bro Covert Channel Detection (BroCCaDe) framework is a Bro extension for detecting covert channels in network protocols. BroCCaDe's implementation is based on the design described in technical report [1] and here we only briefly describe it. BroCCaDe performs analysis based on unidirectional packet flows (packets with same 5-tuple), which can be part of bidirectional flows though. For the detection of covert channels BroCCaDe supports the following metrics which are described in more detail in [1]: regularity test, multi-modality analysis, entropy analysis, entropy rate (CCE) analysis and autocorrelation analysis. The metrics are computed for a certain number of packets of a flow (called a window) and can be either directly compared to predetermined thresholds or can be feed to a trained ML classifier. Currently, for the ML classification BroCCaDe implements a C4.5 decision tree as classifier and tree models are trained with WEKA [2] and loaded into BroCCaDe.

This report describes a number of experiments we have carried out with BroCCaDe to measure its performance in terms of classification accuracy and CPU and memory overhead. We tested BroCCaDe with covert channels embedded in the IP TTL field, packet length (both NTNCC [3] and a simple direct encoding), inter-packet time (both direct simple encoding and Sha's modulo encoding [4]) and packet rate. All experiments were carried out with trace files on a commodity PC. We chose the trace-based approach, so experiments are repeatable and especially with regards to timing there is no random noise introduced during the experiments. To create the traces, we selected a number of publicly available traffic traces and used part of these traces to represent clean traffic while in the other part we encoded the covert channels using CCHEF [5] and custom tools.

We mainly used UDP traffic flows as packet length and inter-packet times cannot be as easily and freely manipulated with TCP (especially not with TCP in trace files). We note that since our detection relies only on the primary features that are manipulated by the covert channels (e.g. IP TTL values, packet lengths and inter-arrival times) there would be not much difference in using UDP or TCP.

Table 1: Specification of computer on which the performance tests were run

Memory	16 GiB
Processor	Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	8192K

From the experiments we calculated the following metrics: To evaluate the accuracy we computed the true positive and false positive rates and to evaluate the performance we measured Bro's run time (as Bro constantly uses 100% of the CPU with trace files) and memory consumption.

Our results show that BroCCaDe can identify the covert channels we used with a high accuracy (true positive rate of 98% and false negative rate of generally less than 1%). Our performance analysis reveals that BroCCaDe requires a small to moderate additional amount of RAM and CPU time. Compared to standard Bro, BroCCaDe increases the resource usage by only a few MB, except for the CCE metric where the increase is more substantial, and the overhead in terms of CPU time is significant but generally under 50%. Notably, a big proportion of the memory overhead is due to storing the feature values. Most of the CPU overhead is a result of the metric computation and feature extraction while the classification of flows requires very little CPU time. Our analysis also reveals which detection metrics are most useful for the detection of particular covert channels.

Section 2 describes our experimental setup (computer specification and traces). Section 3 and Section 4 describe the results of experiments conducted to determine the detection accuracy of BroCCaDe. Section 5 describe the results of the performance experiments. Section 6 concludes the report.

2 Experimental Setup

In this section we describe the experimental setup in terms of the computer (Section 2.1) we used and the datasets we created and used (Section 2.2). While BroCCaDe can classify traffic directly based on the values of metrics, in practice the thresholds are not straight-forward to derive. Hence, in our experiments we focus on the detection using a supervised C4.5 ML classifier. For C4.5, as for any supervised ML classifier, we need to train a classifier model in the training phase that is then used to classify unknown instances in the testing phase. We describe the methodology used for training and testing in Section 2.3 and Section 2.4 respectively.

2.1 Computer Specification

Table 1 shows the computer specification on which we gather the data for the performance of BroCCaDe.

2.2 Data Set

In this section we describe the trace files used for the experiments. Firstly, a flow is defined as a number of packets with the same 5-tuple (source address, source port, destination address, destination port and protocol) as usual. In our study we always consider unidirectional flows, i.e. every bidirectional flow is split into two unidirectional flows into which covert channels are encoded separately and which are analysed separately. Secondly, we define some terminology related to trace files:

- Carrier Trace is a trace file containing flows which will be used to embed covert flows.
- Clean Trace is a trace file which is free from covert channels.
- Covert Trace is a trace file containing flows which are embedded with a covert channel.

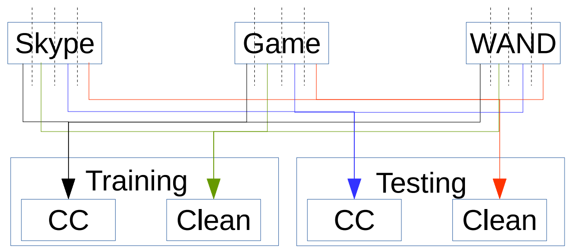


Figure 1: Data set creation

For a covert channel to transmit a meaningful amount of information, the carrier flow must have a minimum length. For this study we chose a minimum length of 500 packets for each unidirectional flow. This is sufficient to transfer at least a few hundred bits, which could be an ex-filtrated key or password. Note that during the initial processing with CCHEF packets are grouped into unidirectional flows by their 5-tuple and we use a flow timeout of 3 seconds, which means an idle time of 3 seconds ends the current flow and starts a new flow with the same 5-tuple.[1] In case the flow timeout splits one long flow into multiple short flows (with the same 5-tuple) we use all the short flows with the same 5-tuple for training if they are at least 500 packets long.

We took several public data sets, from which we selected all UDP flows that were at least 500 packets long:

- Game traffic [6]: These are traffic that contain game traffic captured in a LAN for different games: Quake 3, Quake 4, Enemy Territory, Half-Life Deathmatch, Half-Life Counter-Strike, Half-Life 2 Deathmatch, Half-Life 2 Counter-Strike. We used traces for all different player numbers from the aforementioned games.

- Skype traffic [7]: The trace was captured on an access link of Politecnico in Torino. Specifically, we use the UDP Skype Out calls trace.

- Traffic capture captured at New Zealand ISP [8]: We randomly selected some traces from ISPDSL II (20100106-150000-0.dsl.erf.gz, 20100106-153000-0.dsl.erf.gz, 20100107-163000-0.dsl.erf.gz, 20100109-023000-0.dsl.erf.gz, 20100109-083000-0.dsl.erf.gz, 20100109-143000-0.dsl.erf.gz, 20100109-150000-0.dsl.erf.gz, 20100109-153000-0.dsl.erf.gz, 20100110-163000-0.dsl.erf.gz).

- Traffic captured by the WIDE project [9]: we took a randomly selected traffic capture from one day which was captured at the transit link of WIDE to the upstream ISP (201709091400.pcap.gz).

We split these data sets into four roughly equal-sized traces as illustrated in Figure 1. Two traces are used for training the classifier while the other two traces are used for the testing. In each case (training or testing) the two traces are the clean trace and the carrier trace. The covert traces are generated from the carrier traces either by using CCHEF [5] (for TTL and inter-packet time covert channels) or custom Python scripts (for packet length and packet rate covert channels). In practice, the training data is obtained separately, one run for clean trace and one run for covert trace, and later merged (see Section 2.3). Traffic datasets for testing are created in a similar fashion. Table 2 shows the size of the different datasets used for training and testing.

2.3 Training Methodology

For our evaluation, we train a model for each of the covert channels. To train a model for a particular covert channel, we take as input two separate trace files, the carrier trace (which will be used to embed covert channels) and the clean trace (trace without covert channels in it). During the covert traces creation process, we take individual flows from the carrier trace and embed the covert channels; for TTL, inter-arrival time modulo and inter-arrival time simple covert channels, we use CCHEF [5] to embed the covert traces; for NTNCC, simple packet length and rate channels, we implemented Python scripts to generate covert traces.

[1]This flow timeout ensures that Bro sees the same flows as CCHEF, since Bro also uses a flow timeout of 3 seconds.

Table 2: Number of flows in the datasets used for training and testing

Type	Class	Trace	Flows
Training	Normal	Game	25
		Skype	319
		WAND	385
		MAWI	190
	CC	Game	18
		Skype	333
		WAND	393
		MAWI	177
Testing	Normal	Game	18
		Skype	346
		WAND	418
		MAWI	173
	CC	Game	18
		Skype	317
		WAND	418
		MAWI	178

We use the clean trace to train the auxiliary data for the Analysis Plugin. Currently, this data consists of equiprobable bins (bin intervals) and is computed as follows. We first compute the features for the training data. The equiprobable bin intervals are obtained by storing all the feature values and then dividing the data points according to the number of bins. The boundary values for the bins are then the minimum and the maximum feature values in each particular bin. To be more precise, the boundaries are the midpoint between adjacent buckets except for the smallest and largest feature values, which are set to be the minimum possible value and maximum possible value respectively to cater for feature values that we might not have seen yet during training. For TTL and packet length, we use the actual values as the bins, i.e. we have bin 0 to 255 for TTL and we have bin 0 to 65535 for packet length.

Subsequently, we run the covert trace and the clean trace through Bro equipped with BroCCaDe (Analysis Plugin, Training Plugin, and Feature Extraction Plugin). The Analysis Plugin, using the auxiliary data obtained as described above, will produce analysis result, which will be used as an input to the Training Plugin. The Training Plugin outputs the analysis result in the form of ARFF files for both clean and covert traces. These ARFF files are merged, using a tool (CombineArff, described later in this section), and then used as input to WEKA, which is then used to produce tree models for each individual covert channel. The resulting tree models can then be used as the models for the Classifier Plugin in BroCCaDe for classifying unknown traffic.

Figure 2 shows the training procedure starting from a training trace (both clean and carrier trace for covert channels).

There are several scripts and tools that are involved in training tree models for the decision tree classifier.

- **covert_trace.sh**: This script takes two trace files as input: a clean trace and a carrier trace. Then for each specified covert channel, the script will work on a carrier trace to produce a covert trace and finally a combined trace, i.e., the covert and clean traces combined. The covert traces will be used for training, while the combined trace will *only* be used for testing (see following section). Note that the input traces for training and testing must be different.

- **train_bin.sh**: This script takes as input the clean trace and executes Bro with the Training Plugin to compute the feature values and produce equiprobable bins for various bin sizes.

- **train_feature.sh**: This script takes as input several traces: one clean trace and several covert traces. The script will execute Bro with the Analysis Plugin and Training Plugin on the input traces. Analysis results from the Analysis Plugin are directly passed into the Training Plugin to produce ARFF files. If the input

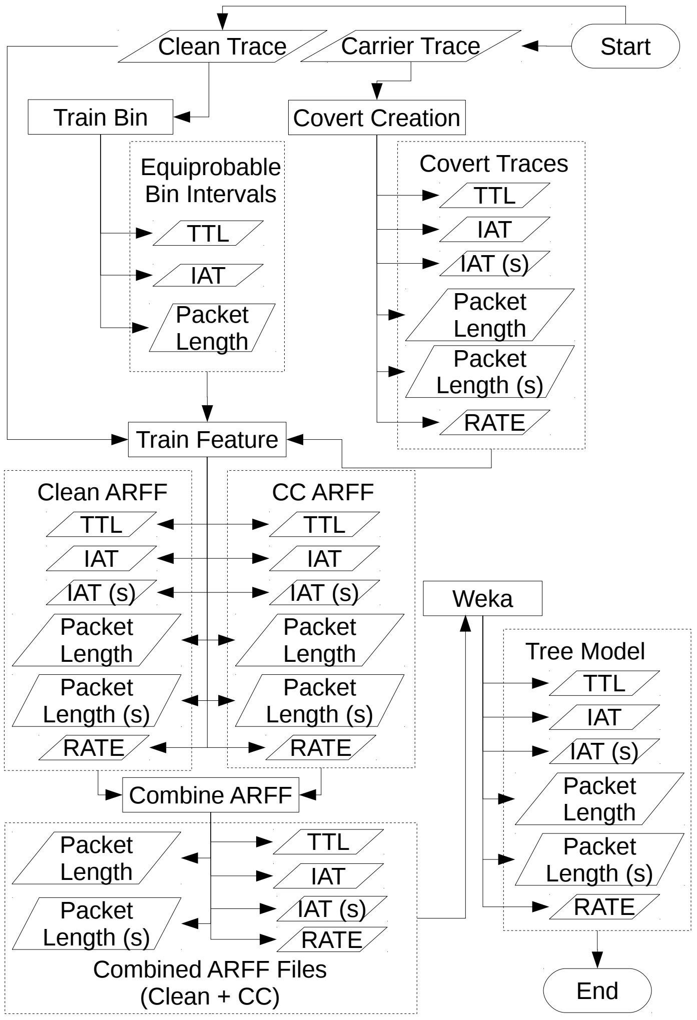


Figure 2: Training methodology

is a clean trace, then each data row in the output file will be labelled as type “Non-CC”. Otherwise, each row in the output file will be labelled as type “CC”. Finally, the script will merge all the ARFF files (using the tool CombineArff) and pass the merged ARFF file to WEKA which will produce a tree classifier model.

- **CombineArff**: This tool takes two input ARFF files and merges them into one ARFF file, combining the unmodified data rows for the two classes (CC and non-CC).
- **automate_training.sh**: This main script implements the whole training work flow as described in Figure 2. It first uses covert_trace.sh to produce covert channels, produce equiprobable bins using train_bin.sh, and then train the tree classifier using train_feature.sh.

2.4 Testing Methodology

For the testing phase, we need the tree model for the Classifier Plugin, which is obtained using the training procedure described in the previous section. We also need a clean trace and a covert trace that differ from the traces used for training. Both traces are feed it into covert_trace.sh and the resulting covert channels are combined with the clean trace to simulate the real-time situation where we do not know beforehand whether a particular flow contains a covert channel or not. Also, covert_trace.sh will produce a list of the flows (5-tuples) with covert channels which is used as ground truth for the accuracy computation.[2]

We then run Bro with BroCCaDe (Analysis Plugin, Classifier Plugin, and Feature Extraction Plugins). The Bro script is used to produce a per-flow history of the classification, i.e. when a classification is triggered on a particular unidirectional flow, the classification result or class label (CC or Non-CC) is appended to the flow. At the end of the execution, Bro will output the 5-tuple connection ID followed by the history of class labels (one for each window classified). Again, note that while many of the traffic flows from the trace data are bidirectional, we analyse both directions of a flow separately, i.e. we analyse undirectional flows.

Since Bro classifies a flow many times (once for every window), there are several ways of identifying whether a flow contains a covert channel or not:

- The whole flow is examined and the ratio of the per-window classifications of covert and non-covert is calculated. If the ratio is above a threshold r, then we classify the flow as containing covert channels. A clear benefit of this approach is that it can accurately find a trade-off between true positive and false positive rate, but it means we must potentially wait for a flow to finish before we can accurately decide whether a flow contains a covert channel or not. Furthermore, it is not necessarily the case that the covert sender will only send covert information; the covert sender may also send some random junk to obfuscate the existence of covert communication.
- A flow is classified as covert channel whenever the number of consecutive windows classified as covert channel exceeds a threshold t. This achieves a relatively quick classification for continuous covert channels, i.e. after t windows a flow is classified, but on the other hand the covert sender could choose not to maximise the channel throughput and try to stay under the threshold to avoid detection. However, the covert sender likely does not know the exact window size and t, making it hard to avoid detection while sending a meaningful amount of covert data at the same time.
- Whenever one window of a flow is classified as covert channel, the flow is labelled as covert channel. The obvious drawback of this approach is that the false positive rate is high. On the other hand, this approach is sensitive even to the slightest possibility that a flow contains a covert channel (and reduces the chance for false negatives).

Figures 3 and 4 show the testing procedure starting from the tree model, clean trace and covert trace. In the testing phase there are several scripts and tools that are involved:

- **covert_trace.sh**: This is the same script as the one used in training, but here we make use of its capability to produce a combined trace.
- **automate_testing.sh**: This script takes as input the tree model, auxiliary data for the Analysis Plugin, and a combined trace file. Then it executes Bro with the Analysis Plugin and necessary feature plugins. The Bro script produces the per flow classification history, which will be used as input to ground_truth.sh.
- **ground_truth.sh**: the script produces the ground truth for covert channels and clean flows. It starts by filtering the flows in the list of covert channels produced by covert_trace.sh based on the output of automate_testing.sh. The filtering process is done by iterating over the list of flows with covert channels and checking which flows are present in the Bro output. If a flow exists in the Bro output, then the flow is added to the ground truth of covert flows. For the ground truth of clean flows, we check which flows are not covert channels and are included in the Bro output. All flows that fulfil this criterion are added to the ground truth of clean flows.
- **history.sh**: This script will take the ground truth produced by ground_truth.sh (both lists of flows with covert channels and clean flows) and calculate the number of true positives, false positives, false negatives,

[2]Not all flows containing covert channels may appear in the Bro output due to various reason, e.g., some flows are too short and did not trigger a classification. Therefore, in our input dataset used to measure the accuracy, we removed all flows that are too short.

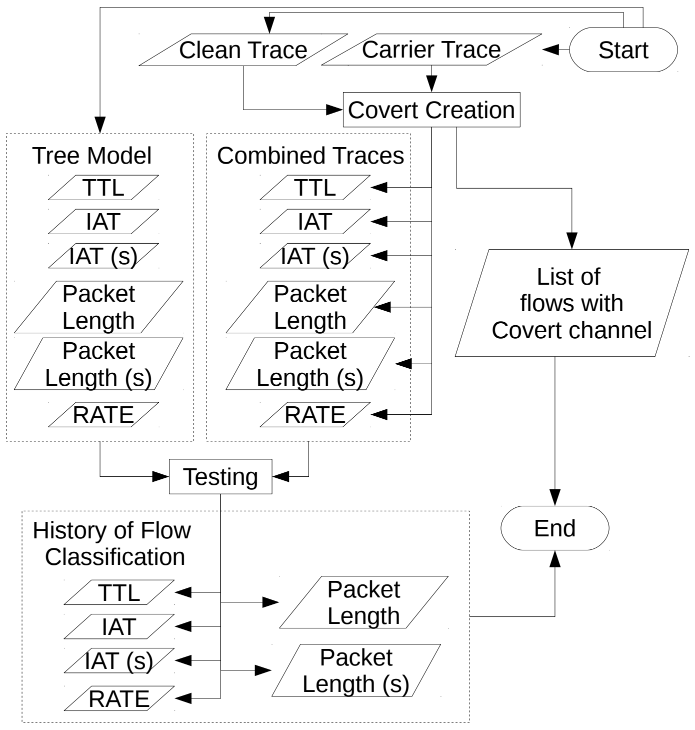


Figure 3: Testing accuracy methodology part 1

and true negatives from the classification output of Bro. There are two modes to classify whether a flow is a covert channel or not: based on the number of consecutive covert classifications or the ratio of covert and non-covert classifications (as described above). We can define the threshold for each of these modes. The case where we label a flow as covert flow as soon as we see one covert classification can be simulated using the consecutive classification method with a threshold of one ($t = 0$).

2.5 Performance Measurements Methodology

To test the performance and memory consumption, we pick arbitrary auxiliary data for the Analysis Plugin and tree models. We iterate over various analysis parameters and record the execution time of Bro, i.e. how long Bro needs to process the input trace file. We measure execution times, since Bro always uses 100% of the CPU (one CPU core to be precise) when working on trace files and so CPU load measurements are not useful.

We also use a tool called "pidstat", which measures various statistics of an executed program, to record the average memory usage given by the Resident Set Size (RSS) statistic. RSS is the non-swapped physical memory used by a process in Megabytes (MB). Note that the physical memory of our test machine was large enough, so that no swapping occurred during the active experiments.

3 Accuracy

In this section, we report the accuracy in terms of true positive rate and the false negative rate for various parameters. Note that for measuring the accuracy, we use the classification method where a flow is classified as containing a covert channel when $t = 5$ consecutive analysis results indicate a covert channel.

We use the following abbreviations for the different covert channels:

- IAT (modulo inter-packet channel by Sha [4]);

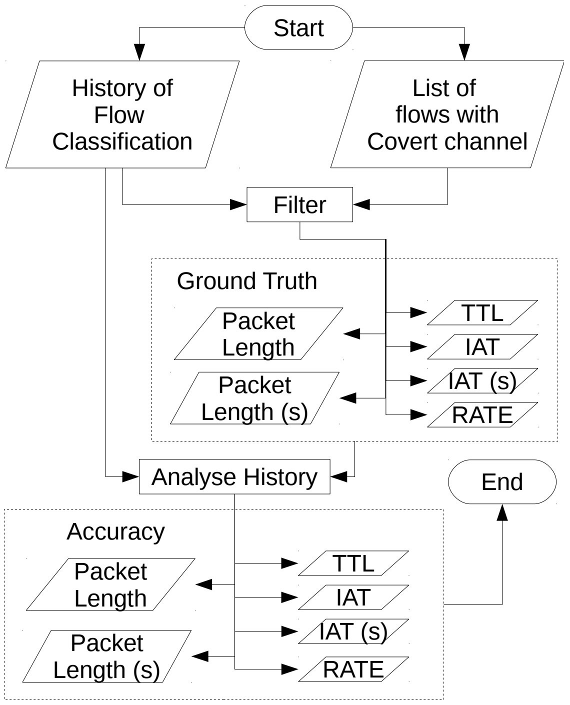


Figure 4: Testing accuracy methodology part 2

- IAT_SIMPLE (binary channel with direct mapping of 0/1 bits to low/high inter-packet gaps);
- PLEN (NTNCC [3] packet length channel);
- PLEN_SIMPLE (binary channel with direct mapping of 0/1 bits to small/large packet lengths);
- RATE (binary channel with direct mapping of 0/1 bits to zero rate and non-zero rate);
- TTL (binary channel encoded in the TTL field [10]).

3.1 All Parameters

First, we evaluated the accuracy for detecting the different covert channels using all different analysis metrics with three different configurations. In these experiments the step size was fixed to 250. Note that where the window size exceeded the size of the flow (as described above the minimum size of flows was 500 packets), at least one incomplete window was classified. For example, if window size was set to 1000, a flow with 500 packets would be classified once based on the 500 packets.

In the following we describe our parameter settings. We used three different sets of parameters: LIGHT, MID and HEAVY (see Table 3). In the LIGHT configuration, the parameters for each of the analysis metrics is configured so that it is the least resource intensive. The MID configuration is the middle ground between LIGHT and HEAVY. In the HEAVY configuration, we set the parameters so that the analysis will be the most resource intensive.

Figure 5 shows the true positive rate and false positive rate for the different covert channels depending on the three parameters sets describe above. Changing from LIGHT to MID does significantly improve the true positive rate for PLEN while true positive rates for the other channels remain largely the same. On the other hand, changing from LIGHT to MID slightly increases the false positive rate for PLEN, RATE and TLL while reducing the false positive rate for IAT.

Table 3: LIGHT, MID and HEAVY parameter settings

Parameter	LIGHT	MID	HEAVY
Window size for AC	100	500	1000
Number of bins for EN and MM	8192	16192	65536
Number of bins for CCE	5	10	15
Pattern size for CCE	5	10	15
AC lags	10	50	100
Window size for REG	100	500	1000
Number of windows for REG	100	500	1000

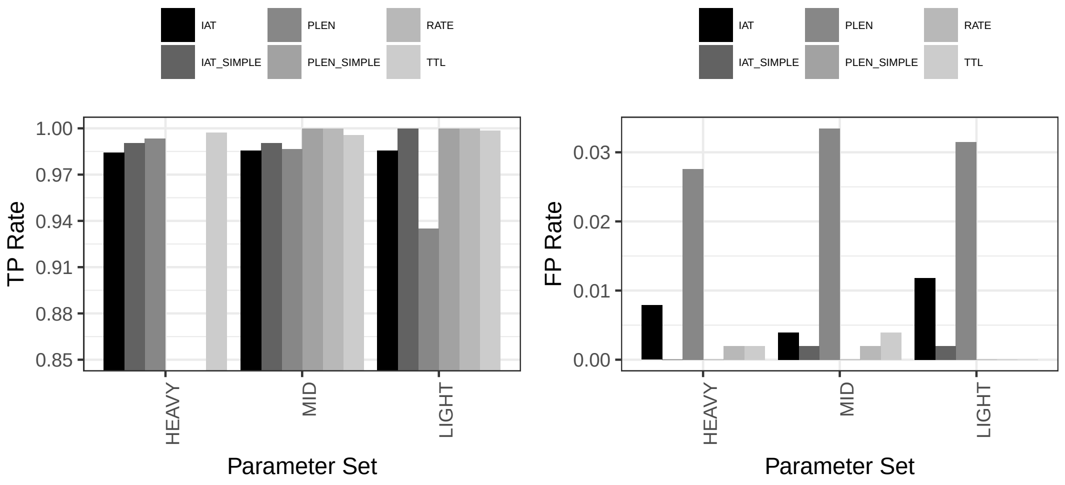


Figure 5: Accuracy for Various Parameters

3.2 Step Size

We also ran an experiment to quantify the effect of step size on the accuracy as shown in Figure 6. In this experiment all other parameters were set to LIGHT. A smaller step size means that there are more data points generated, which may result in overfitting. A smaller step size also means that there are less packets missing from the training. This is because only the packets contained in the last window are used for some of the analysis (such as AC), which means that there will be missing packets equal to the difference between the window size and the step size. The results show that there is not much difference between step sizes of 250 and 500. When decreasing the step size from 250 to 50, the true positive rate of PLEN is increased, but at the cost of an increased false positive rate, while for the other channels there is no significant increase of the true positive rate, but the false positive rate is increased.

Note that with the current BroCCaDe implementation a small step size may produce zero values for CCE and regularity, if the step size is smaller than the window size or the CCE pattern length. This can have a negative impact on the classification accuracy. In the rest of the experiments in Section 3 the step size was set to 250.

3.3 Parameters for Regularity Test

For this experiment, we varied the window size and the number of windows for the regularity analysis. Figure 7 shows the results with only the regularity analysis (top) and the combination of all analysis metrics (bottom).

When we look at the individual regularity analysis, we can see that increasing the window size tends to increase the true positive rate but it also increases the false positive rate, while there is a better improvement

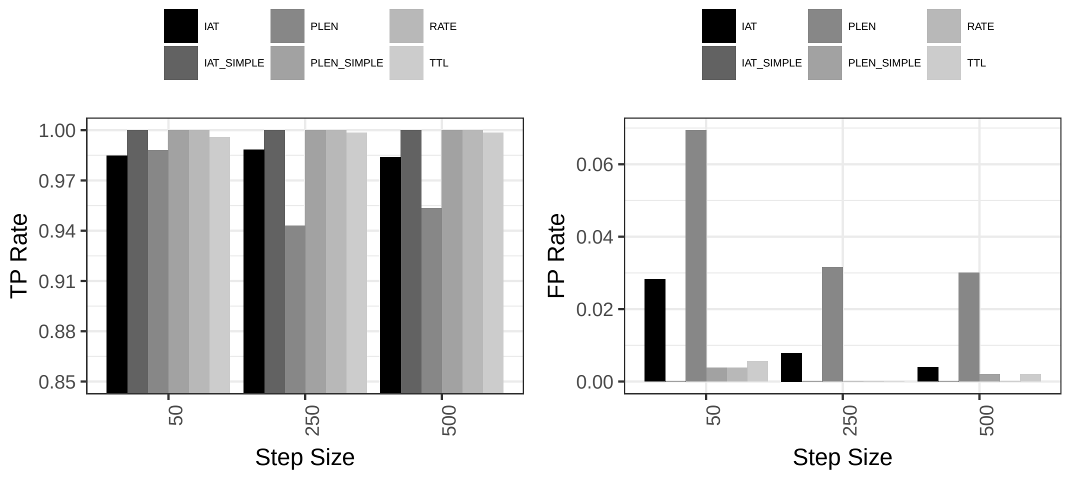


Figure 6: Accuracy for Step Size

with less adverse effects from increasing the number of windows. Based on this alone, we can argue that it is better to have a smaller window size and larger number of windows. However, when we look at the combined analysis, it seems that a good trade-off for the window size is somewhere in the middle ground, where the true positive rate is high and the false positive rate low. In general, increasing the number of windows seems to be good since it increases the true positive rate and reduces the false positive rate.

3.4 Parameters for Entropy Analysis

In this experiment, we varied the number of bins for entropy analysis. Figure 8 shows the results with only the entropy analysis (top) and the combination of all analysis metrics (bottom).

When we look at the performance of entropy analysis in isolation, it seems that increasing the number of bins tends to lower the false positive rate while roughly maintaining the true positive rate. With all metrics combined however, there is not much impact of the number of bins on the accuracy (except for IAT where it reduces the false positive rate somewhat). From this result we can also see that entropy analysis performs well for the TTL, IAT_SIMPLE and PLEN_SIMPLE covert channels, i.e. it provides almost 100% true positive rate while it has negligible false positive rate. Entropy also performs quite well for IAT. It is not useful for PLEN.

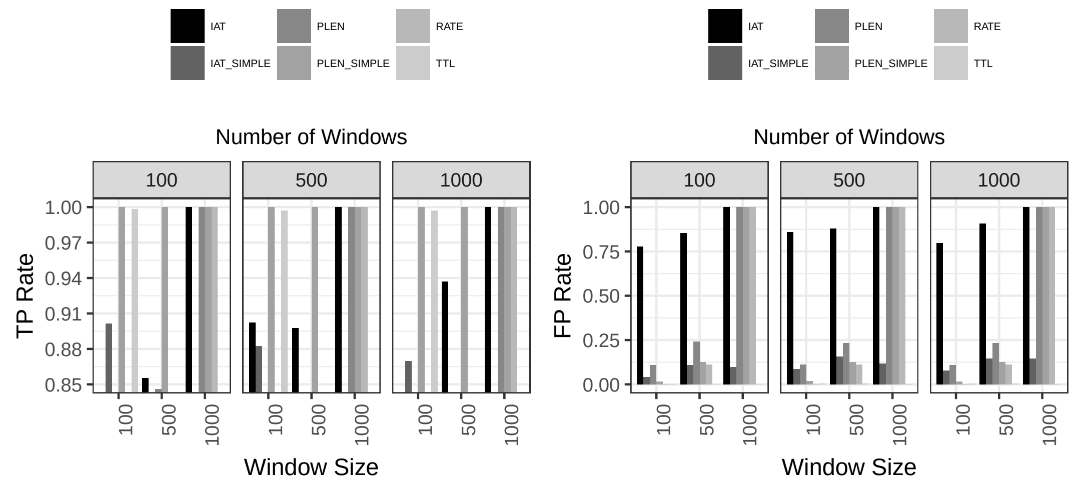
3.5 Parameters for Multi-Modality Analysis

We also examined the effect of varying the number of bins on the multi-modality analysis. Figure 9 shows the results with only the multi-modality analysis (top) and the combination of all analysis metrics (bottom).

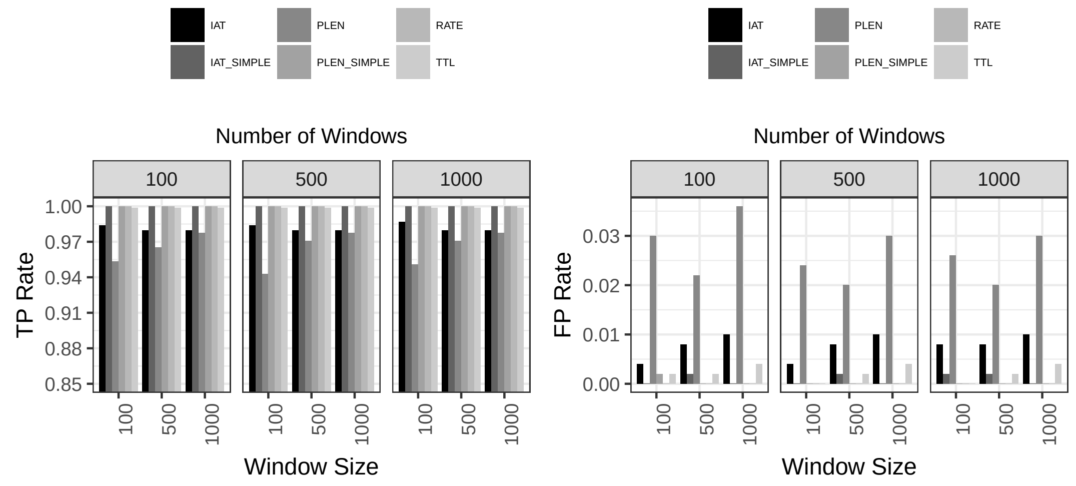
Since multi-modality analysis shares the same histogram data as entropy analysis, the result for overall accuracy is the same as in Figure 8 (bottom). The results suggest that unlike for entropy increasing the number of bins is not necessarily good for multi-modality analysis. We can see that there is a drop in the true positive rate when increasing the number of bins from 16192 to 65536. Multi-modality analysis seems to be handling TTL and PLEN_SIMPLE well, but for other channels either the true positive rate is very low or the false positive rate is very high.

3.6 Parameters for CCE Analysis

In this experiment, we varied the number of bins and the pattern size for the CCE metric. Figure 10 shows the results with only CCE analysis (top) and the combination of all analysis metrics (bottom).



(a) Using only regularity metric



(b) Using all metrics

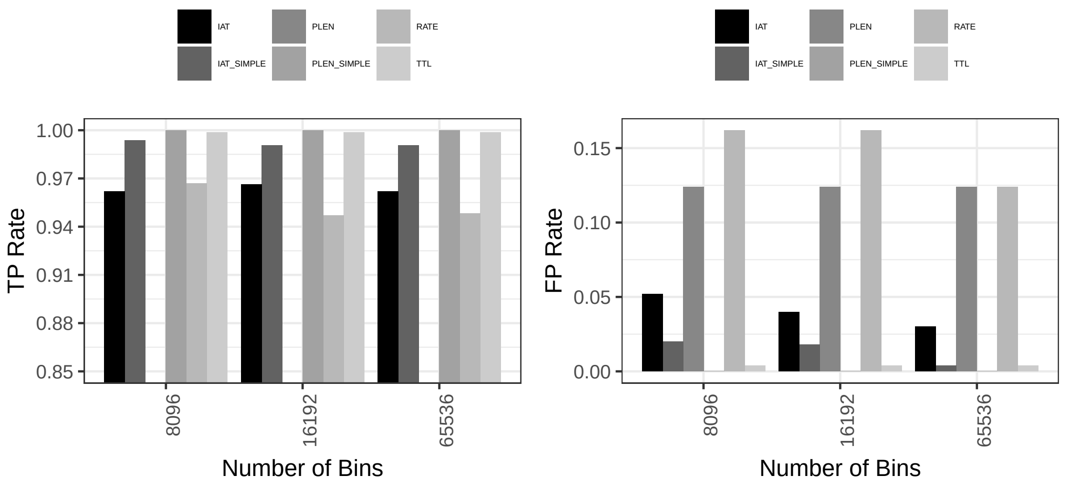
Figure 7: Accuracy for regularity depending on the window size and the number of windows

Looking at the individual CCE analysis, it seems that increasing the number of bins benefits some channels while adversely affecting other channels in terms of false positive rate. There is also not much benefit from increasing the pattern size beyond 5. These findings also apply to the overall result on the effect of varying CCE parameters, although here it is mainly only PLEN that benefits from a higher number of bins in terms of a reduced false positive rate.

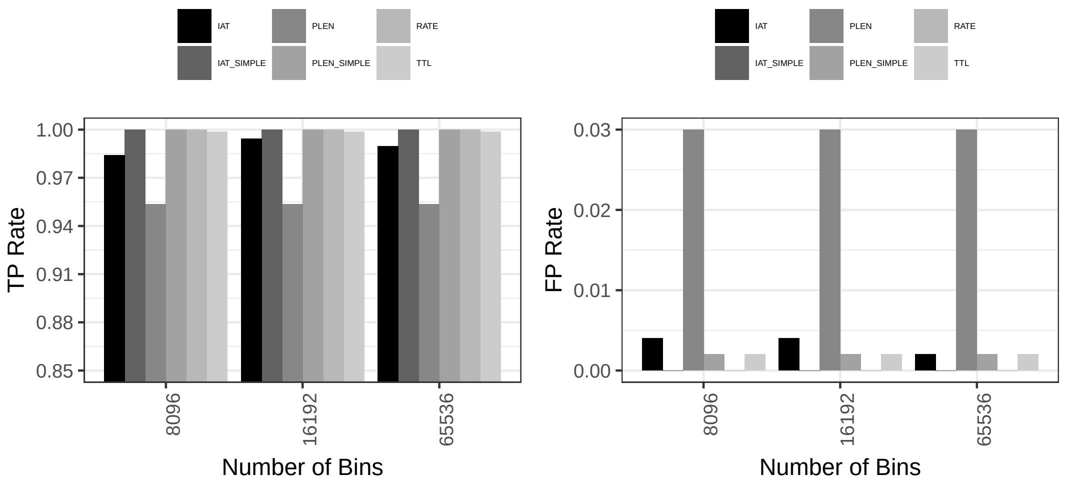
3.7 Parameters for Autocorrelation

In this experiment, we vary the window size and the number of lags for the AC analysis. Note that a number of lags l means we calculate the sum of the autocorrelation for lags $1, \dots, l$. For example, if $l = 10$ we calculate the sum of the autocorrelation for lags $1, \dots, 10$. Figure 10 shows the results with only AC analysis (top) and the combination of all analysis metrics (bottom).

Based on the result, in terms of the true positive rate it seems that increasing both parameters yields better results for most of the channels except both inter-arrival time channels. Increasing the number of lags increases

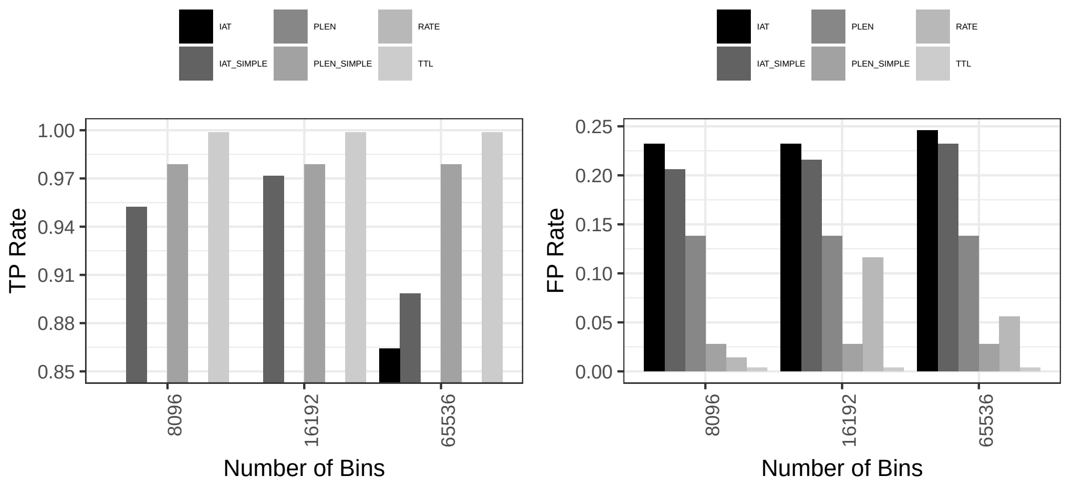


(a) Using only entropy

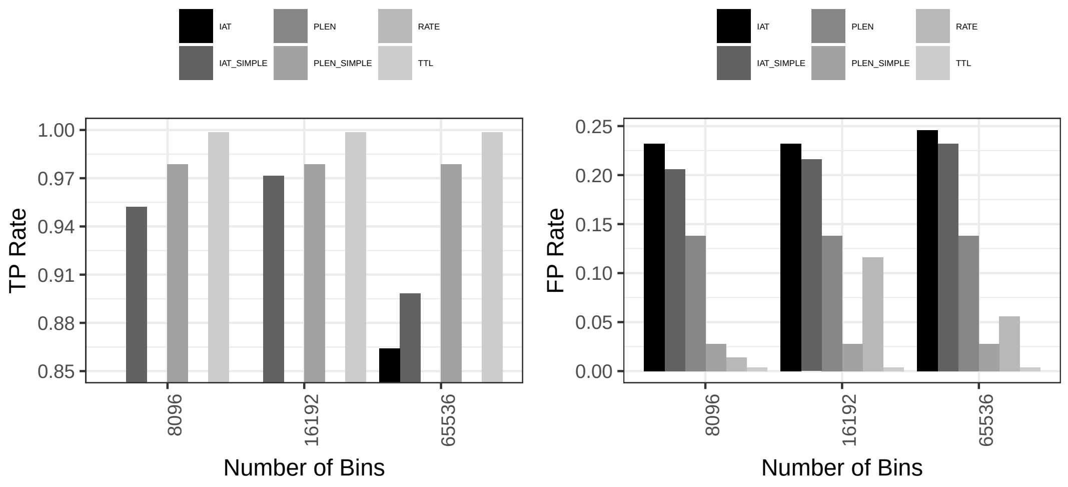


(b) Using all metrics

Figure 8: Accuracy for entropy depending on the number of bins

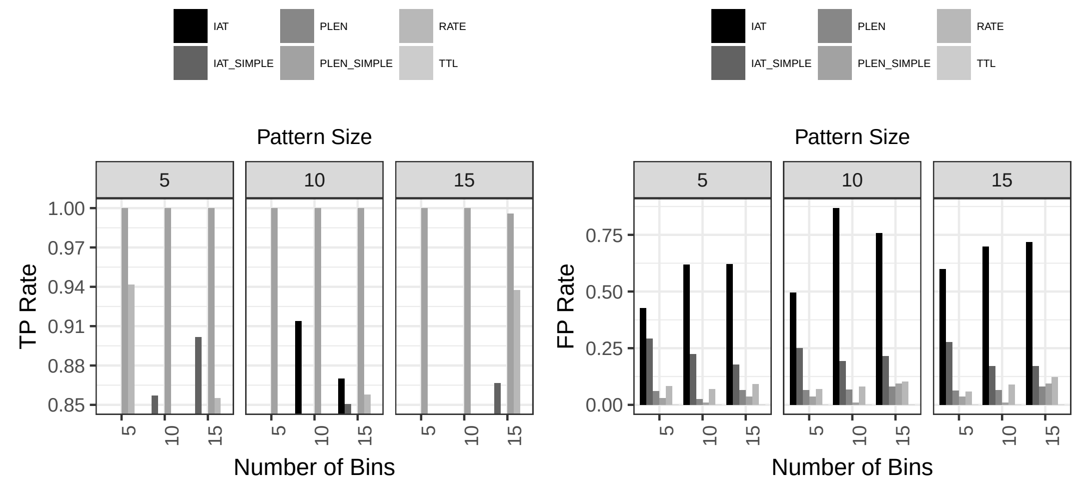


(a) Using only multi-modality

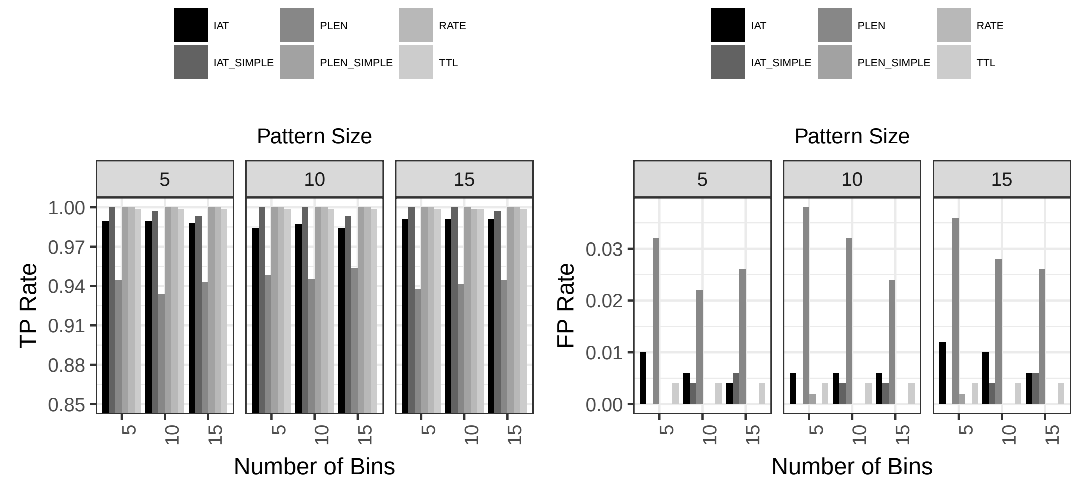


(b) Using all metrics

Figure 9: Accuracy for multi-modality depending on the number of bins



(a) Using only CCE



(b) Using all metrics

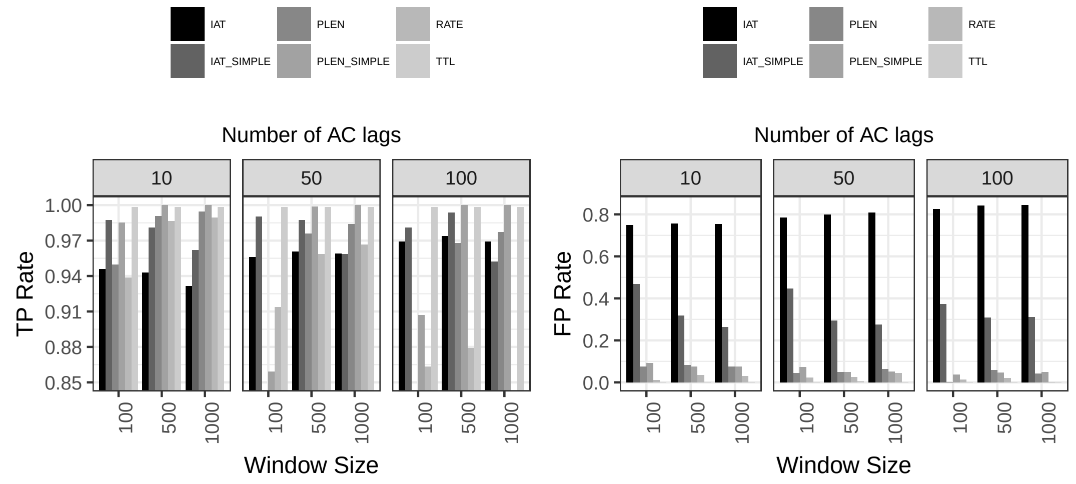
Figure 10: Accuracy for CCE depending on the number of bins and the pattern size

the false positive rate although not by much. Increasing the window size lowers the false positive rate for IAT_SIMPLE. The observation for the true positive rate also applies to the combined analysis. However, it is a different story to the false positive rate as there is a decrease in the false positive rate when increasing the number of lags.

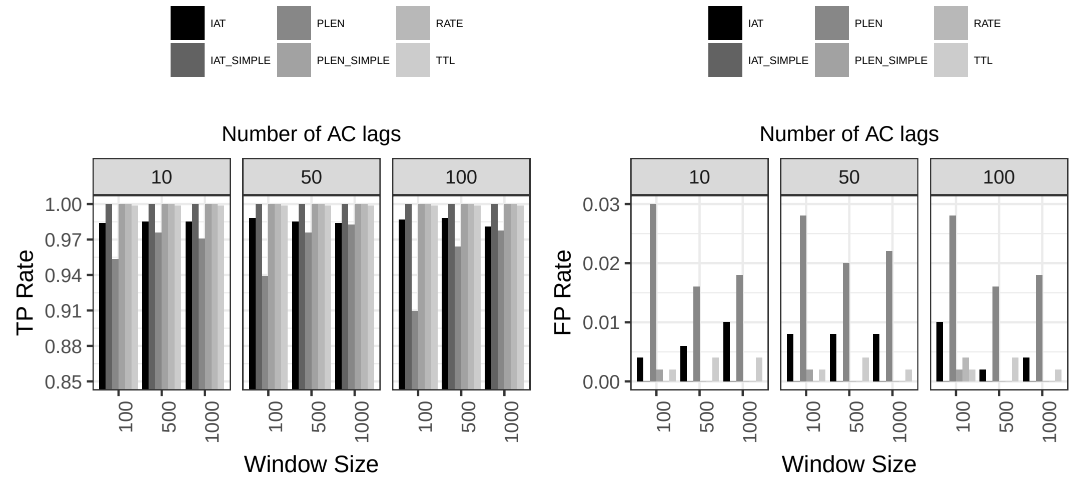
3.8 Conclusions

Based on the experiments that we have conducted, we arrived at the following conclusions:

- Increasing the number of bins for entropy analysis and multi-modality analysis has a positive effect on accuracy. Although, when using multi-modality as the only metric a high number of bins is not necessarily better, the small adverse effect is more than offset by a significant benefit of using a higher number of bins for entropy analysis.
- Increasing the window size and number of autocorrelation lags for AC analysis is also beneficial for the classification accuracy.



(a) Using only autocorrelation



(b) Using all metrics

Figure 11: Accuracy for Autocorrelation depending on window size and number of AC lags

- For CCE we suggest setting the parameters to the minimum used in our experiments (in our case 5 for both the number of bins and pattern size) as larger values do not significantly improve accuracy.
- Multi-Modality, as expected, is very good to detect covert channels that create multiple modes (multiple peaks in a feature values histogram) and its accuracy is higher with a low to moderate number of bins.

3.9 Feature Selection Discussion

Multi-modality, entropy and autocorrelation work well to detect TTL, packet length simple and simple inter-arrival time covert channels. This is because these channels create some distinct modes. Multi-modality works well in this case because it will be able to pick up distinct modes which correspond to the covert bit encoding. The distinct modes also mean that values are more predictable, i.e. entropy values are lower, which means that entropy-analysis also works well for these covert channels. Autocorrelation is also able to detect these channels because of the changed correlation between values compared to normal traffic.

As already highlighted by Gianvecchio et al. [11], entropy analysis performs quite well to detect the presence of jitterbug (modulo inter-arrival time channels). The other metrics seem to fail to properly detect this type of covert channel due to various reason. Obviously, the channel does not necessarily create distinct modes, so multi-modality is not a good fit. The low granularity of CCE means that it is not sensitive enough to pick up the slight changes in the inter-arrival time values.

Autocorrelation seems to be able to detect NTNCC while the other metrics fail. This is because the NTNCC destroys the correlation even though it somehow maintains the value distribution.

The rate channel can be detected by autocorrelation, entropy, and CCE. This is because fixing the intervals between packets creates some sort of predictability, where entropy and entropy rate will be useful. That said, using entropy still results in a significant false positive rate.

4 Accuracy Depending on Flow Classification Mechanism

Here we present a comparison of the different approaches to classify flows. Figure 12 shows the classification accuracy of BroCCaDe based on classifying a flow as covert channel as soon as one analysis result shows that the flow is a covert channel. Based on the figure we can see that we achieve a very high true positive rate, but also a high positive rate especially for NTNCC.

Figure 13 shows the accuracy of BroCCaDe when a flow is classified as covert channel when some percentage of the analysis results r is positive. The figure shows that the true positive rate and the false positive rate tends decrease with an increase of r. A value of $r = 30\%$ is a good trade-off. At this rate the true positive rate is still high for all channels, while the false positive rate is significantly lower than for smaller r.

Figure 14 shows the accuracy result of BroCCaDe using the classification technique where we classify a flow as a covert channel when t consecutive positive analysis results are observed. As we can see from the figure, the true positive rate decreases with increasing t, but even for $t = 7$ the decrease is relatively small. On the other hand, increasing t significantly reduces the false positive rate. Setting t to 5 or 6 is the most beneficial in our experiments. Here the true positive rate is not impacted much, while the false positive is much smaller than for larger t.

Based on our experiments, it seems that the approach where we classify a flow as a covert channel when t consecutive windows classify the flow as covert channel is the best approach, i.e., it has a good trade-off between timeliness and true positive rate. The approach where we classify a flow as a covert channel as soon as an analysis result indicates a covert channel gives us a high false positive rate. The approach where we make a decision after a percentage of the analysis results r indicates that the flow contains a covert channel provides similar accuracy, but is slower to detect covert channels, i.e. it requires more analysis results before a decision can be made. For example, to estimate a rate of 30% with reasonable accuracy we found that we need at least 10 windows – twice as many as with the consecutive window approach. However, for off-line analysis, where flows are analysed after the fact, the percentage approach also works well.

5 Performance (CPU and Memory Usage)

Now we evaluate the performance of Bro with BroCCaDe (relative to the performance of unmodified Bro) in terms of CPU time and memory usage depending on different analysis parameter settings. The parameters we vary are step size, the window size for autocorrelation analysis, the window size and number of windows for the regularity metric, the number of bins and pattern length for the CCE metric, the number of bins for entropy and multi-modality metrics, the number of lags for the autocorrelation metric and whether we use the decision tree classifier or not.

The impact of BroCCaDe on the computational performance is analysed as the ratio of the measured execution time of Bro with BroCCaDe divided by the measured execution time of the unmodified Bro (the baseline execution time) for a particular parameter set. We refer to this as Execution Time Overhead (ETO). To characterise the impact of BroCCaDe on memory we measured the RSS (non-swapped physical memory used) of Bro with pidstat in MB (see Section 2.5).

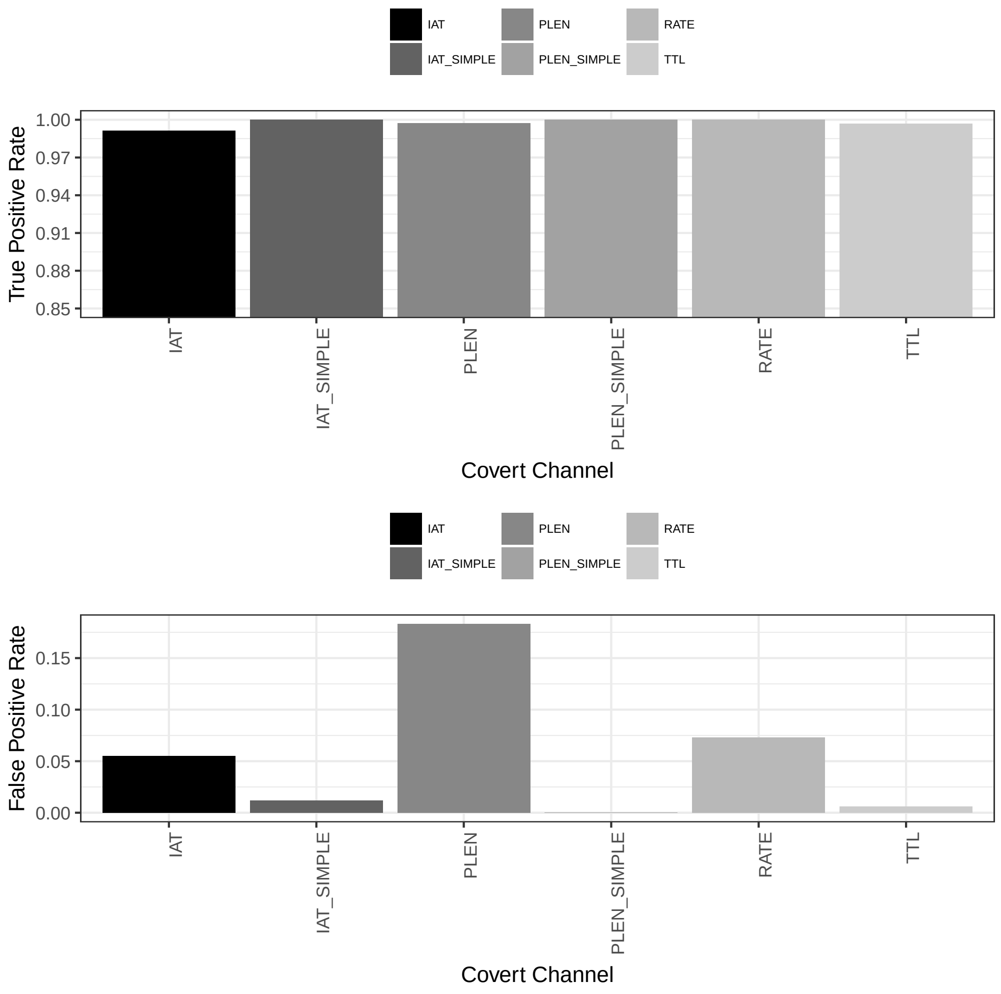


Figure 12: Accuracy when classifying a flow as covert channel as soon as we get one positive result from the analysis

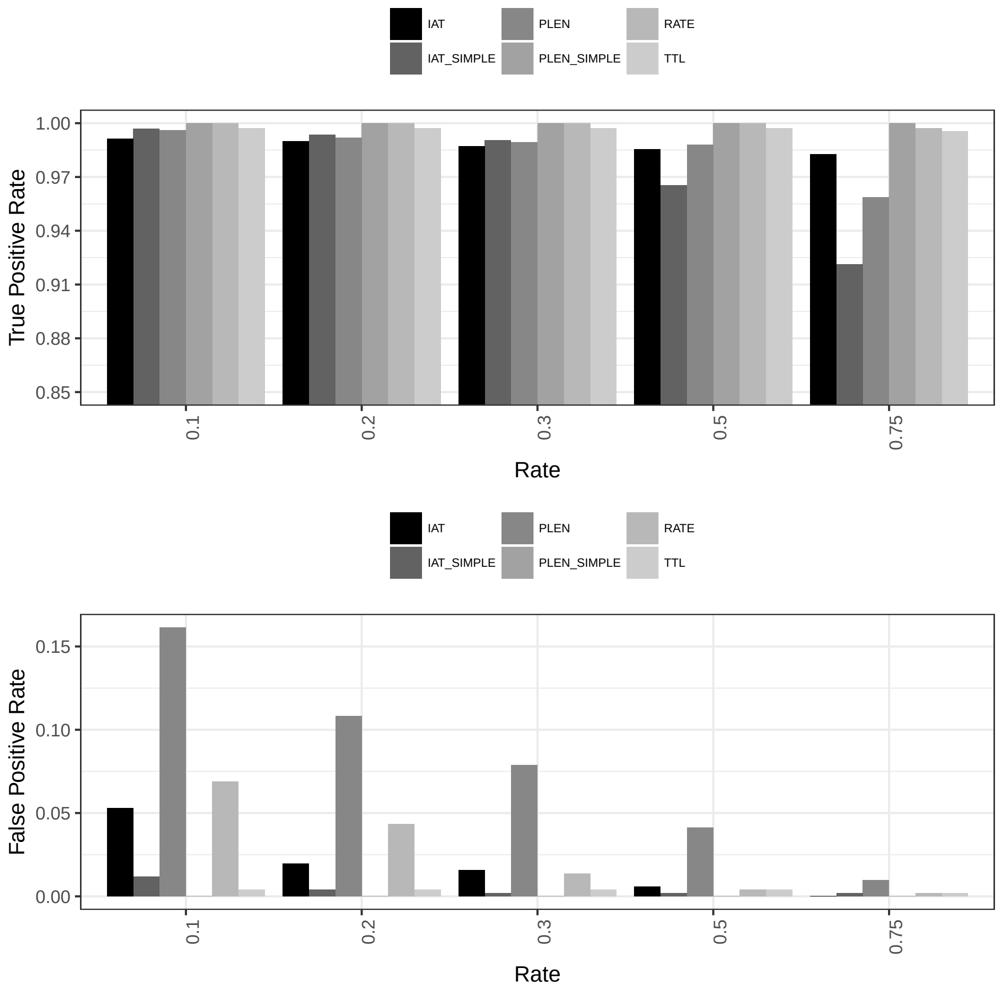


Figure 13: Accuracy based on the percentage of positive analysis results

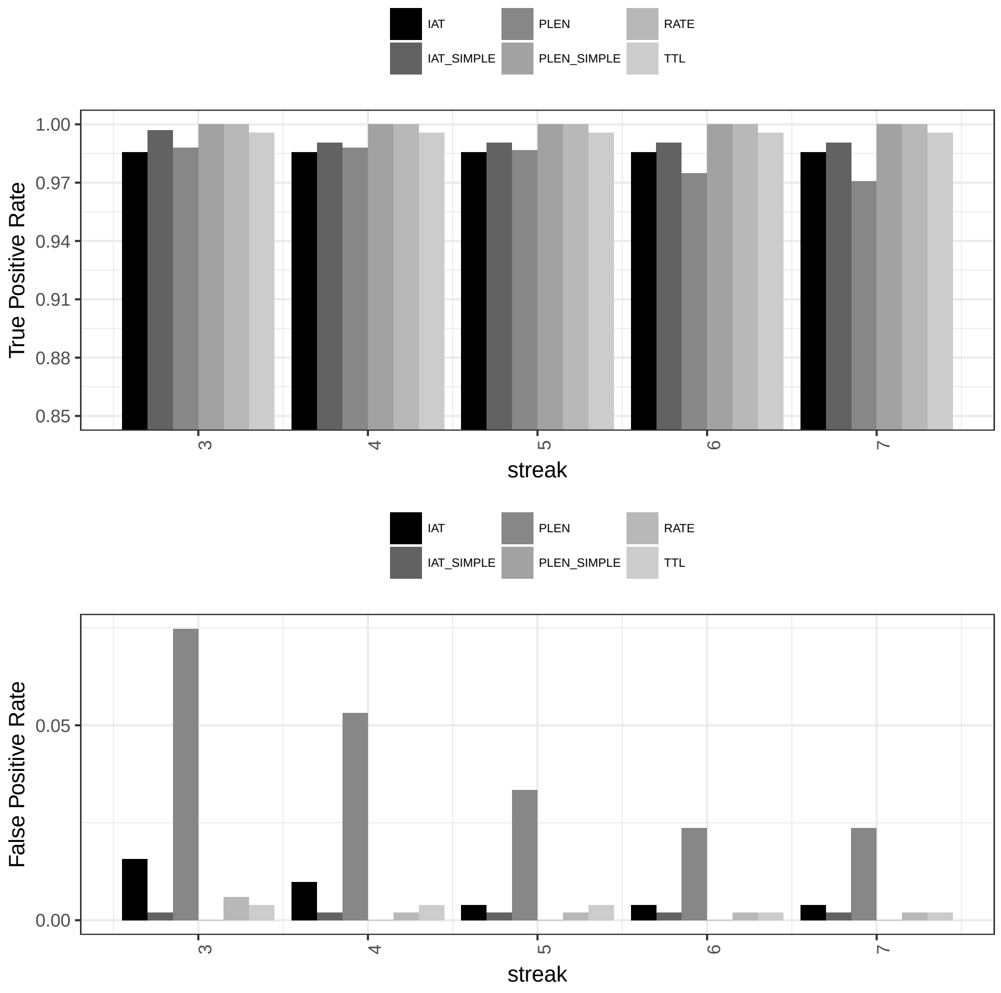


Figure 14: Accuracy based on consecutive number of positive analysis results

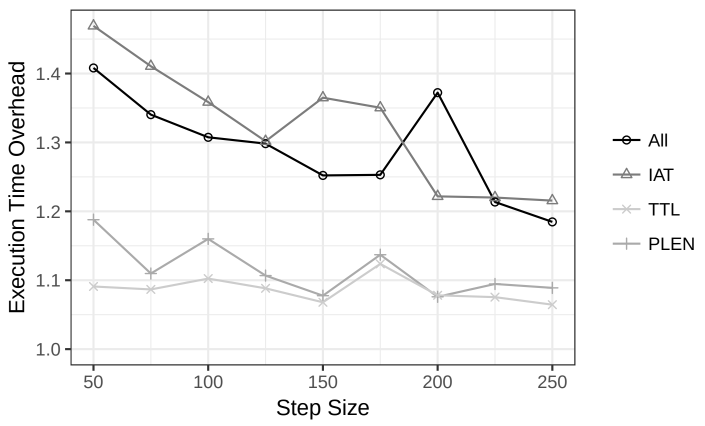


Figure 15: Execution time ratio depending on step size

All ETO and RSS measurements were done as a single run for each parameter combination. This means there is some noise in the measurements, but the trends are still useful. In future work we will perform multiple runs and report the average statistics over multiple runs.

5.1 Step Size

Figure 15 shows the ETO depending on the step size. From the graph we can see that a smaller step size increases the overhead in a roughly linear fashion as the number of metric calculations is inverse proportional to the step size. However, an increasing step size does not increase the memory consumption as the window size was constant during the experiment. We performed separate measurements for IAT, TTL, PLEN and all types of covert channels combined. From the results it is clear that the overhead is much smaller for TTL and PLEN than for IAT. This is mainly because for inter-arrival times there is a much wider range of values and the overhead for a number of metrics, e.g. CCE, is higher with a wider range. In addition, for IAT the inter-arrival times also need to be computed first, whereas for TTL and PLEN the values are basically provided by Bro. The overhead for TTL is lowest, as the value range of TTL is the smallest. Note that the overhead for ALL is lower than TTL, as All is a mix of all three types of channels.

5.2 Parameters for Regularity Test

Figure 16 shows the ETO depending on the number of windows, the window size and the step size. Since we are using the rapid calculation method for each window (we only keep the count, sum and sum of squares for each window, which will reset when the window is full), the window size does not have much impact on the ETO. The minor effect the window size has is that the standard deviation values are obtained quicker with smaller window sizes, e.g. if we define the number of windows to be 100, a window size of 10 means we obtain 100 standard deviations with 1000 packets, while a window size of 50 means we obtain 100 standard deviations with 5000 packets.

The number of windows used in the regularity calculation is affecting the overall performance quadratically in theory, since the complexity of the calculation is $O(n \cdot \frac{(1+n)}{2})$, where n is the number of windows, due to the calculation of pairwise standard deviation of the windows. However, in our experiments we do not have many flows that are long enough to fill more than 500 windows. Furthermore, while of quadratic complexity the actual calculation is very quick. Thus, in our experiments we found no significant increase in ETO for an increasing number of windows, but we expect this would be different with a dataset of flows that are all long enough to fill the maximum number of windows.

The amount of memory required is independent of the window size due to using the rapid calculation method and only increases with the number of windows. However, since with the rapid calculation method only three

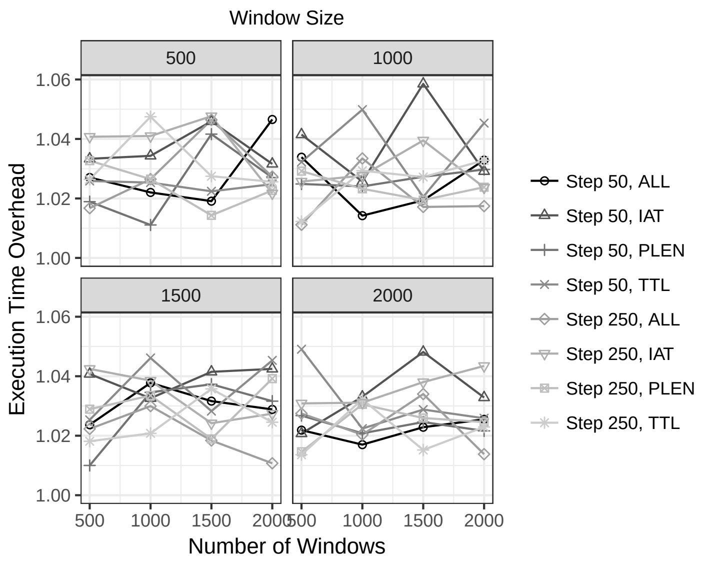


Figure 16: Execution time overhead for regularity depending on number of windows, window size and step size

variables need to be kept per window, the memory increase depending on the number of windows is negligible compared to the rest of the memory used by Bro (with the number of windows being between 500 and 2000). In our experiments, we also do not have the full number of windows for each flow. Hence, the measured RSS is fairly stable around 64–65MB and any variations are entirely due to noise, and thus we do not show a graph here.

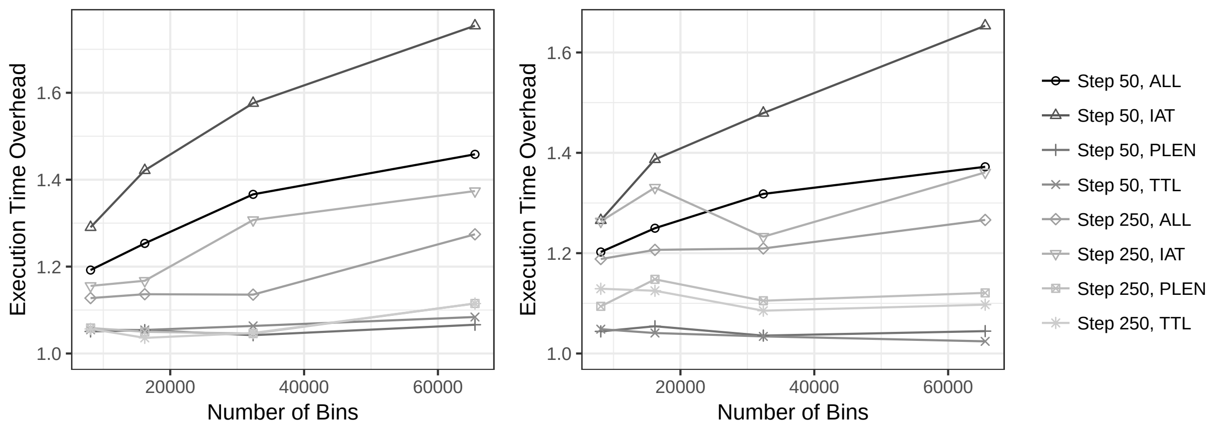
5.3 Parameters for Entropy and Multi-Modality Test

Figures 17a and 17b shows the ETO depending on the number of bins used for entropy and multi-modality respectively. Results are shown for the smallest step size 50 and the largest step size of 250. The number of bins for the Entropy and Multi-Modality increases the ETO logarithmically. This behaviour is because a smaller number of bins implies larger bin sizes. When bins are large, many values fall into the same bin, which means that there are fewer bins used to calculate the metric. Consistent with the results in Section 5.1, the overhead increases with decreasing step size and the overhead for PLEN and TTL is insignificant compared to the more substantial overhead for IAT. Also, as expected the overhead for ALL (a mix of the different channels) is between the overhead for IAT and PLEN/TTL.

For the reasons explained above, the increase in memory consumption also follows a logarithmic pattern as shown in Figure 18.

5.4 Parameters for CCE Analysis

Figure 19 shows the ETO depending on the number of bins and the pattern size for CCE. Both parameters have significant impact on the overhead for CCE. This is because of the tree structure used for the CCE calculation, where the complexity is $O(n^k)$ in terms of pattern size (depth) n and bin numbers (possible branches) k. In practice, we can see that the number of bins increase the overhead and memory consumption logarithmically while the pattern size increases the overhead exponentially. The logarithmic behaviour for the number of bins can be explained in the same way as above for the number of bins for entropy and multi-modality metrics. The exponential increase for pattern size is due to the exponential increase in the number of tree nodes for increasing pattern size.



(a) Entropy

(b) Multi Modality

Figure 17: Execution time overhead for entropy and multi-modality depending on the number of bins

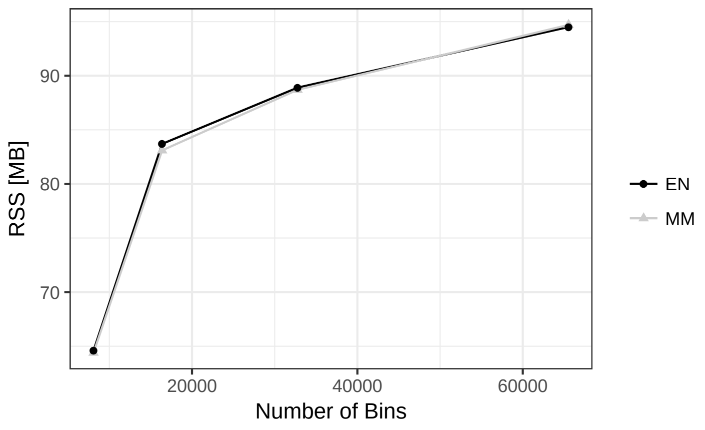


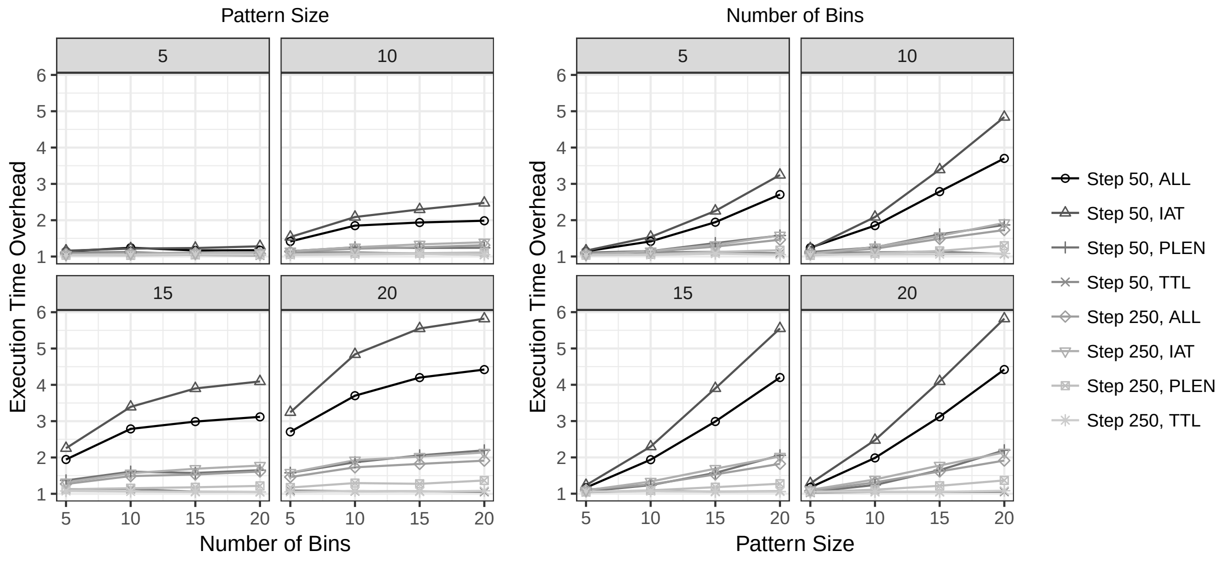
Figure 18: Memory consumption for entropy and multi-modality depending on number of bins

Figure 20 shows the memory consumption which follows the same trends as the ETO for the reasons explained previously.

5.5 Parameters for Autocorrelation

Figure 21 shows the ETO depending on the number of AC lags and the window size for the autocorrelation. From the figure we can see that the ETO increases roughly linearly for the number of lags, bounded by the window size used to calculate the autocorrelation. This is due to the main loop of the metric calculation which iterates over $(N - lag)$ data points for each lag, where N is the size of the window. The resulting complexity for the number of lags is $O(lag \cdot \frac{(2N-lag-1)}{2}) \approx O\left(N \cdot lag - lag^2\right)$ and since N is much larger than the lags the expected increase is roughly linear. The ETO also increases linearly with the window size since the algorithm iterates over all of the data points from the current window.

Figure 22 shows that the memory consumption increases linearly with the window size as the window size determines how much data is stored but is unaffected by the number of AC lags as independently of the number of lags only the data from one window is used (differences in the figure are simply noise).



(a) Number of bins

(b) Pattern size

Figure 19: Execution time overhead of CCE depending on the number of bins and pattern size

5.6 Using Decision Tree Classifier

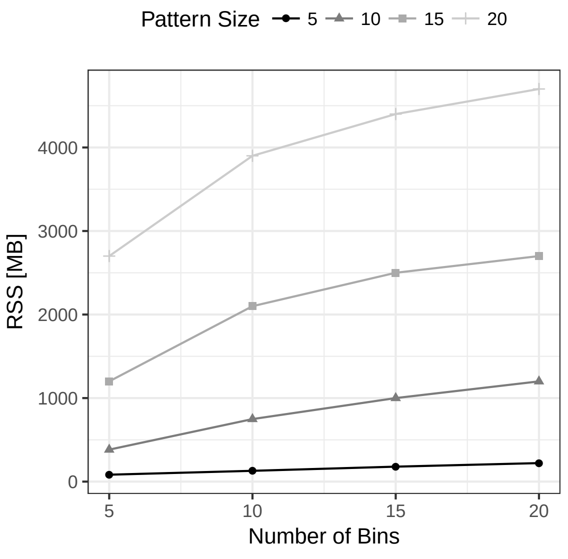
We also compared the run time of BroCCaDe for the different covert channels with the decision tree classifier and without the decision tree classifier. Figure 23 shows the results in absolute execution time when all metrics were used (LOW settings). Based on the results we conclude that decision tree classifier has a negligible impact on the overall performance. The overhead of BroCCaDe is dominated by the feature extraction and metric calculation.

6 Conclusions

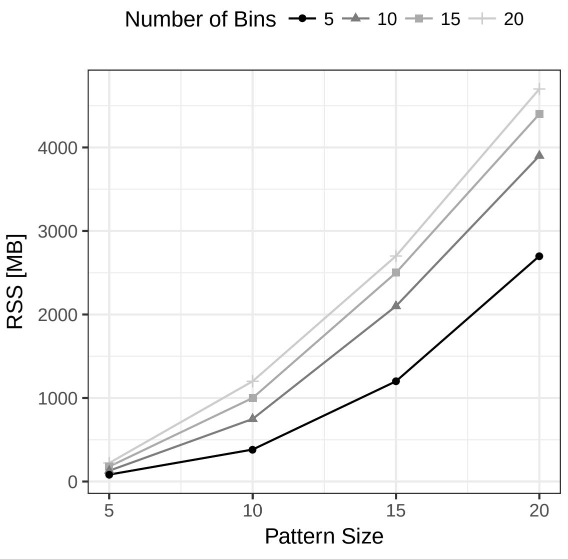
This report describes experiments we have carried out with BroCCaDe to measure its performance in terms of classification accuracy and CPU and memory overhead. We tested BroCCaDe with a number of different traffic types and covert channels embedded in the IP TTL field, packet length, inter-packet time and packet rate. Our results show that BroCCaDe can identify these channels with a high accuracy (true positive rate of 98% and false negative rate of generally less than 1%).

Our performance analysis reveals that BroCCaDe requires a small to moderate additional amount of RAM and CPU time. Most of the implemented metrics, apart from CCE, have a low overhead even with high parameter values compared to the overhead caused by Bro events and storing the feature values. For CCE the overhead is still acceptable for low parameters settings for which CCE already performs well in terms of accuracy; with high parameter settings the overhead is too high for CCE to be of practical use. Compared to standard Bro, BroCCaDe increases the resource usage by only a few MB, except for the CCE metric where the increase is more substantial, and the overhead in terms of CPU time is significant but generally under 50%.

A limitation of our study is the trace data we used. Future studies could extend our study with larger sets of flows from more diverse datasets. Our study is based on UDP traffic only, but some of the covert channels we use, such as the packet length and the simple inter-packet timing channels, can only be implemented with UDP traffic. However, future studies could include TCP traffic for channels that can be applied to TCP, such as the TTL channel.



(a) Number of bins



(b) Pattern size

Figure 20: Memory consumption for CCE depending on the number of bins and pattern size

Acknowledgements

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

References

[1] Hendra Gunadi and Sebastian Zander. Bro Covert Channel Detection (BroCCaDe) Framework: Design and Implementation. Technical Report 20171117B, School of Engineering and IT, Murdoch University, 2017.

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

[3] Liping Ji, Haijin Liang, Yitao Song, and Xiamu Niu. A normal-traffic network covert channel. In *International Conference on Computational Intelligence and Security (CIS)*, pages 499–503, 2009.

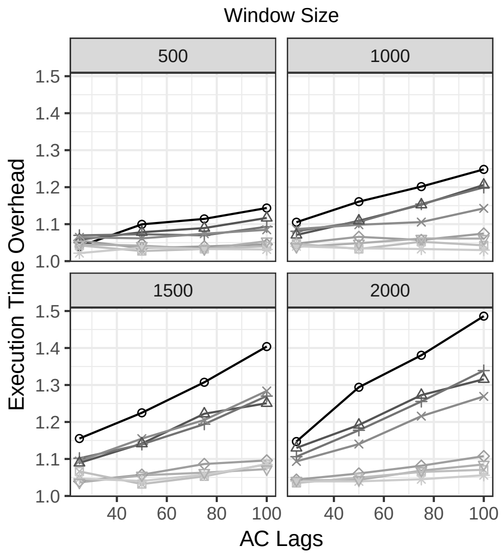
[4] Gaurav Shah, Andres Molina, and Matt Blaze. Keyboards and covert channels. In *USENIX Security Symposium*, volume 15, 2006.

[5] S. Zander. Covert Channels Evaluation Framework (CCHEF). `https://sourceforge.net/projects/cchef/`, Accessed: 11 September 2017.

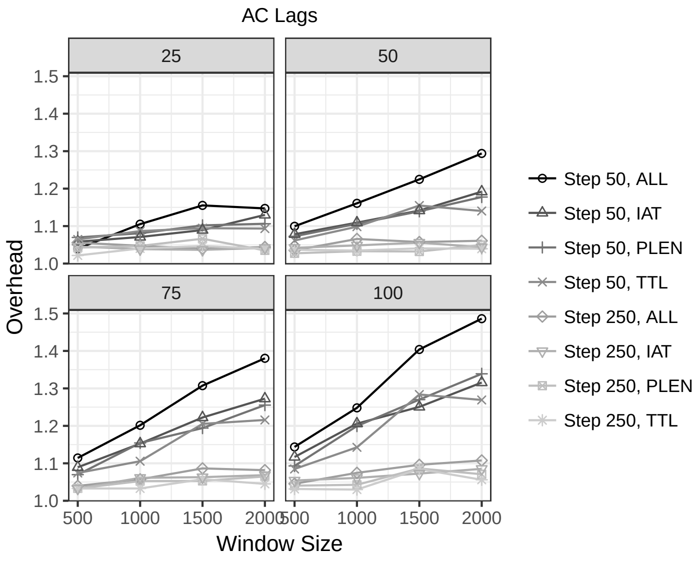
[6] SONG - Simulating Online Networked Games Database. `http://caia.swin.edu.au/sitcrc/song/`, Accessed: 11 September 2017.

[7] Tstat - TCP STatistic and Analysis Tool. `http://tstat.polito.it/traces-skype.shtml`, Accessed: 11 September 2017.

[8] WITS: Waikato Internet Traffic Storage - ISPDSL II. `https://wand.net.nz/wits/ispdsl/2/`, Accessed: 11 September 2017.



(a) AC lags



(b) Window size

Figure 21: Execution time overhead for autocorrelation depending on the AC lags and window size

[9] MAWI Working Group Traffic Archive. `http://mawi.wide.ad.jp/mawi/`, Accessed: 1 November 2017.

[10] Sebastian Zander, Grenville Armitage, and Philip Branch. Covert channels in the IP time to live field. In *Australian Telecommunication Networks and Application Conference (ATNAC)*, 2006.

[11] 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.

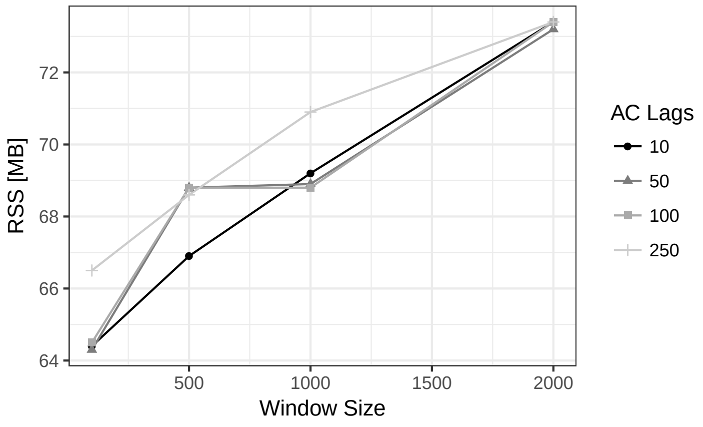


Figure 22: Memory consumption for autocorrelation depending on the AC lags and window size

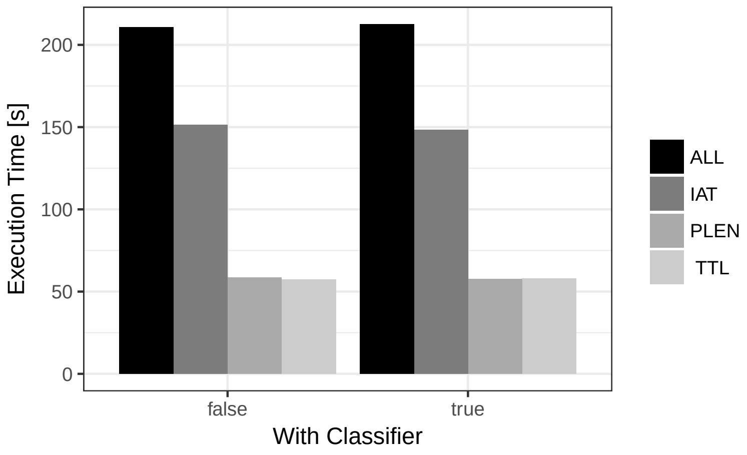


Figure 23: Execution time with and without the decision tree classifier