Project IRESI

Simon Bihel, Florestan De Moor
ENS Rennes, Computer Science Department, 1st year

November 22, 2015

Abstract

In this report we present our work on the project of the course IRESI. We will first explain how we implemented the `SketchMin` algorithm, which is a metric used to detect DDoS. We coded this using the Python 3 language. We will then compare the results of the algorithm to exact values.

1 DDoS and server security

A DDoS (Distributed Denial of Service) attack is nowadays a well-known issue : it consists of sending a huge number of requests to a server, so that it crashes, and therefore to make it unavailable for the other users. The requests are sent to the server by the router, which receives data streams. If the attack is concentrated in a single stream, it is easy do detect it.

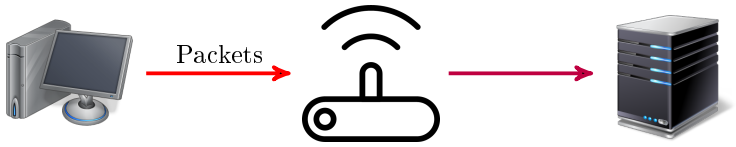

Figure 1: One user sending requests to a server, through the router

However, if the attack is scattered among several streams, it is much more difficult to detect it.

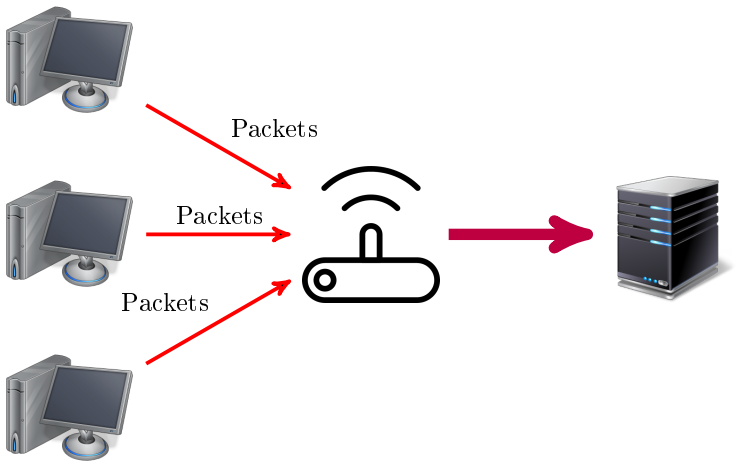

Figure 2: Three users sending requests to a server, through the router

That's why the following method was implemented : the router receives different data streams, and calculates for each peer a correlation indicator which is the codeviance of the frequency vectors corresponding to the streams. The codeviance formulas are presented below in section **2.2**. Once this was computed, it becomes possible to determine if an attack is happening or not, by analysing the codeviance values.

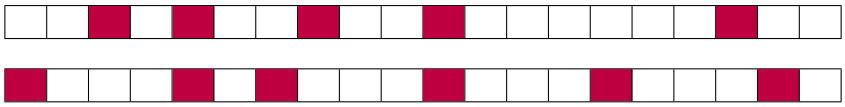

Figure 3: Correlation between two data streams

Nevertheless, the stream sizes are huge, we can't afford to calculate the precise codeviance. The `SketchMin` algorithm is an easier way to calculate an estimated codeviance using hashing functions in which precision is controlled by a set of parameters (ε, δ). We will present in next section how we implemented it using the Python 3 language. We will then compare its results with exact values in section **3**, by running it with real and randomly generated data traces.

2 Programming the SketchMin algorithm

For the detection we need to first select the entries to then apply the algorithm that will produce the material in which we will find whether or not there is an attack.

2.1 Extracting information from data traces

In data traces, each line corresponds to a request made to the server. To extract a certain kind information, like the source of a request, there is just to go through each line which represents a line. And each time split the line based on white-spaces and extract a certain entry. The entries are put in a list, and if needed are converted to integers without losing the fact that some entries are identical and others different. This can be done by creating a list of only the distinct elements and then converting each element of the original list to its index in the distinct elements list.

This process has to be extended to the extraction of multiple traces in a single time because what we want in the end is eventually find correlation between multiple traces. This was the most difficult part of the work. Same entries must be converted to the same integer, even if they're not in the same trace. We were first making the extraction considering only one trace, but we finally had to make the extraction of all traces at once. To do that the distinct elements list has just to be shared between the multiple extractions. And so, identical requests can be correlated after applying the `SketchMin` algorithm and act in reaction.

2.2 Computing the codeviance

First, we implemented two functions to compute the average, and codeviance of an array. To do this, we browse the array, and use the following formulas :

$$\mathrm{E}(X) = \frac{1}{|X|} \sum_{x_i \in X} x_i$$

$$\mathrm{Cod}(X, Y) = \mathrm{E}(XY) - \mathrm{E}(X)\,\mathrm{E}(Y)$$

We have in entry a data trace D, and precision parameters (ε, δ). We start by defining the following constants :

$$k = \left\lceil \frac{1}{\varepsilon} \right\rceil$$

k is the number of partitions we will create.

$$t = \left\lceil \log(\frac{1}{\delta}) \right\rceil$$

t is the number of hashing functions we will consider.

The data trace has integers values which stand between 0 and u. We have to define t universal hashing functions h_i :

$$h_i(x) = ((a_i x + b_i) \mod u) \mod k \quad \forall i \in \{1 \dots t\}$$

where a_i, b_i are randomly generated in $\{1 \dots t-1\}$ for a_i, and $\{0 \dots t-1\}$ for b_i.

We can then create the partitions, and we get a matrix $R_{0 \leq i \leq t-1, 1 \leq j \leq k}$ where

$$R_{i,j} = \#\{x \in D : h_i(x) = j\}$$

We can now create a function that receives two data traces. It calculates the R matrix for both trace, and then computes the codeviance :

$$\widetilde{\mathrm{Cod}}(D_1, D_2) = \min_{0 \leq i \leq t-1} \mathrm{Cod}(R_1[i], R_2[i])$$

where $R[i]$ is the line i of the matrix.

We can then create the full codeviance matrix $C_{1 \leq i,j \leq p}$ when we have p data traces :

$$C_{i,j} = \widetilde{\mathrm{Cod}}(D_i, D_j) \quad \forall i, j \in \{1 \dots p\}$$

We can notice that the codeviance is symmetrical (i.e. $\mathrm{Cod}(X, Y) = \mathrm{Cod}(Y, X)$). That's why only $p(p+1)/2$ computations are made to create this matrix (instead of p^2).

Once we calculated this matrix, we can plot it in 3D to have a visual result. To do this, we used the packages `numpy` and `matplotlib` from the Python library.

This is one of the reasons why we chose to program this algorithm using the Python 3 language : there are a lot of libraries that enable to create a visual plotting very easily. It makes it much easier to test the program, as we will see in next section. Another reason is that the Python language is very simple, with an intuitive syntax. It also deals with reading files quite well, and this was important to extract information from the traces in an effective way.

3 Experimental results

To evaluate the correctness of the algorithm, we just need to see if the results have the same appearance than the real values, with possibly a different scale.

3.1 Real data traces

For the real traces, we used the file-names as entries for the algorithm, after converting them to integers by simply matching the index of the distinct elements. We used file-names because the calgaryaccess traces didn't really have an entry for the sources of the requests, and we wanted a consistent information extraction over the multiple traces to eventually find correlations. Here are the rest of the parameters for the experiment :

- $\varepsilon = 0.001$ which means $k = 1000$ partitions
- $\delta = 0.001$ which means $t = 7$ hashing functions

In the figure 4 we can see the result of the experiment where the result of the algorithm is close to the original. Both chart are plotted with a logarithmic scale.

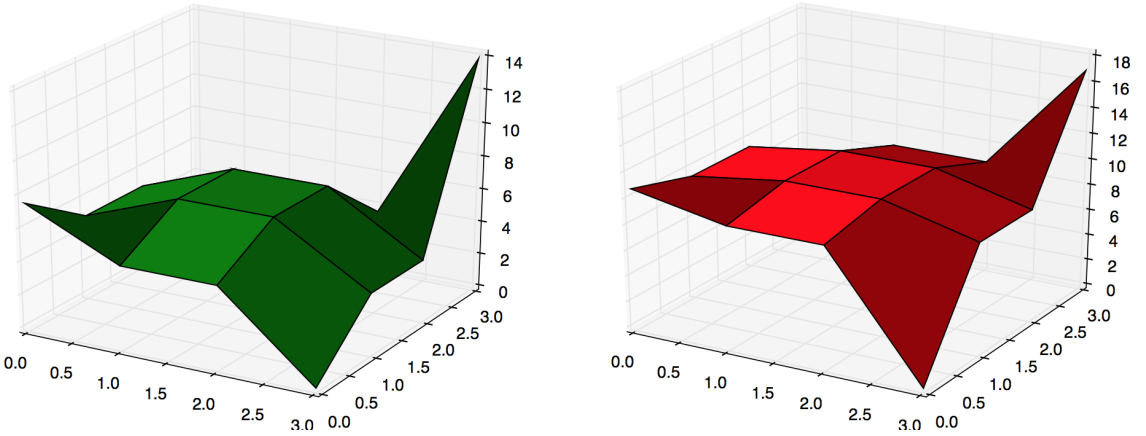

Figure 4: Codeviance matrix of real data traces:
real one (green) - calculated with `SketchMin` algorithm (red)

The red chart is not exactly the result given by the algorithm. In fact, it is the average result over 10 runs of the algorithm. It is important to make several runs, and to consider the average because we use hashing functions that are randomly generated. The standard deviation of this experiment is given by the following chart (which was not plotted in logarithmic scale).

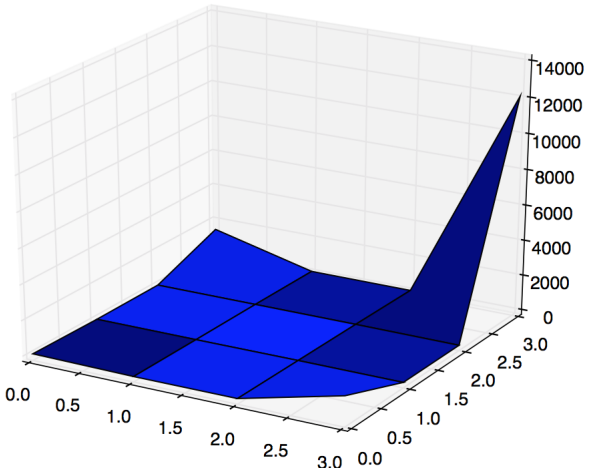

Figure 5: Standard deviation for 10 calculations of the `SketchMin` algorithm

3.2 Generated data traces

We generated artificial data traces using some probabilistic laws. These traces were generated using the package `numpy.random` from the Python library. Here are the parameters of this experiment, and the list of the traces we generated :

- size of the traces : $size = 10000$
- integers values are generated between 0 and $u = 100$
- $\varepsilon = 0.1$ which means $k = 10$ partitions
- $\delta = 0.001$ which means $t = 7$ hashing functions

	Probabilistic laws	Parameters
Trace 0	Uniform	
Trace 1	Zipfian	$\alpha = 2$
Trace 2	Zipfian	$\alpha = 3$
Trace 3	Zipfian	$\alpha = 4$
Trace 4	Zipfian	$\alpha = 5$
Trace 5	Zipfian	$\alpha = 6$
Trace 6	Poisson	$\lambda = u/(2)$
Trace 7	Poisson	$\lambda = u/(2^2)$
Trace 8	Poisson	$\lambda = u/(2^3)$
Trace 9	Poisson	$\lambda = u/(2^4)$
Trace 10	Poisson	$\lambda = u/(2^5)$
Trace 11	Binomial	$p = 0.42$
Trace 12	Negative Binomial	$p = 0.42$

We calculated the real codeviance matrix, and we applied the `SketchMin` algorithm. Here are the results we got in figure 6 below. We can notice that the two shapes are really similar, though the scales are different.

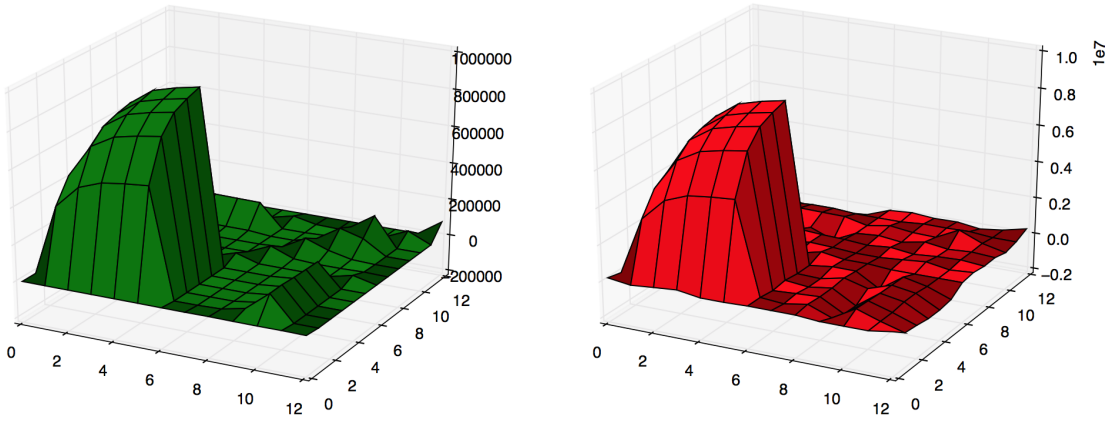

Figure 6: Codeviance matrix of generated data traces:
real one (green) - calculated with `SketchMin` algorithm (red)

The red chart is also an average over 10 runs, here is the standard deviation of this experiment :

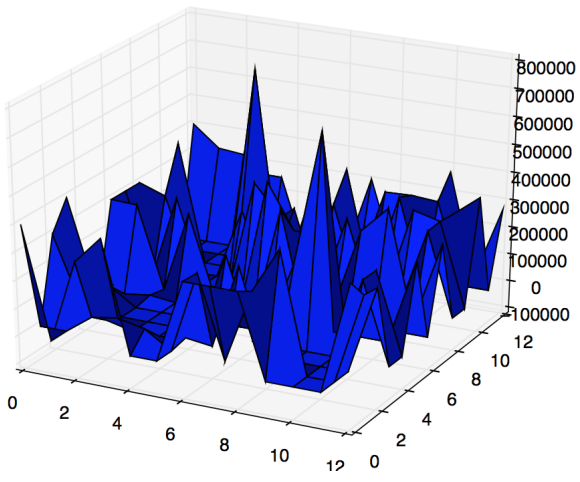

Figure 7: Standard deviation for 10 calculations of the `SketchMin` algorithm

Conclusion

In the end the results of the algorithm seem to match the exact entries, with a difference of scale. That's why the `Sketchmin` algorithm seems to be an efficient way to calculate the codeviance of huge-sized data traces.

However, our program isn't applicable to a real application because we were here only considering a static data stream. But in reality, we have to apply this to moving streams, and so to determine when to drop data. Considering several-months-old data does not seem very wise. The next step would be to deal with this. Several algorithms exist, using buffers.

It is also possible to improve the time complexity of our program. We used only sequential programming, but it would be better to use parallel programming, especially because the computations related to a hashing function are independent from those related to another hashing function.

Finally, we programmed a procedure to calculate the codeviance matrix of several data streams. But, in order to use this information, it is necessary to make empirical learning, so as to become able to tell when an attack is happening, according to the values calculated.

References

- E. Anceaume, Y. Busnel, "Deviation Estimation between Distributed Data Streams", 10th European Dependable Computing Conference (EDCC 2014), May 2014, Newcastle, United Kingdom. pp.35-45, 2014
- Wikipedia, "Universal hashing"