650.060 - Small Project in Artificial Intelligence and Cybersecurity

An API for Mutual Information estimation

Francesco Bombassei De Bona Summer Term 2022/2023 12138677

1 Introduction

Mutual Information (MI) plays a pivotal role in the realms of artificial intelligence and cybersecurity. It serves as a measure to quantify the amount of information shared between two variables. In the domain of cybersecurity, especially in side-channel attacks, an accurate estimation of MI can be the difference between a successful breach and a failed attempt. However, the current methodologies for estimating MI, especially in mixed cases, present challenges in terms of accuracy and computational efficiency. This project aims to address these challenges by introducing a publicly available C++ API that employs the methodologies presented in [1]. Furthermore, we provide a comparative analysis of our proposed implementation with the traditional histogram estimator method.

2 Background and terminology

2.1 Mutual Information

Mutual Information (MI) is a measure of the statistical dependence between two random variables. In the context of side-channel analysis, MI is used to quantify the dependence between an observed leakage and a secret key. Higher values of MI indicate a stronger dependence between the two variables. Thus, an higher MI value leads to a higher leakage of information about the secret key by observing a side-channel trace.

The MI between two random variables X and Y is defined as:

$$\begin{aligned} I(X;Y) &= H(X) - H(X|Y) \quad (1)\\ &= H(X) + H(Y) - H(X,Y) \quad (2) \end{aligned}$$

where $H(X)$ is the entropy of X, $H(X|Y)$ is the conditional entropy of X given Y and is defined as

$$H(X|Y) = \begin{cases} \sum_y p(y)H(X|Y=y) & \text{if } Y \text{ is discrete} \\ \int_y p(y)H(X|Y=y)dy & \text{if } Y \text{ is continuous} \end{cases}$$

and $H(X,Y)$ is the joint entropy of X and Y.

As presented in [1], it's possible to observe three different cases of MI:

- X and Y discrete that leads to the discrete MI;

- X and Y continuous that leads to the continuous MI;
- X discrete and Y continuous that leads to the mixed MI.

In this work, we focus on the mixed MI case.

For this case there are two different ways to compute the MI between X and Y: using an estimator for the conditional/joint density in combination with the H2 or H3 formulas; or using an estimator based on nearest neighbors search.

Side-channel analysis is a technique where attackers try to find a correlation between information leaked from physical implementations of cryptographic algorithms (like power consumption or electromagnetic emissions) and the secret key used during the cryptographic operations. MI, in this context, quantifies the amount of information about the secret key that can be inferred from these leakages.

The three different cases of MI—discrete-discrete, continuous-continuous, and discrete-continuous—have their unique applications. For instance, discrete-discrete MI might be used in analyzing categorical data, while continuous-continuous could be applied in regression analysis. The mixed MI, which is the focus of our work, is particularly challenging due to the inherent complexities in handling both discrete and continuous data types.

2.2 Histogram estimator

The histogram estimator is a method for estimating the MI between two random variables X and Y. The core idea behind the histogram estimator is to discretize the continuous random variables into bins and then compute the mutual information based on the joint and marginal frequencies of these bins. The MI between X and Y is then estimated using the H2 equation as:

$$I(X;Y) = H(Y) - H(Y|X) \tag{3}$$

where $H(Y)$ is the entropy of Y and $H(Y|X)$ is the conditional entropy of Y given X.

2.3 GKOV estimator

The GKOV estimator is a method for estimating the MI between two random variables X and Y introduced by Gao, Krishnan, Oh and Vishwanath [4]. This estimator can be used on a combination of discrete and continuous random variables. The GKOV estimator is defined as:

$$I_n(X;Y) = \frac{1}{n}\sum_{i=1}^{n} \hat{I}_i = \sum_{i=1}^{n} \big(\psi(\tilde{t}_i) + \log n - \log(n_{x,i}+1) - \log(n_{y,i}+1)\big) \tag{4}$$

where ψ is the digamma function.

3 Project Execution

The project aimed to develop a C++ API for the GKOV estimator, the histogram estimator, and a simulator for the leakage of a device. All the code was written in C++ and using an OO approach.

In the 'Simulator' subsection, our chosen leakage model is reflective of many real-world scenarios where a device's physical leakages (like power consumption) can be modeled as a function of its cryptographic operations, added with some noise. This noise can arise from various sources, including environmental factors or inherent device fluctuations.

3.1 GKOV estimator

The GKOV estimator was implemented as described in [4]. The implementation is based on the class `GKOVEstimator` which is initialized with the callback function to compute the value t_n as described in [1]. The class `GKOVEstimator` provides the method `estimate` which takes as input the discrete variable X, the continuous variable Y, and the dimension of the two variables. The method `estimate` returns the estimated MI between X and Y. Under the hood, the method `estimate` computes the value t, builds the matrix data starting from the discrete and continuous variables, builds the search trees for the two single variables and the combined matrix, and finally computes the GKOV estimator as described in [1].

To perform the nearest neighbors search, the project makes usage of the MLPack library [2]. The class `GKOVEstimator` makes usage of the class `mlpack::neighbor::NeighborSearch` to perform the nearest neighbors search over the combined matrix built from the discrete and continuous variables. While to perform the computation of lines 4, 8, and 9 of Algorithm 1 in [1], the class `GKOVEstimator` makes usage of the class `mlpack::neighbor::RangeSearch` to perform the distance search over the combined variables and the single variables. In particular, the tree used for both types of search is the `mlpack::tree::BallTree`.

3.2 Histogram estimator

The histogram estimator was implemented to be used as a baseline for comparing the GKOV estimator. Following the description in Section 2.2, the histogram estimator was implemented as a class `HistogramEstimator`. The class `HistogramEstimator` is initialized with the number of dimensions of the continuous variable Y, the number of bins for each dimension, and the ranges of the bins. The underlying assumption about the ranges is that the bins are equally spaced. The class `HistogramEstimator` provides the method `estimate` which takes as input the discrete variable X, its probability distribution, the continuous variable Y, the size of the two variables and the dimension of the Y variable.

The method `estimate` makes usage of the methods `build_histogram`, `pdf_entropy`, and `conditional_entropy` to compute the MI between X and Y. `build_histogram` builds histogram of Y based on the dimensions of the variable and computes the pdf of the histogram. To ensure the efficiency of the histogram estimator, the method makes usage of the GSL library [3] for building the 1D and 2D histograms. The method `pdf_entropy` computes the entropy of the pdf of Y. The method `conditional_entropy` computes the conditional entropy of Y given X. Finally, the method `estimate` returns the MI between X and Y computed as described in equation 3.

3.3 Simulator

The simulator was implemented to provide a way to test the GKOV estimator and the histogram estimator. This class has the purpose of simulating the leakage of a device using the following model:

$$\text{leakage} = \text{leakage\_function}(\text{crypto\_function}(key, \text{plaintext})) + \text{noise} \tag{5}$$

where `leakage_function` is the function that simulates the leakage of the device, `crypto_function` is the function that simulates the cryptographic function of the device, and `noise` is the noise added to the leakage. At time of writing, the simulator supports gaussian and laplacian noise distributions but the code was made to be easily extensible to other distributions. Instead, the cryptographic function is passed as argument of the methods of the simulator.

Regarding the type of trace that can be simulated, the simulator supports one dimensional traces, but new can be easily added by extending the main class.

Finally, the simulator saves the simulated traces in a HDF5 file to allow the user to use the traces for further analysis.

3.4 Utilities

The project provides a class that implements some utilities used by the other classes in the project.

In particular, the class `Utils` provides the methods:

- `flatten` and `to_gkov_format` to convert an n-dimensional array to, respectively, a 1-dimensional array and a 2-dimensional array;
- `compute_distribution` to compute the distribution of a discrete variable;
- `read_traces` and `write_traces` to read and write traces from and to a HDF5 file.

4 Project Evaluation

The project was evaluated by comparing the GKOV estimator with the histogram estimator.

4.1 Simulation of the traces

The first phase of the evaluation was to simulate the traces of a device. The `Simulator` class was used to achieve this goal.

The simulation was done for an increasing length of the traces, from 10 to 163840 samples. The traces were simulated using the AES-128 algorithm with a random key, a gaussian noise with mean 0 and standard deviation 1, and a leakage function that computes the Hamming weight of the intermediate value of the AES-128 algorithm. The traces were saved in a HDF5 file to allow the user to use the traces for further analysis along with the secret key saved as attribute of the HDF5 file.

4.2 Setup of the experiments

The second phase of the evaluation was to set up the experiments. From a first run of a potential attack, it was notice that the computation of the GKOV estimator was computationally expensive. To improve the performance of the GKOV estimator, the usage of the Python library `ray` [5] was considered.

The idea was to use the Python library `ray` to parallelize the computation of the GKOV estimator across a cluster of machines. This library is focused on the parallelization of AI tasks, but it can be used for any kind of parallelization.

The parallelization of the attack was done by generating a list of tasks, where each task is the computation of the MI between a single trace and one of the 256 possible values of the secret key. The list of task was then split across the machines of the cluster. This procedure was repeated for each trace generated in the previous phase.

AWS was chosen as platform for the deployment of the cluster since it is natively supported by `ray`. The machines of the cluster were configured as follows:

- 1 head node of the type `c6a.xlarge` with 4 vCPUs and 8 GB of RAM, responsible for the coordination of the cluster and saving the results of the computation;
- 30 worker nodes of the type `c6a.large` with 2 vCPUs and 4 GB of RAM, responsible for the computation of MI values using both the GKOV estimator and the histogram estimator.

All the vCPUs of the machines are based on 3rd generation AMD EPYC processors with a turbo clock speed of 3.6 GHz.

4.3 Attack

The third phase of the evaluation was to run the attack.

The attack was implemented as described in Algorithm 1 with the following parameters:

- `t` was set to be $\log_{10}$

Algorithm 1 Attack

```
Require: Filename, Key
Ensure: Estimated MI values
1: $traces \leftarrow$ read_traces($Filename$)
2: $Y\_gkov \leftarrow$ to_gkov_format($traces$)    ▷ Convert traces to GKOV format
3: $Y\_hist \leftarrow$ flatten($traces$)    ▷ Convert traces to histogram format
4: $gkov\_estimator \leftarrow$ GKOVEstimator($t$)
5: $hist\_estimator \leftarrow$ HistogramEstimator($dims, bins, range$)
6: $X \leftarrow$ hamming_weight(aes_intermidiate($traces, Key$))    ▷ Compute the discrete variable
7: $gkov\_estimates \leftarrow gkov\_estimator.estimate(X, Y\_gkov, sizeOfX, sizeOfY)$    ▷ Compute the GKOV estimates
8: $hist\_estimates \leftarrow hist\_estimator.estimate(X, pX, Y\_hist, size, dims)$    ▷ Compute the histogram estimates
```

- `dims` was set to 1;
- `bins` was set to 10;
- `range` was set to $[\min Y, \max Y]$;
- `pX` was set to the distribution of 1s in the intermediate value of the AES-128 algorithm;

4.4 Results

After running the attack, the results were collected from the cluster and analyzed. The results were collected in a JSON file using the following format:

```
{
    key : {
        "gkov": value,
        "hist": value
    }
}
```

The results are presented in the following table:

samples	key	gkov	gkov_mi	hist	hist_mi
10	235	20	-1.426090	32	2.05532
20	233	244	-1.225250	176	1.58543
40	17	104	-1.084880	167	0.518031
80	140	140	0.086083	86	-3.3962
160	115	115	0.220301	97	-9.97996
320	55	55	0.177586	55	-22.8171
640	107	107	0.246235	107	-50.9727
1280	131	131	0.369563	222	-99.1721
2560	252	252	0.368122	252	-229.192
5120	12	12	0.365723	12	-450.422
10240	238	238	0.422796	238	-900.454
20480	186	186	0.404086	186	-1847.61
40960	120	120	0.412238	168	-3492.770000
81920	114	114	0.407635	15	-7112.370000
163840	179	179	0.440852	206	-14527.300000

First, it can be observed that the GKOV estimator is more accurate than the histogram estimator. In fact, the GKOV estimator is able to recover the correct key for all the traces if the number of samples is greater than 80, while the histogram estimator is able to recover the correct key only for 6 traces over the 15 traces. It is unclear why the histogram estimator is not able to recover the correct key for all the traces, but it is possible that the parameters of the histogram estimator were not chosen correctly. In fact, the choice of the correct binning size is not trivial and it is possible that the binning size chosen for the histogram estimator is not the correct one.

Second, it can be observed, by plotting the MI values of the GKOV estimator, that these values seem to converge to a value between 0.4 and 0.5, as shown in Figure 1. This is in line with the results presented in [1] where it is shown that the GKOV estimator converge to the true MI value as the number of samples increases.

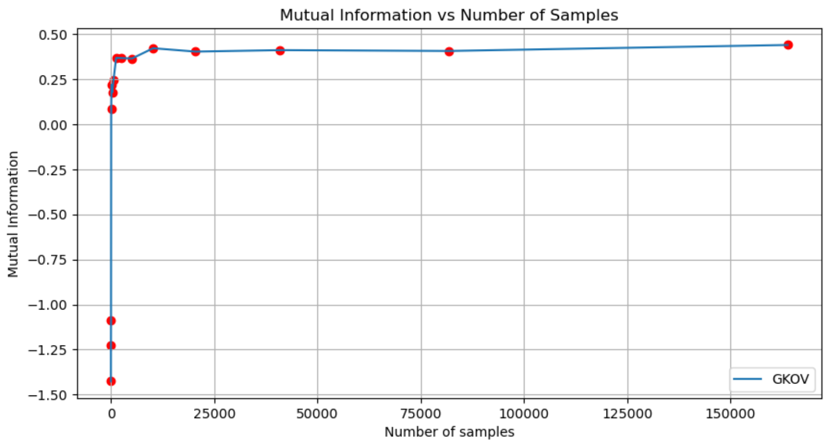

Figure 1: GKOV MI values

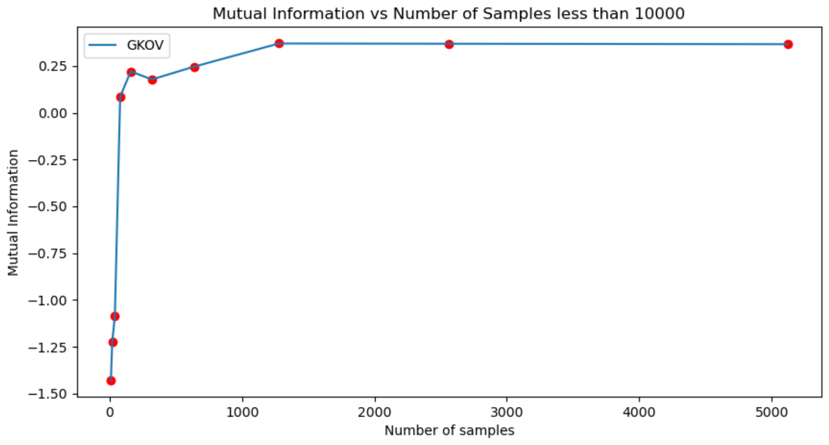

Figure 2: GKOV MI values for less than 10000 samples

5 Conclusion

Our project successfully introduces a C++ API for estimating Mutual Information, with a particular focus on the mixed MI case. The GKOV estimator, as demonstrated, offers a more accurate and reliable estimation compared to the traditional histogram estimator. This advancement holds significant implications for the field of cybersecurity, especially in the realm of side-channel attacks. Future work could delve into optimizing the histogram estimator's parameters or exploring other estimation methodologies to further enhance accuracy and computational efficiency.

References

[1] Chowdhury, A., Roy, A., Brunetta, C., and Oswald, E. Leakage Certification Made Simple, 2022. Publication info: Preprint.

[2] Curtin, R. R., Edel, M., Shrit, O., Agrawal, S., Basak, S., Balamuta, J. J., Birmingham, R., Dutt, K., Eddelbuettel, D., Garg, R., Jaiswal, S., Kaushik, A., Kim, S., Mukherjee, A., Sai, N. G., Sharma, N., Parihar, Y. S., Swain, R., and Sanderson, C. mlpack 4: a fast, header-only c++ machine learning library. *Journal of Open Source Software 8*, 82 (2023), 5026.

[3] Galassi, M., Ed. *GNU scientific library reference manual: for GSL version 1.12*, 3. ed ed. Network Theory, Bristol, 2009.

[4] Gao, W., Kannan, S., Oh, S., and Viswanath, P. Estimating Mutual Information for Discrete-Continuous Mixtures.

[5] Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., and Stoica, I. Ray: A distributed framework for emerging ai applications, 2018.