Package ‘sideChannelAttack’

February 20, 2015

Version 1.0-6
Date 2013-04-12
Title Side Channel Attack
Author Liran Lerman <l1erman@ulb.ac.be>, Gianluca Bontempi
<gbonte@ulb.ac.be>, Olivier Markowitch
<olivier.markowitch@ulb.ac.be>
Maintainer Liran Lerman <l1erman@ulb.ac.be>
Depends R (>= 2.10), MASS, corpcor, mmap, ade4, infotheo
Description This package has many purposes: first, it gives to the
community an R implementation of each known side channel attack
and countermeasures as well as data to test it, second it
allows to implement a side channel attack quickly and easily.
License GPL-3
URL http://student.ulb.ac.be/~l1erman/
NeedsCompilation no
Repository CRAN
Date/Publication 2013-04-12 17:46:02

R topics documented:

sideChannelAttack-package ........................................ 2
dpa ................................................................. 3
filter.MAX ....................................................... 5
filter.mRMR ...................................................... 6
filter.NULL ...................................................... 7
filter.PCA ....................................................... 8
filter.RegressionTreeFilter .................................. 9
gaussian ......................................................... 10
plot.verify.cv ................................................. 12
plot.verify.ho ................................................ 13
plot.verify.loo ............................................... 14
**Side Channel Attack**

**Description**

Most embedded devices require security and privacy protection. Cryptography algorithms and protocols are designed with the aim to provide such security. However, cryptanalysis attacks based on physical measures (such as the device power consumption named trace) realized on secure devices can be used to challenge their actual security. These attacks called side channel attacks and consist in retrieving secret data by observing physical properties of the device. Note that each observed physical properties are described with a vector of real values and each component of this vector is seen as a variable.

Side channel attacks are probably among the most dangerous attacks in cryptanalysis. However, we observe that many scientific papers presenting new side channel attack do not give any implementation of their attacks neither any data would allowed to verify the claimed results. In other side, when a researcher need to realize an attack, he/she has to implement it. This package has many purposes: first, it gives to the community an R implementation of each known attack/countermeasure as well as data to test it, second it allows to implement a side channel attack quickly and easily.

**Details**

This package is divided in 4 blocks.

The first block is the **simulator** that allows to simulate a cryptographic device without any knowledge of VHDL, FPGA, etc. On the base of a key and messages, it provides a physical measure such as power consumption. Its goal is to facilitate and accelerate the collect of data, to compare attacks with the same dataset and to implement countermeasures. Note that simulator can be categorized into several groups depending on their level of abstraction: at the gate level, at the instruction level and the function level, etc. And note that you can obtain traces through `powerC` given in this package.

The second block is the **filter**, a feature selection, intended to reduce the number of points per trace and therefore to accelerate the attack.

The third is the **model** (an attack) which returns the key used by the cryptographic device knowing a trace.

The last block is the **verify** function who estimates the quality of the attacks and the corresponding countermeasures. It can be based on the execution time of the attack, the amount of memory used, the number of power consumption traces required or only by the number of times the model predicts correctly the key.
**dpa1**

**Author(s)**

Liran Lerman & Gianluca Bontempi & Olivier Markowitch

**References**


**Examples**

```r
#data collection
data(powerC)
traces = powerC[, -301]
traces = traces[, 1:100]
key = powerC[, 301]
newindice = sample(1:256)
traces = traces[newindice,]
key = key[newindice] + 1

#model checking
attack=verify.ho(model=gaussian, filter=filter.PCA, Xlearn=traces[1:128,], Ylearn=key[1:128], Xval=traces[129:256,], Yval=key[129:256])
plot(attack)
```

<table>
<thead>
<tr>
<th><strong>dpa1</strong></th>
<th><strong>DPA</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td></td>
</tr>
<tr>
<td>The <code>dpa1</code> function applies a Differential Power Analysis (DPA) to a set of traces in order to find the key used by the cryptographic device.</td>
<td></td>
</tr>
<tr>
<td><strong>Usage</strong></td>
<td></td>
</tr>
<tr>
<td><code>dpa1(x, y, ...)</code></td>
<td></td>
</tr>
</tbody>
</table>
Arguments

x A matrix where each row is a trace.
y A vector where the \( i^{th} \) element of the vector \( y \) is the key for the \( i^{th} \) trace in the matrix \( x \).

Details

The \textit{dpa1} function is an example of DPA. It calculates the average \( X_i \) of traces for each key \( Y_i \). Then, to estimate the key from a trace \( T \), it returns the key \( Y_i \) which maximizes the equation \( \arg \max_{Y_i}(\text{cor}(T, X_i)) \).

Value

The \textit{dpa1} function returns an object which knowing a trace can be used with the \textit{predict} function that estimates the value of the key. The value of this function is an object of class \textit{dpa1}, which is a list with the following components:

- mean a list of arithmetics means, one for each possible key

Author(s)

Liran Lerman <lberman@ulb.ac.be> & Gianluca Bontempi <gbonte@ulb.ac.be@ulb.ac.be> & Olivier Markowitch <olivier.markowitch@ulb.ac.be>

References


Examples

```r
# data collection
data(powerC)
traces = powerC[-301]
key = powerC[301]+1

# model creation
attack=dpa1(traces[-1,],factor(key[-1]))

# model prediction
predict(attack, traces[1,])
```
**Description**

The `filter.MAX` function returns the maximum values of a physical measure.

**Usage**

```r
filter.MAX(nbreVarX_, ...) 
```

**Arguments**

- `nbreVarX_`  
  The number of variables which represents each physical measures after the reduction.

- `...`  
  Currently ignored.

**Details**

The `filter.MAX` function is the feature selection MAX. It converts a set of physical measures to another one with less components.

**Value**

The `filter.MAX` function returns an object which can be used with the `predict` function to reduce each physical measure. This physical measure can be the same or another one than contained in `X`.

The value of this function is an object of class `filter.MAX`, which is a list with the following components:

- `nbreVarX`  
  number of component to get after the reduction of a physical measure.

**Author(s)**

Liran Lerman <llerman@ulb.ac.be> & Gianluca Bontempi <gbonte@ulb.ac.be> & Olivier Markowitch <olivier.markowitch@ulb.ac.be>

**Examples**

```r
#data collection
data(powerC)
traces = powerC[,301]
key = powerC[,301]

#model creation
attack=filter.MAX(nbreVarX_=2)

#model prediction
predict(attack,t(traces[1,]))
```
Description

The filter.mRMR function applies the feature selection mRMR to a set of physical measures.

Usage

filter.mRMR(X, Y, nbreVarX__, ...)

Arguments

- **X**: A matrix where each row is a physical measure.
- **Y**: A vector where the $i^{th}$ element of the vector $y$ is the key for the $i^{th}$ physical measure in the matrix $x$.
- **nbreVarX__**: Number of component to get after the reduction by the mRMR of a physical measure.
- **...**: Currently ignored.

Details

The filter.mRMR function is the feature selection mRMR. It returns an object which can be used with the predict function to convert a set of physical measures to another one with less variables.

Value

The filter.mRMR function returns an object which can use with the predict function to reduce each physical measure. This physical measure can be the same or an other one than contained in $X$.

The value of this function is an object of class filter.mRMR, which is a list with the following components:

- **filter**: sorted list of the best variables returned by the mRMR algorithm.

Author(s)

Liran Lerman <lberman@ulb.ac.be> & Gianluca Bontempi <gbonte@ulb.ac.be> & Olivier Markowitch <olivier.markowitch@ulb.ac.be>

References

filter.NULL

Examples

```r
# data collection
data(powerC)
traces = powerC[, -301]
key = powerC[, 301]

# model creation
attack = filter.mRMR(X = traces[-1, 1:10], Y = key[-1], nbreVarX = 2)

# model prediction
predict(attack, t(traces[1, ]))
```

filter.NULL filter:NULL

Description

The `filter.NULL` function applies the feature selection NULL to a set of physical measures. In other words, it returns all this data.

Usage

`filter.NULL(...)`

Arguments

... Currently ignored.

Details

The `filter.NULL` function is the feature selection NULL. It returns an object which can be used with the `predict` function to convert a set of physical measures to another one. But in this particular case, it returns all the data without reduction.

Value

The `filter.NULL` function returns an object which can be used with the `predict` function to reduce each physical measure. This data can be the same or an other one than contained in `X`.

The value of this function is an object of class `filter.NULL`, which is a list without element.

Author(s)

Liran Lerman <lberman@ulb.ac.be> & Gianluca Bontempi <gbonte@ulb.ac.be> & Olivier Markowitch <olivier.markowitch@ulb.ac.be>
**Examples**

```r
# data collection
data(powerC)
traces = powerC[, -301]
key = powerC[, 301]

# model creation
attack = filter.NNULL()

# model prediction
predict(attack, t(traces[1, ]))
```

**Description**

The `filter.PCA` function applies the feature selection Principal Component Analysis (PCA) to a set of physical measures.

**Usage**

```r
filter.PCA(X, nbreVarX, ...)
```

**Arguments**

- `X`  
  A matrix where each row is a physical measures.

- `nbreVarX`  
  The number of variables which represents each physical measures after the reduction by the PCA.

- `...`  
  Currently ignored.

**Details**

The `filter.PCA` function is the feature selection PCA. It converts a set of physical measures to another one with less components.

**Value**

The `filter.PCA` function returns an object which can be used with the `predict` function to reduce each physical measure. This physical measure can be the same or another one than contained in `X`. The value of this function is an object of class `filter.PCA`, which is a list with the following components:

- `mod`  
  a model of PCA.

- `nbreVarX`  
  number of component to get after the projection by the PCA of a physical measure.
Author(s)
Liran Lerman <lerman@ulb.ac.be> & Gianluca Bontempi <gbonte@ulb.ac.be> & Olivier Markowitch <olivier.markowitch@ulb.ac.be>

References

Examples

```r
# data collection
data(powerC)
traces = powerC[,301]
traces = traces[,1:100]
key = powerC[,301]

# model creation
attack=filter.PCA(X=traces[-1,],nbVarX_=2)

# model prediction
predict(attack,t(traces[1,]))
```

Description
The `filter.RegressionTreeFilter` function implements the feature selection RegressionTreeFilter to a set of physical measures.

Usage

`filter.RegressionTreeFilter(X,nbreVarX_,...)`

Arguments

- `X`  
  A matrix where each row is a physical measure.
- `nbreVarX_`  
  The number of variables which represents each physical measure after the reduction by the RegressionTreeFilter.
- `...`  
  Currently ignored.

Details
The `filter.RegressionTreeFilter` function is the feature selection RegressionTreeFilter. It returns an object which can be used with the `predict` function to convert a set of physical measures to another one with less variables.
The filter.RgressionTreeFilter function returns an object which can be used with the predict function to reduce each physical measure. This side channel can be the same or an other one than contained in $X$.

The value of this function is an object of class filter.RgressionTreeFilter, which is a list with the following components:

- `nbreVarX` number of variable to get after the projection in the new basis.

**Author(s)**

Liran Lerman <lorman@ulb.ac.be> & Gianluca Bontempi <gbonte@ulb.ac.be> & Olivier Markowitch <olivier.markowitch@ulb.ac.be>

**References**


**Examples**

```
# data collection
data(powerC)
traces = powerC[, -301]
key = powerC[, 301]

# model creation
attack = filter.RgressionTreeFilter(X = traces[-1,], nbreVarX = 2)

# model prediction
predict(attack, t(traces[1,]))
```

---

**gaussian**

*Template Attack*

**Description**

The `gaussian` function applies a template attack (TA) to a set of traces in order to find the key used by the cryptographic device.

**Usage**

`gaussian(x, y,...)`
Arguments

x  A matrix where each row is a trace.
y  A vector where the \(i^{th}\) element of the vector \(y\) is the key for the \(i^{th}\) trace in the matrix \(x\).

...  Currently ignored.

Details

The `gaussian` function is an example of TA. It estimates the conditional probability of the trace for each key and then returns the key which maximizes this probability. It extracts all possible informations available in each trace and is hence the strongest form of side channel attack possible in an information theoretic sense that relies on a parametric Gaussian estimation approach.

Value

The `gaussian` function returns an object which can be used with the `predict` function to estimate the value of the key knowing a trace.

The value of this function is an object of class `gaussian`, which is a list with the following components:

- `mean`  a list of arithmetic means, one for each possible key.
- `cov`    a list of covariance matrices, one for each possible key.
- `detCov` a list of determinants of each covariance matrix.

Author(s)

Liran Lerman <lberman@ulb.ac.be> & Gianluca Bontempi <gbonte@ulb.ac.be> & Olivier Markowitch <olivier.markowitch@ulb.ac.be>

References


Examples

```r
#data collection
data(powerC)
traces = powerC[,1:301]
traces = traces[,1:100]
key = powerC[,301]+1

#feature selection
filter = filter.PCA(X=traces,nbreVarX.=2)
traces = predict(filter,traces)

#model creation
attack=gaussian(traces,-1,],factor(key[-1]))
```
#model prediction
predict(attack,traces[1,])

plot.verify.cv

---

Description

Plot the quality of an attack of a `verify.cv` object.

Usage

```r
## S3 method for class 'verify.cv'
plot(x, ...)
```

Arguments

- `x` An objet of class `verify.cv`
- `...` Currently ignored.

Details

The `verify.cv` function estimates the quality of the attack with a k-cross-validation approach and this function plots it.

Value

This function plots the quality of an attack of a `verify.cv` object.

Author(s)

Liran Lerman <llerman@ulb.ac.be> & Gianluca Bontempi <gbonte@ulb.ac.be> & Olivier Markowitch <olivier.markowitch@ulb.ac.be>

Examples

```r
# data collection
data(powerC)
traces = powerC[,301]
traces = traces[,1:10]
key = powerC[,301]
newindice = c(sample(1:128,15),sample(129:256,15))
traces = traces[newindice,]
key = key[newindice]+1

# model checking
attack=verify.cv(model=dpa1,filter=filter.PCA,X=traces,Y=key,nbreVarX=c(2:4),k=3)
plot(attack)
```
Description
Plot the quality of an attack of a verify.ho object.

Usage
```r
## S3 method for class 'verify.ho'
plot(x, ...)
```

Arguments
- `x` : An objet of class 'verify.ho'.
- `...` : Currently ignored.

Details
The `verify.ho` function estimates the quality of the attack with a hold-out approach and this function plots it.

Value
This function plots the quality of an attack of a verify.ho object.

Author(s)
Liran Lerman <lerman@ulb.ac.be> & Gianluca Bontempi <gbonte@ulb.ac.be@ulb.ac.be> & Olivier Markowitch <olivier.markowitch@ulb.ac.be>

Examples
```r
#data collection
data(powerC)
traces = powerC[-301]
traces = traces[1:10]
key = powerC[301]
newindice = sample(1:256)
traces = traces[newindice,]
key = key[newindice]+1

#model checking
attack=verify.ho(model=gaussian,filter=filter.PCA,Xlearn=traces[1:128,],Ylearn=key[1:128],Xval=traces[129:256],
plot(attack)
```
Description

Plot the quality of an attack of a `verify.loo` object.

Usage

```r
## S3 method for class 'verify.loo'
plot(x, ...)  
```

Arguments

- `x`: An object of class `verify.loo`.
- `...`: Currently ignored.

Details

The `verify.loo` function estimates the quality of the attack with a hold-out approach and this function plots it.

Value

This function plots the quality of an attack of a `verify.loo` object.

Author(s)

Liran Lerman (<llerman@ulb.ac.be>) & Gianluca Bontempi (<gbonte@ulb.ac.be>) & Olivier Markowitch (<olivier.markowitch@ulb.ac.be>)

Examples

```r
# data collection
data(powerC)
traces = powerC[,1:301]
traces = traces[,1:10]
key = powerC[,301]
newindice = c(sample(1:128,15),sample(129:256,15))
traces = traces[newindice,]
key = key[newindice]+1

# model checking
attack=verify.loo(model=dpa1,filter=filter.mRMR,X=traces,Y=key,nbreVarX=c(2:3))
plot(attack)
```
Description

This is a set of traces from a cryptographic device Xilinx Spartan XC3s5000 running at a frequency around 33 MHz and collected by an oscilloscope Agile infiniium DSO80204B 2Ghz 40GSa/s. The cryptographic device implemented the bloc cipher 'Triple Data Encryption Algorithm' and the traces have been reduced with the feature selection mRMR in order to obtain only 300 components from 20001 initially.

Usage

data(powerC)

Format

`powerC` is a matrix with 256 rows and 301 columns. From the first to the 300th component of `powerC`, each row is a trace. The secret to retrieve by these traces is a bit of the secret key used by the cryptographic device. The value of this bit is given by the 301st component of `powerC`.

Author(s)

Liran Lerman <lberman@ulb.ac.be> & Gianluca Bontempi <gbonte@ulb.ac.be> & Olivier Markowitch <olivier.markowitch@ulb.ac.be>

Examples

```r
# data collection
data(powerC)
traces = powerC[, -301]
key = powerC[, 301] + 1

# model creation
attack = dpa1(traces[-1, ], factor(key[-1]))

# model prediction
predict(attack, traces[1, ])
```
**Description**

The `simulator.Simple1` function simulates an unprotected cryptographic device implementing a bloc cipher.

**Usage**

```scheme
simulator.Simple1(message, key, noise=0)
```

**Arguments**

- `message`: A matrix where each row is a binary plaintext of 6 components. Each component has the value 1 or 0.
- `key`: A vector representing the binary secret key of 6 components. Each component has 1 or 0.
- `noise`: A positive integer to represent noise on traces returned by this function. Higher is its value, more there are noise on data.

**Details**

The `simulator.Simple1` function allows to simulate a cryptographic device implementing a bloc cipher. Its goal is to facilitate and to accelerate the collect of data and to compare attacks with the same dataset. With a key and messages, it returns a trace which is the power consumption, in volts, during an encryption by the cryptographic device.

The algorithm, implemented in the `simulator.Simple1` function, is detailed below. First, it applies the `xor` function between a `message`, and the `key`. The result is inserted in a S-Box which is a nonlinear function that takes 6 components and returns 4 components.

Then it calculates the hamming distance between the result and the previous result of the S-Box or the value null when encrypting the first message.

The result of the hamming distance is a point in the trace.

**Value**

The `simulator.Simple1` function returns a trace which represents the power consumption, in volts, during an encryption by a cryptographic device.

**Author(s)**

Liran Lerman <lorman@ulb.ac.be> & Gianluca Bontempi <gbonte@ulb.ac.be@ulb.ac.be> & Olivier Markowitch <olivier.markowitch@ulb.ac.be>
References


Examples

```r
n=100
clee = c(round(runif(6)))
inp=matrix(round(runif(6*n)),ncol=6)

#creating a trace without noise through the simulator
traces=simulator.Simple1(inp,clee)
```

Description

The `verify.cv` function allows to estimate the quality of a model (an attack) with a k-cross-validation approach.

Usage

`verify.cv(model, filter, X, Y, nbreVarX, k, param.model=list(), param.fs=list(), ...)`

Arguments

- `model`: A model such as randomForest, gaussian, svm, etc.
- `filter`: A feature selection such as filter.PCA, filter.mRMR, etc.
- `X`: A matrix where each row is a physical measure.
- `Y`: A vector where the \(i^{th}\) element of the vector \(y\) is the key for the \(i^{th}\) physical measure in the matrix \(x\).
- `nbreVarX`: The number of variables which represents each physical measure after the reduction by the feature selection.
- `k`: The number of traces in the validation set.
- `param.model`: A list of parameters to insert into the model.
- `param.fs`: A list of parameters to insert into the feature selection algorithm.
- `...`: Currently ignored.

Details

The `verify.cv` function estimates the quality of the attack with a k-cross-validation approach. It is executed in \(N/k\) rounds. Each round uses \(N - k\) traces to learn a model and the remaining trace to assess the generalization accuracy of the model. This is repeated until every set of \(k\) traces has been used for testing purposes. The best model is the one that maximizes the value returned by k-cross-validation.
Value

The `verify.cv` function returns an object which can be used with the `plot` function to plot the quality of the model.

- TP: number of true positive
- TN: number of true negative
- FN: number of false negative
- FP: number of false positive
- dim: the number of variables which represents each physical measure after the reduction by the feature selection.

Author(s)

Liran Lerman <llerman@ulb.ac.be> & Gianluca Bontempi <gbonte@ulb.ac.be> & Olivier Markowitch <olivier.markowitch@ulb.ac.be>

Examples

```r
# data collection
data(powerC)
traces = powerC[,301]
traces = traces[,1:10]
key = powerC[,301]
newIndice = c(sample(1:128,15),sample(129:256,15))
traces = traces[newIndice,]
key = key[newIndice]+1

# model checking
attack=verify.cv(model=dpa1,filter=filter.PCA,X=traces,Y=key,nbreVarX=c(2:4),k=2)
plot(attack)
```

Description

The `verify.ho` function allows to estimate the quality of a model (an attack) with a hold-out approach.

Usage

```r
verify.ho(model, filter, Xlearn, Ylearn, Xval, Yval, nbreVarX, param.model=list(), param.fs=list(),
```
Arguments

- **model**: A model such as randomForest, gaussian, svm, etc.
- **filter**: A feature selection such as filter.PCA, filter.mRMR, etc.
- **Xlearn**: A matrix where each row is a physical measure for the training set.
- **Ylearn**: A vector where the \(i\)th element of the vector \(Y\text{learn}\) is the key for the \(i\)th physical measure in the matrix \(X\text{learn}\).
- **Xval**: A matrix where each row is a physical measure for the validation set.
- **Yval**: A vector where the \(i\)th element of the vector \(Y\text{val}\) is the key for the \(i\)th physical measure in the matrix \(X\text{val}\).
- **nbreVarX**: The number of variables which represents each physical measure after the reduction by the feature selection.
- **param.model**: A list of parameters to insert into the model.
- **param.fs**: A list of parameters to insert into the feature selection algorithm.
- **...**: Currently ignored.

Details

The `verify.ho` function estimates the quality of the attack with a hold-out approach. It cuts randomly the set of observations in two subsets. The first one is retained as the training and the second is used for the validation phase where the quality of the model is estimated by counting the number of time where model predicts correctly the key.

Value

The `verify.ho` function returns an object which can be used with the `plot` function to plot the quality of the model.

The value of this function is an object of class `verify.ho`, which is a list with the following components:

- **TP**: number of true positive
- **TN**: number of true negative
- **FN**: number of false negative
- **FP**: number of false positive
- **dim**: the number of variables which represents each physical measure after the reduction by the feature selection.

Author(s)

Liran Lerman <lberman@ulb.ac.be> & Gianluca Bontempi <gbonte@ulb.ac.be@ulb.ac.be> & Olivier Markowitch <olivier.markowitch@ulb.ac.be>
Examples

```r
# data collection
data(powerC)
traces = powerC[, -301]
traces = powerC[, 1:10]
key = powerC[, 301]
newindice = sample(1:256)
traces = traces[newindice,]
key = key[newindice] + 1

# model checking
attack = verify.loo(model = gaussian, filter = filter.PCA,
Xlearn = traces[1:128, ], Ylearn = key[1:128], Xval = traces[128:256, ],
Yval = key[128:256], ...)

plot(attack)
```

Description

The `verify.loo` function allows to estimate the quality of a model (an attack) with a leave-one-out approach.

Usage

```
verify.loo(model, filter, X, Y, nbreVarX, param.model=list(), param.fs=list(), ...)
```

Arguments

- `model`: A model such as randomForest, gaussian, svm, etc.
- `filter`: A feature selection such as filter.PCA, filter.mRMR, etc.
- `X`: A matrix where each row is a physical measure.
- `Y`: A vector where the  \( i^{th} \) element of the vector \( y \) is the key for the  \( i^{th} \) physical measure in the matrix \( x \).
- `nbreVarX`: The number of variables which represents each physical measure after the reduction by the feature selection.
- `param.model`: A list of parameters to insert into the model.
- `param.fs`: A list of parameters to insert into the feature selection algorithm.
- `...`: Currently ignored.

Details

The `verify.loo` function estimates the quality of the attack with a leave-one-out approach. It is executed in \( N \) rounds. Each round uses \( N - 1 \) traces to learn a model and the remaining trace to assess the generalization accuracy of the model. This is repeated until every trace has been used for testing purposes. The best model is the one that maximizes the value returned by leave-one-out.

The advantages of this estimation is its accuracy but the estimation process is expensive in a computational point of view.
verify.loo

Value

The verify.loo function returns an object which can be used with the plot function to plot the quality of the model.

TP number of true positive
TN number of true negative
FN number of false negative
FP number of false positive
dim the number of variables which represents each physical measure after the reduction by the feature selection.

Author(s)

Liran Lerman <llerman@ulb.ac.be> & Gianluca Bontempi <gbonte@ulb.ac.be> & Olivier Markowitch <olivier.markowitch@ulb.ac.be>

Examples

# data collection
data(powerC)
traces = powerC[-301]
traces = powerC[1:10]
key = powerC[301]
newindice = c(sample(1:128,15),sample(129:256,15))
traces = traces[newindice,]
key = key[newindice]+1

# model checking
attack=verify.loo(model=dpa1,filter=filter.PCA,X=traces,Y=key,nbreVarX=c(4:5))
plot(attack)
Index

dpa1, 3
filter.MAX, 5
filter.mRMR, 6
filter.NULL, 7
filter.PCA, 8
filter.RegressionTreeFilter, 9
gaussian, 10
plot.verify.cv, 12
plot.verify.ho, 13
plot.verify.loo, 14
powerC, 15
predict.dpa1(dpa1), 3
predict.filter.MAX(filter.MAX), 5
predict.filter.mRMR(filter.mRMR), 6
predict.filter.NULL(filter.NULL), 7
predict.filter.PCA(filter.PCA), 8
predict.filter.RegressionTreeFilter
(filter.RegressionTreeFilter), 9
predict.gaussian(gaussian), 10

sideChannelAttack
(sideChannelAttack-package), 2
sideChannelAttack-package, 2
simulator.Simple1, 16

verify.cv, 17
verify.ho, 18
verify.loo, 20