Cryptocode

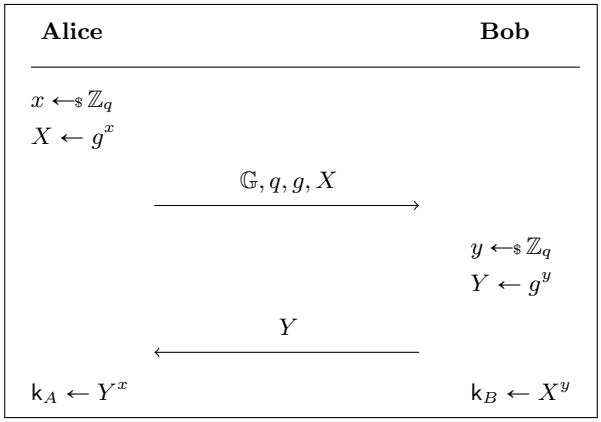
Typesetting Cryptography

Arno Mittelbach
mail@arno-mittelbach.de

March 28, 2015

Abstract

The cryptocode package is targeted at cryptographers typesetting their results in LaTeX. It provides various predefined commands for different topics in cryptography. In particular it provides an easy interface to write pseudocode, protocols, game based proofs and draw black-box reductions.



Contents

Chapter 1

Cryptocode by Example

Cryptocode is a LaTeXpackage to ease the writing of cryptographic papers. It provides mechanisms for writing pseudocode, protocols, game-based proofs and black-box reductions. In addition it comes with a large number of predefined commands. In this chapter we present the various features of cryptocode by giving small examples. But first, let's load the package

```
\usepackage[
  n,
  advantage,
  operators,
  sets,
  adversary,
  landau,
  probability,
  notions,
  logic,
  ff,
  mm,
  primitives,
  events,
  complexity,
  asymptotics,
  keys
  ]{cryptocode}
```

Note that all the options refer to a set of commands. That is, without any options cryptocode will provide the mechanisms for writing pseudocode, protocols, game-based proofs and black-box reductions but not define additional commands, such as \pk or \sk (for typesetting public and private/secret keys) which are part of the keys option. We discuss the various options and associated commands in Chapter 2.

1.1 Pseudocode

The cryptocode package tries to make writing pseudocode easy and enjoyable. The \pseudocode command takes a single parameter where you can start writing code in mathmode using \\ as line breaks. Following is an IND-CPA game definition using various commands from cryptocode to ease writing keys (\pk,\sk), sampling (\sample), and more:

$b \leftarrow\!\!\$\ \{0,1\}$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\!\$\ \mathsf{KGen}(1^n)$
$(\mathsf{state}, m_0, m_1) \leftarrow\!\!\$\ \mathcal{A}(1^n, \mathsf{pk}, c)$
$c \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{pk}, m_b)$
$b' \leftarrow\!\!\$\ \mathcal{A}(1^n, \mathsf{pk}, c, \mathsf{state})$
$\mathbf{return}\ b = b'$

The above code is generated by (the code is actually wrapped in an fbox).

```
\pseudocode[linenumbering,syntaxhighlight=auto]{%
  b \sample \bin  \\
  (\pk,\sk) \sample \kgen (\secparam)  \\
  (\state,m_0,m_1) \sample \adv(\secparam, \pk, c)  \\
  c \sample \enc(\pk,m_b)  \\
  b' \sample \adv(\secparam, \pk, c, \state)  \\
  return b = b' }
```

The pseudocode command thus takes a single mandatory argument (the code) plus an optional argument which allows you to specify options in a key=value fashion. In the above example we used the linenumbering option (which not surprisingly adds line numbers to the code) as well as the syntaxhighlighting option which highlights certain keywords (in the example it is responsible for setting "return" as **return**).

It is easy to define a heading for your code. Either specify the header using the option "head" or use the \procedure command which takes an additional argument to specify the headline.

$\text{IND-CPA}^{\mathcal{A}}_{\mathsf{Enc}}$

1 : $b \leftarrow_{\$} \{0,1\}$
2 : $(\mathsf{pk},\mathsf{sk}) \leftarrow_{\$} \mathsf{KGen}(1^n)$
3 : $(\mathsf{state}, m_0, m_1) \leftarrow_{\$} \mathcal{A}(1^n, \mathsf{pk}, c)$
4 : $c \leftarrow_{\$} \mathsf{Enc}(\mathsf{pk}, m_b)$
5 : $b' \leftarrow_{\$} \mathcal{A}(1^n, \mathsf{pk}, c, \mathsf{state})$
6 : **return** $b = b'$

```
\procedure[linenumbering]{$\indcpa_\enc^\adv$}{%
  b \sample \bin  \\
  (\pk,\sk) \sample \kgen (\secparam)  \\
  (\state,m_0,m_1) \sample \adv(\secparam, \pk, c)  \\
  c \sample \enc(\pk,m_b)  \\
  b' \sample \adv(\secparam, \pk, c, \state) \\
  \pcreturn b = b' }
```

Here in the example we have not turned on the automatic syntax highlighting but used the command \pcreturn to highlight the return statement. Besides \pcreturn there are a variant of predefined "keywords" such as \pcfor, \pcif, etc. (all prefixed with pc)

There is a lot more that we will discuss in detail in Chapter 3. Here, for example is the same code with an overlay explanation and a division of the pseudocode.

$\text{IND-CPA}^{\mathcal{A}}_{\mathsf{Enc}}$

1 : $b \leftarrow_{\$} \{0,1\}$
2 : $(\mathsf{pk},\mathsf{sk}) \leftarrow_{\$} \mathsf{KGen}(1^n)$

.....Setup Completed.....

3 : $(m_0, m_1) \leftarrow_{\$} \mathcal{A}(1^n, \mathsf{pk}, c)$
4 : $c \leftarrow_{\$} \mathsf{Enc}(\mathsf{pk}, m_b)$
5 : $b' \leftarrow_{\$} \mathcal{A}(1^n, \mathsf{pk}, c, \mathsf{state})$
6 : **return** $b = b'$

$\mathsf{KGen}(1^n)$ samples a public key pk and a private key sk.

```
\begin{pcimage}
\procedure[linenumbering]{$\indcpa_\enc^\adv$}{%
   b \sample \bin  \\
   (\pk,\sk) \sample \kgen (\secparam)\pcnode{kgen}  \pclb
  \pcintertext[dotted]{Setup Completed}
   (m_0,m_1) \sample \adv(\secparam, \pk, c)  \\
    c \sample \enc(\pk,m_b)  \\
```

```
    b' \sample \adv(\secparam, \pk, c, \state) \\
    \pcreturn b = b' }

\pcdraw{
  \node[rectangle callout,callout absolute pointer=(kgen),fill=orange]
      at ([shift={(+3,-1)}]kgen) {
    \begin{varwidth}{3cm}
      $\kgen(\secparam)$ samples a public key $\pk$ and a private key $\sk$.
    \end{varwidth}
  };
}
\end{pcimage}
```

1.2 Columns

The \pseudocode and \procedure commands allow the usage of multiple columns. You switch to a new column by inserting a \>. This is similar to using an align environment and placing a tabbing & character.[1]

First	**SecondThird**	**Fourth**
$b \leftarrow_{\$} \{0,1\}$	$b \leftarrow_{\$} \{0,1\} b \leftarrow_{\$} \{0,1\}$	$b \leftarrow_{\$} \{0,1\}$

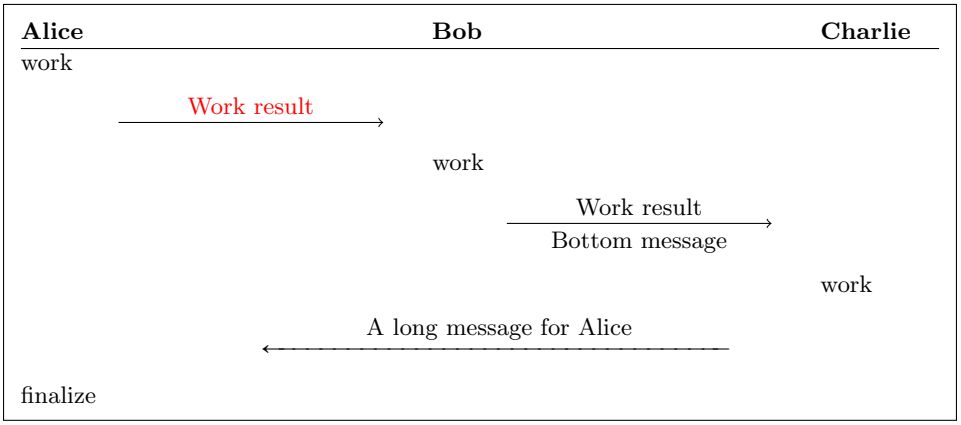
```
\pseudocode{%
  \textbf{First} \> \textbf{Second} \> \textbf{Third} \> \textbf{Fourth} \\
   b \sample \bin \> b \sample \bin \> b \sample \bin \> b \sample \bin}
```

As you can see the first column is left aligned the second right, the third left and so forth. In order to get only left aligned columns you could thus simply always skip a column by using \>\>. You can also use \< a shorthand for \>\>.

First	**Second**	**Third**	**Fourth**
$b \leftarrow_{\$} \{0,1\}$	$b \leftarrow_{\$} \{0,1\}$	$b \leftarrow_{\$} \{0,1\}$	$b \leftarrow_{\$} \{0,1\}$

1.3 Protocols

Using columns makes it easy to write even complex protocols. Following is a simple three party protocol




```
\pseudocode{%
  \textbf{Alice} \< \< \textbf{Bob} \< \< \textbf{Charlie} \\[][\hline]
  \text{work} \< \< \< \< \\
  \< \sendmessageright{top=Work result,topstyle=red} \< \< \< \\
```

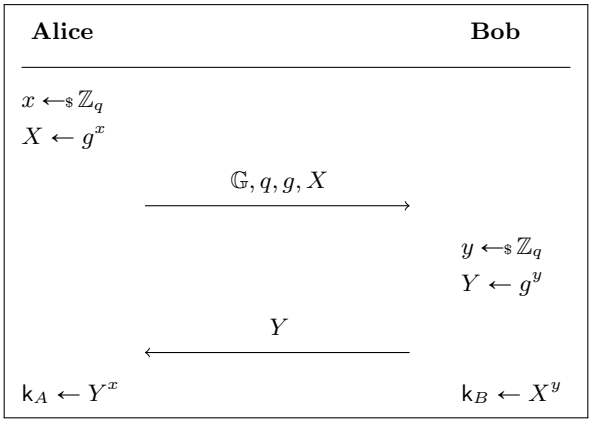
[1] In fact, the *pseudocode* command is based on amsmath's flalign environment.

```
 \< \< \text{work} \< \< \\
 \< \< \<  \sendmessageright{top=Work result,bottom=Bottom message} \< \\
 \< \<  \< \< \text{work} \\
 \< \sendmessageleftx{8}{\text{A long message for Alice}} \< \\
 \text{finalize} \< \< \< \< }
```

The commands \sendmessageright and \sendmessageleft are very flexible and allow to style the sending of messages in various ways. Also note the \\[][\hline] at the end of the first line. Here the first optional argument allows us to specify the lineheight (similarly to the behavior in an align environment). The second optional argument allows us to, for example, draw a horizontal line.

In multi player protocols such as the one above the commands \sendmessagerightx and \sendmessageleftx (note the x at the end) allow to send messages over multiple columns. In the example, as we were using \< the final message thus spans 8 columns.

For basic protocols you might also utilize the \sendmessageright* and \sendmessageleft* commands which simply take a message which is displayed.




```
\pseudocode{%
\textbf{ Alice} \< \< \textbf{ Bob}   \\[0.5\baselineskip][\hline]
 \<\< \\[-0.5\baselineskip]
x \sample \ZZ_q \< \< \\
 X \gets g^x \<\< \\
 \< \sendmessageright*{\GG,q,g,X} \< \\
 \<\< y \sample \ZZ_q \\
 \<\< Y \gets g^y \\
 \< \sendmessageleft*{Y} \< \\
 \key_A \gets Y^x \<\< \key_B \gets X^y }
```

We will discuss protocols in greater detail in Chapter 5.

1.4 Game-based Proofs

Cryptocode supports authors in visualizing game-based proofs. It defines an environment gameproof which allows to wrap a number of game procedures displaying helpful information as to what changes from game to game, and to what each step is reduced.

some hint

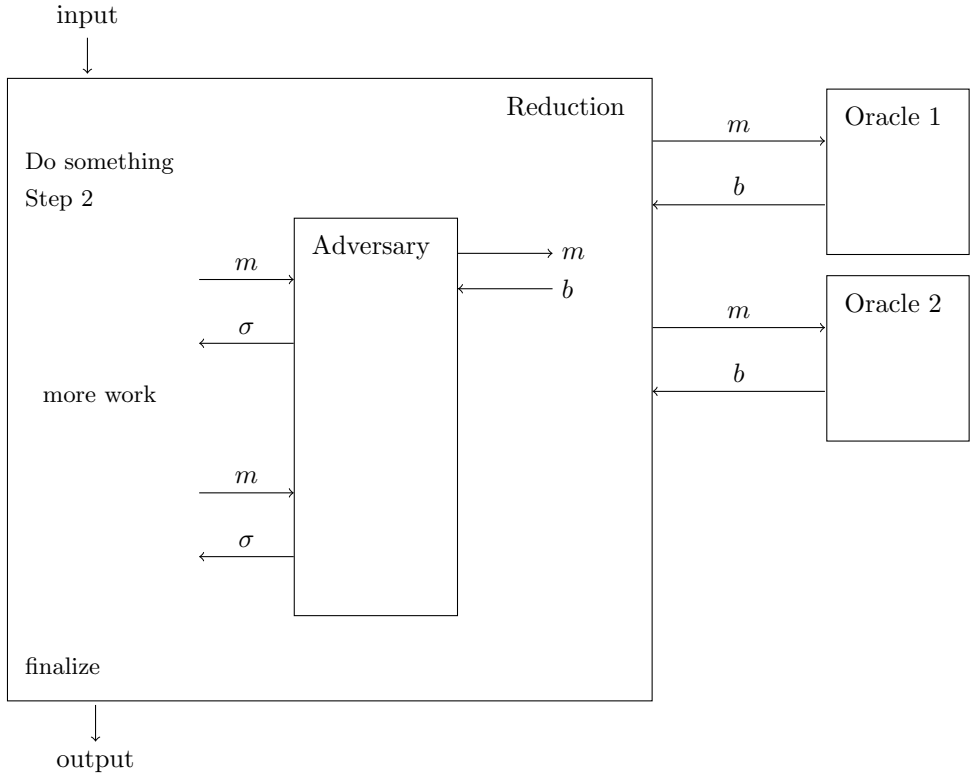
	$\mathsf{Game}_1(n)$	$\mathsf{Game}_2(n)$
1 :	Step 1	Step 1
2 :	Step 2	Step 2 is different
3 :	Step 3	Step 3

```
\begin{gameproof}
\gameprocedure[linenumbering,mode=text]{%
  Step 1 \\
  Step 2 \\
  Step 3
  }
\gameprocedure[mode=text]{%
    Step 1  \\
    \gamechange{Step 2 is different} \\
    Step 3
  }
\addgamehop{1}{2}{hint={\footnotesize some hint}}
\end{gameproof}
```

Note that we made use of the option "mode=text" in the above example which tells the underlying pseudocode command to not work in math mode but in plain text mode. We'll discuss how to visualize game-based proofs in Chapter 6.

1.5 Black-box Reductions

Cryptocode provides a strucured syntax to visualize black-box reductions. Basically cryptocode provides an environment to draw boxes that may have oracles and that can be communicated with. Cryptocode makes heavy use of TIKZ (`https://www.ctan.org/pkg/pgf`) for this, which gives you quite some control over how things should look like. Additionally, as you can specify node names (for example the outer box in the next example is called "A") you can easily extend the pictures by using plain TIKZ commands.




```
\begin{bbrenv}{A}
  \begin{bbrbox}[name=Reduction]
  \pseudocode{
    \text{Do something} \\
    \text{Step 2}
  }

  \begin{bbrenv}{B}
    \begin{bbrbox}[name=Adversary,minheight=4cm]
    \end{bbrbox}

```

```
      \bbrmsgto{top=$m$}
      \bbrmsgfrom{top=$\sigma$}
      \bbrmsgtxt{\pseudocode{%
        \text{more work}
      }}
      \bbrmsgto{top=$m$}
      \bbrmsgfrom{top=$\sigma$}

      \bbrqryto{side=$m$}
      \bbrqryfrom{side=$b$}
    \end{bbrenv}

    \pseudocode{
      \text{finalize}
    }

    \end{bbrbox}
    \bbrinput{input}
    \bbroutput{output}

    \begin{bbroracle}{OraA}
      \begin{bbrbox}[name=Oracle 1,minheight=1cm]
      \end{bbrbox}
    \end{bbroracle}
    \bbroraclequeryto{top=$m$}
    \bbroraclequeryfrom{top=$b$}

    \begin{bbroracle}{OraB}
      \begin{bbrbox}[name=Oracle 2,minheight=1cm]
      \end{bbrbox}
    \end{bbroracle}
    \bbroraclequeryto{top=$m$}
    \bbroraclequeryfrom{top=$b$}
  \end{bbrenv}
```

We'll discuss the details in Chapter 7.

Chapter 2

Cryptographic Notation

In this section we'll discuss the various commands for notation that can be loaded via package options.

```
\usepackage[
  n,
  advantage,
  operators,
  sets,
  adversary,
  landau,
  probability,
  notions,
  logic,
  ff,
  mm,
  primitives,
  events,
  complexity,
  asymptotics,
  keys
  ]{cryptocode}
```

Remark. The commands defined so far are far from complete and are currently mostly targeted at what I needed in my papers (especially once you get to cryptographic notions and primitives). So please if you feel that something should be added drop me an email.

2.1 Security Parameter

In cryptography we make use of a security parameter which is usually written as 1^n or 1^λ. The cryptocode package, when loading either option "n" or option "lambda" will define the commands

```
  \secpar
  \secparam
```

The first command provides the "letter", i.e., either n or λ, whereas `\secparam` points to 1^n.

2.2 Advantage Terms

Load the package option "advantage" in order to define the command `\advantage` used to specify advantage terms such as:

$$\mathsf{Adv}^{\mathrm{prf}}_{\mathcal{A},\mathsf{PRF}}(n) = \mathsf{negl}(n)$$

```
  \advantage{prf}{\adv,\prf} = \negl
```

Specify an optional third parameter to replace the (n).

```
\advantage{prf}{\adv,\prf}[(arg)]
```

In order to redefine the styles in which superscript and subscript are set redefine

```
\renewcommand{\pcadvantagesuperstyle}[1]{\mathrm{\MakeLowercase{#1}}}
\renewcommand{\pcadvantagesubstyle}[1]{#1}
```

2.3 Math Operators

The "operators" option provides the following list of commands:

Command	Description	Result	Example
\sample	Sampling from a distribution, or running a randomized procedure	$\leftarrow_{\$}$	$b \leftarrow_{\$} \{0,1\}$
\floor{42.5}	Rounding down	$\lfloor 42.5 \rfloor$	
\ceil{41.5}	Rounding up	$\lceil 41.5 \rceil$	
\Angle{x,y}	Vector product	$\langle x, y \rangle$	
\abs{42.9}	Absolute number	$\lvert 42.9 \rvert$	
\norm{x}	Norm	$\lVert x \rVert$	
\concat	Verbose concatenation (I usually prefer simply |)	$\Vert$	$x \leftarrow a \Vert b$
\emptystring	The empty string	ε	$x \leftarrow \varepsilon$

2.4 Adversaries

The "adversary" option provides the following list of commands:

Command	Description	Result
\adv	Adversary	$\mathcal{A}$
\bdv	Adversary	$\mathcal{B}$
\cdv	Adversary	$\mathcal{C}$
\ddv	Adversary	$\mathcal{D}$
\mdv	Adversary	$\mathcal{M}$
\pdv	Adversary	$\mathcal{P}$
\sdv	Adversary	$\mathcal{S}$

The style in which an adversary is rendered is controlled via

```
\renewcommand{\pcadvstyle}[1]{\mathcal{#1}}
```

2.5 Landau

The "landau" option provides the following list of commands:

Command	Description	Result
\bigO{n^2}	Big O notation	$\mathcal{O}\left(n^2\right)$
\smallO{n^2}	small o notation	$\mathrm{o}\left(n^2\right)$
\bigOmega{n^2}	Big Omega notation	$\Omega\left(n^2\right)$
\bigsmallO{n^2}	Big and small O notation	$\Theta\left(n^2\right)$

2.6 Probabilities

The "probability" option provides commands for writing probabilities. Use

```
\prob{X=x}
\probsub{x\sample{\bin^n}}{x=5}
\condprob{X=x}{A=b}
\condprobsub{x\sample{\bin^n}}{x=5}{A=b}
```

to write basic probabilities, probabilities with explicit probability spaces and conditional probabilities.

$$\Pr[X = x]$$
$$\Pr_{x \leftarrow\$ \{0,1\}^n}[X = x]$$
$$\Pr[X = x \mid A = b]$$
$$\Pr_{x \leftarrow\$ \{0,1\}^n}[x = 5 \mid A = b]$$

You can control the probability symbol (Pr) by redefining

```
\renewcommand{\probname}{Pr}
```

For expectations you can use

```
\expect{X}
\expsub{x,y\sample\set{1,\ldots,6}}{x+y}
\condexp{X+Y}{Y>3}
\condexpsub{x,y\sample\set{1,\ldots,6}}{x+y}{y>3}
```

yielding

$$\mathbb{E}[X]$$
$$\mathbb{E}_{x,y \leftarrow\$ \{1,\ldots,6\}}[x + y]$$
$$\mathbb{E}[X + Y \mid Y > 3]$$
$$\mathbb{E}_{x,y \leftarrow\$ \{1,\ldots,6\}}[x + y \mid y > 3]$$

You can control the expactation symbol ($\mathbb{E}$) by redefining

```
\renewcommand{\expectationname}{\ensuremath{\mathbb{E}}}
```

The support $\mathrm{Supp}(X)$ of a random variable X can be written as

```
\supp{X}
```

where again the name can be controlled via

```
\renewcommand{\supportname}{Supp}
```

For denoting entropy and min-entropy use

```
\entropy{X}
\minentropy{X}
\condminentropy{X}{Y=5}
```

This yields

$$\mathrm{H}(X)$$
$$\mathrm{H}_\infty(X)$$
$$\tilde{\mathrm{H}}_\infty(X|Y = 5)$$

2.7 Sets

The "sets" option provides commands for basic mathematical sets. You can write sets and sequences as

```
\set{1, \ldots, 10}
\sequence{1, \ldots, 10}
```

which is typeset as

$$\{1, \ldots, 10\}$$
$$(1, \ldots, 10)$$

In addation the following commands are provided

Command	Description	Result
\bin	The set containing 0 and 1	$\{0,1\}$
\NN	Natural numbers	$\mathbb{N}$
\ZZ	Integers	$\mathbb{Z}$
\QQ	Rational numbers	$\mathbb{Q}$
\CC	Complex numbers	$\mathbb{C}$
\RR	Reals	$\mathbb{R}$
\PP		$\mathbb{P}$
\FF		$\mathbb{F}$

2.8 Crypto Notions

The "notions" option provides the following list of commands:

Command	Description	Result
\indcpa	IND-CPA security for encryption schemes	IND-CPA
\indcca	IND-CCA security for encryption schemes	IND-CCA
\indccai	IND-CCA1 security for encryption schemes	IND-CCA1
\indccaii	IND-CCA2 security for encryption schemes	IND-CCA2
\priv	PRIV security for deterministic public-key encryption schemes	PRIV
\ind	IND security (for deterministic public-key encryption schemes)	IND
\prvcda	PRV-CDA security (for deterministic public-key encryption schemes)	PRV-CDA
\prvrcda	PRV$-CDA security (for deterministic public-key encryption schemes)	PRV$-CDA
\kiae	Key independent authenticated encryption	KIAE
\kdae	Key dependent authenticated encryption	KDAE
\mle	Message locked encryption	MLE
\uce	Universal computational extractors	UCE

The style in which notions are displayed can be controlled via redefining

```
\renewcommand{\pcnotionstyle}[1]{\ensuremath{\mathrm{#1}}}
```

2.9 Logic

The "logic" option provides the following list of commands:

Command	Description	Result
\AND	Logical AND	AND
\OR	Logical OR	OR
\NOT	not	NOT
\xor	exclusive or	$\oplus$
\false	false	false
\true	true	true

2.10 Function Families

The “ff” option provides the following list of commands:

Command	Description	Result
\kgen	Key generation	KGen
\pgen	Parameter generation	Pgen
\eval	Evaluation	Eval
\il	Input length	il
\ol	Output length	ol
\kl	Key length	kl
\nl	Nonce length	nl
\rl	Randomness length	rl

The style in which these are displayed can be controlled via redefining

```
\renewcommand{\pcalgostyle}[1]{\ensuremath{\mathsf{#1}}}
```

2.11 Machine Model

The “mm” option provides the following list of commands:

Command	Description	Result
\CRKT	A circuit	C
\TM	A Turing machine	M
\PROG	A program	P
\uTM	A universal Turing machine	UM
\uC	A universal Circuit	UC
\uP	A universal Program	UEval
\tmtime	Time (of a TM)	time
\ppt	Probabilistic polynomial time	PPT

The style in which these are displayed can be controlled via redefining

```
\renewcommand{\pcmachinemodelstyle}[1]{\ensuremath{\mathsf{#1}}}
```

2.12 Crypto Primitives

The “primitives” option provides the following list of commands:

Command	Description	Result
\prover	Proover	P
\verifier	Verifier	V
\nizk	Non interactie zero knowledge	NIZK
\hash	A hash function	H
\gash	A hash function	G
\fash	A hash function	F
\enc	Encryption	Enc
\dec	Decryption	Dec
\sig	Signing	Sig
\verify	Verifying	Vf
\obf	Obfuscation	O
\iO	Indistinguishability obfuscation	iO
\diO	Differing inputs obfuscation	diO
\mac	Message authentication	MAC
\puncture	Puncturing	$\mathsf{Puncture}$
\source	A source	S
\predictor	A predictor	P
\sam	A sampler	Sam
\distinguisher	A distinguisher	Dist
\dist	A distinguisher	D
\simulator	A simulator	Sim
\ext	An extractor	Ext

The style in which these are displayed can be controlled via redefining

```
\renewcommand{\pcalgostyle}[1]{\ensuremath{\mathsf{#1}}}
```

2.13 Events

The "events" option provides the following list of commands.

To classify an event use

```
\event{Event}
\nevent{Event}
```

where the second is meant as the negation. These are typset as

$$\mathsf{Event} \qquad \overline{\mathsf{Event}}$$

For bad events, use \bad (bad).

2.14 Complexity

The "complexity" option provides the following list of commands:

Command	Result
\npol	NP
\conpol	coNP
\pol	P
\bpp	BPP
\ppoly	P/poly
\NC{1}	NC^1
\AC{1}	AC^1
\TC{1}	TC^1
\AM	AM
\coAM	coAM

The style in which these are displayed can be controlled via redefining

```
\renewcommand{\pccomplexitystyle}[1]{\ensuremath{\mathsf{#1}}}
```

2.15 Asymptotics

The "asymptotics" option provides the following list of commands:

Command	Description	Result
\negl	A negligible function	$\mathsf{negl}(n)$ (takes an optional argument \negl[a] ($\mathsf{negl}(a)$). Write \negl[] for negl.)
\poly	A polynomial	$\mathsf{poly}(n)$ (takes an optional argument \poly[a] ($\mathsf{poly}(a)$). Write \poly[] for poly.)
\pp	some polynomial p	p
\qq	some polynomial q	q

The style in which these are displayed can be controlled via redefining

```
\renewcommand{\pcpolynomialstyle}[1]{\ensuremath{\mathrm{#1}}}
```

2.16 Keys

The "keys" option provides the following list of commands:

Command	Description	Result
\pk	public key	pk
\vk	verification key	vk
\sk	secret key	sk
\key	a plain key	k
\hk	hash key	hk
\gk	gash key	gk
\fk	function key	fk

The style in which these are displayed can be controlled via redefining

```
\renewcommand{\pckeystyle}[1]{\ensuremath{\mathsf{#1}}}
```

Chapter 3

Pseudocode

In this chapter we discuss how to write pseudocode with the cryptocode library.

3.1 Basics

The cryptocode package provides the command *pseudocode* in order to write simple cryptostyle algorithms. Consider the following definition of an IND-CPA game

$b \leftarrow_{\$} \{0,1\}$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow_{\$} \mathsf{KGen}(1^n)$
$(m_0, m_1) \leftarrow_{\$} \mathcal{A}(1^n, \mathsf{pk}, c)$
$c \leftarrow_{\$} \mathsf{Enc}(\mathsf{pk}, m_b)$
$b' \leftarrow_{\$} \mathcal{A}(1^n, \mathsf{pk}, c)$
return $b = b'$

which is generated as

```
\pseudocode{%
    b \sample \bin   \\
    (\pk,\sk) \sample \kgen (\secparam)   \\
    (m_0,m_1) \sample \adv(\secparam, \pk, c)   \\
    c \sample \enc(\pk,m_b)   \\
    b' \sample \adv(\secparam, \pk, c)   \\
    \pcreturn b = b' }
```

As you can see the pseudocode command provides a math based environment where you can simply start typing your pseudocode separating lines by \\.

Boxed appearance Although most examples here appear centered and boxed this is not directly part of the pseudocode package but due to the examples being typeset as

```
\begin{center}
\fbox{%
Code
}
\end{center}
```

3.1.1 Customizing Pseudocode

Besides the mandatory argument the \pseudocode command can take an optional argument which consists of a list of key=value pairs separated by commas (,).

```
\pseudocode[options]{body}
```

The following keys are available:

head A header for the code

width An exact width. If no width is specified, cryptocode tries to automatically compute the correct width.

lnstart The starting line number when using line numbering.

lnstartright The starting line number for right aligned line numberswhen using line numbering.

linenumbering Enables line numbering.

syntaxhighlight When set to "auto" cryptocode will attempt to automatically hightlight keywords such as "for", "foreach" and "return"

keywords Provide a comma separated list of keywords for automatic syntax highlighting. To customize the behavior of automatic spacing you can provide keywords as

keywordsindent After seeing this keyword all following lines will be indented one extra level.

keywordsunindent After seeing this keyword the current and all following lines will be unindented one extra level.

keywordsuninindent After seeing this keyword the current line will be unindented one level.

addkeywords Provide additional keywords for automatic syntax highlighting.

altkeywords Provide a second list of keywords for automatic syntax highlighting that are highlighted differently.

mode When set to text pseudocode will not start in math mode but in text mode.

space Allows you to enable automatic spacing mode. If set to "keep" the spaces in the input are preserved. If set to "auto" it will try to detect spacing according to keywords such as "if" and "fi".

xshift Allows horizontal shifting

colsep Defines the space between columns.

addtolength Is added to the automatically computed width of the pseudocode (which does not take colsep into account).

nodraft Forces syntax highlighting also in draft mode.

The following code

```
\pseudocode[linenumbering,syntaxhighlight=auto,head=Header]{ return null }
```

creates

Header
1 : **return null**

3.1.2 Customized Pseudocode Commands

Besides the \pseudocode command the command \procedure provides easy access to generate code with a header. It takes the following form

```
\procedure[options]{Header}{Body}
```

Examples

$\mathrm{IND\text{-}CPA}^{\mathcal{A}}_{\mathsf{Enc}}$

$b \leftarrow_{\$} \{0,1\}$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow_{\$} \mathsf{KGen}(1^n)$
$(m_0, m_1) \leftarrow_{\$} \mathcal{A}(1^n, \mathsf{pk}, c)$
$c \leftarrow_{\$} \mathsf{Enc}(\mathsf{pk}, m_b)$
$b' \leftarrow_{\$} \mathcal{A}(1^n, \mathsf{pk}, c)$
return $b = b'$

which is generated as

```
\procedure{$\indcpa_\enc^\adv$}{%
   b \sample \bin   \\
   (\pk,\sk) \sample \kgen(\secparam)   \\
   (m_0,m_1) \sample \adv(\secparam, \pk, c)   \\
   c \sample \enc(\pk,m_b)   \\
   b' \sample \adv(\secparam, \pk, c)   \\
   \pcreturn b = b' }
```

You can define customized pseudocode commands with either take one optional argument and two mandatory arguments (as the procedure command) or one optional and one mandatory argument (as the pseudocode command). The following

```
\createprocedurecommand{mypseudocode}{}{}{linenumbering}
\createpseudocodecommand{myheadlesscmd}{}{}{linenumbering}
```

creates the commands \mypseudocode and \myheadlesscmd with line numbering always enabled. The first command has an identical interface as the \pseudocode command, the second has an interface as the \procedure command. The second and third argument that we kept empty when generating the commands allows us to specify commands that are executed at the very beginning when the command is called (argument 2) and a prefix for the header.

3.2 Indentation

In order to indent code use \pcind or short \t. You can also usecustomized spacing such as \quad or \hspace when using the pseudocode command in math mode.

for $i = 1..10$ **do**
  $T[i] \leftarrow_{\$} \{0,1\}^n$
for $i = 1..10$ **do**
  $T[i] \leftarrow_{\$} \{0,1\}^n$

which is generated as

```
\pseudocode{%
   \pcfor i = 1..10 \pcdo   \\
   \pcind T[i] \sample \bin^n   \\
   \pcfor i = 1..10 \pcdo   \\
   \t T[i] \sample \bin^n }
```

You can specify multiple levels via the optional first argument

```
\pcind[level]
```

for $i = 1..10$ **do**
  $T[i] \leftarrow\$ \{0,1\}^n$
    $T[i] \leftarrow\$ \{0,1\}^n$
      $T[i] \leftarrow\$ \{0,1\}^n$
        $T[i] \leftarrow\$ \{0,1\}^n$
          $T[i] \leftarrow\$ \{0,1\}^n$

```
\pseudocode{%
    \pcfor i = 1..10 \pcdo  \\
    \pcind T[i] \sample \bin^n  \\
    \pcind\pcind T[i] \sample \bin^n \\
    \pcind[3] T[i] \sample \bin^n \\
    \pcind[4] T[i] \sample \bin^n \\
    \pcind[5] T[i] \sample \bin^n }
```

You can customize the indentation shortcut by redefining

```
\renewcommand{\pcindentname}{t}
```

Automatic Indentation

The pseudocode command comes with an option "space=auto" which tries to detect the correct indentation from the use of keywords. When it sees one of the following keywords

```
\pcif, \pcfor, \pcwhile, \pcrepeat, \pcforeach
```

it will increase the indentation starting from the next line. It will again remove the indentation on seeing

```
\pcfi, \pcendif, \pcendfor, \pcendwhile, \pcuntil, \pcendforeach
```

Additionally, on seeing

```
\pcelse, \pcelseif
```

it will remove the indentation for that particular line. Thus the following

for $a \in [10]$ **do**
  for $a \in [10]$ **do**
    for $a \in [10]$ **do**
      if $a = b$ **then**
        some operation
      elseif $a = c$ **then**
        some operation
      else
        some default operation
      fi
    endfor
  endfor
endfor
return a

can be obtained by:

```
\pseudocode[space=auto]{%
\pcfor a \in [10] \pcdo \\
  \pcfor a \in [10] \pcdo \\
    \pcfor a \in [10] \pcdo \\
      \pcif a = b \pcthen \\
        \text{some operation} \\
      \pcelseif a = c \pcthen \\
        \text{some operation} \\
      \pcelse \\
        \text{some default operation} \\
      \pcfi \\
    \pcendfor \\
  \pcendfor \\
\pcendfor \\
\pcreturn a}
```

Note that the manual indentation in the above example is not necessary for the outcome. Further note that the same works when using automatic syntax highlighting (see Section 3.4).

Keep Input Indentation

The pseudocode package comes with an experimental feature that preserves the spacing in the input. This can be enabled with the option "space=keep". Thus the above can also be written as

$$\begin{array}{l}\textbf{for } i = 1..10 \textbf{ do}\\ \quad T[i] \leftarrow\!\!\$\ \{0,1\}^n\\ \quad\quad T[i] \leftarrow\!\!\$\ \{0,1\}^n\\ \quad\quad\quad T[i] \leftarrow\!\!\$\ \{0,1\}^n\\ \quad\quad\quad\quad T[i] \leftarrow\!\!\$\ \{0,1\}^n\\ \quad\quad\quad\quad\quad T[i] \leftarrow\!\!\$\ \{0,1\}^n\end{array}$$

```
\pseudocode[space=keep]{%
\pcfor i = 1..10 \pcdo  \\
    T[i] \sample \bin^n  \\
        T[i] \sample \bin^n \\
            T[i] \sample \bin^n \\
                T[i] \sample \bin^n \\
                    T[i] \sample \bin^n }
```

Note that automatic spacing only works when the \pseudocode command is not wrapped within another command. Thus in order to get a frame box \fbox{\pseudocode[space=keep]{code}} will not work but you would need to use an environment such as one offered by the *mdframed* package ((`https://www.ctan.org/pkg/mdframed`). Also see Section 8.1.

3.3 Textmode

By default pseudocode enables LaTeX' math mode. You can change this behavior and tell the pseudocode command to interpret the content in text mode by setting the option "mode=text".

This is
  simply text

```
\pseudocode[mode=text]{%
This is \\
\t simply text}
```

3.4 Syntax Highlighting

In the above examples we have used commands \pcreturn and \pcfor to highlight certain keywords. Besides the *pcreturn*, *pcfor* and *pcdo* (where the pc stands for pseudocode) that were used in the above examples the package defines the following set of constants:

name	**usage**	**outcome**
pccontinue	\pccontinue	**continue**
pccomment	\pccomment{comment}	// comment
pcdo	\pcdo	**do**
pcdone	\pcdone	**done**
pcfalse	\pcfalse	**false**
pcif	\pcif	**if**
pcfi	\pcfi	**fi**
pcendif	\pcendif	**endif**
pcelse	\pcelse	**else**
pcelseif	\pcelseif	**elseif**
pcfor	\pcfor	**for**
pcendfor	\pcendfor	**endfor**
pcforeach	\pcforeach	**foreach**
pcendforeach	\pcendforeach	**endforeach**
pcglobvar	\pcglobvar	**gbl**
pcin	\pcin	**in**
pcnew	\pcnew	**new**
pcnull	\pcnull	**null**
pcparse	\pcparse	**parse**
pcrepeat	\pcrepeat{10}	**repeat** 10 **times**
pcuntil	\pcuntil	**until**
pcreturn	\pcreturn	**return**
pcthen	\pcthen	**then**
pctrue	\pctrue	**true**
pcwhile	\pcwhile	**while**
pcendwhile	\pcendwhile	**endwhile**

Note that \pcdo, \pcin and \pcthen have a leading space. This is due to their usual usage scenarios such as

$$\textbf{for } i \textbf{ in} \{1, \ldots, 10\}$$

Furthermore all constants have a trailing space. This can be removed by adding the optional parameter [] such as

$$\textbf{for } i\textbf{in} \{1, \ldots, 10\}$$

```
\pseudocode{\pcfor i \pcin[] \{1,\ldots,10\}}
```

In order to change the font you can overwrite the command \highlightkeyword which is defined as

```
\newcommand{\highlightkeyword}[2][\ ]{\ensuremath{\mathbf{#2}}#1}
```

Automatic Syntax Highlighting

The pseudocode command comes with an experimental feature to automatically highlight keywords. This can be activated via the option “syntaxhighlight=auto”. The preset list of keywords it looks for are

```
for ,foreach ,return ,{ do }, in ,new ,if ,null ,null,true ,true,until ,{ to },false ,false,{
    then },repeat ,else ,done ,done,fi
```

Note that the keywords are matched with spaces and note the grouping for trailing spaces. That is, the " do " keyword won't match within the string "don't". Via the option "keywords" you can provide a custom list of keywords. Thus the following bubblesort variant (taken from `http://en.wikipedia.org/wiki/Bubble_sort`)

Bubblesort(A : list of items)

$n \leftarrow \mathsf{length}(A)$
repeat
  $s \leftarrow$ **false**
  for $i = 1 to n - 1 do$
    // **if** this pair is out of order
    if $A[i-1] > A[i]$ **then**
      // swap them and remember something changed
      $\mathsf{swap}(A[i-1], A[i])$
      $s \leftarrow$ **true**
until $\neg s$

can be typeset as

```
\procedure[syntaxhighlight=auto]{Bubblesort(A : list of items)}{
   n \gets \mathsf{length}(A) \\
   repeat \\
   \t s \gets false \\
   \t  for i = 1 to n-1 do \\
   \t\t  \pccomment{if this pair is out of order} \\
   \t\t   if A[i-1] > A[i] then \\
   \t\t\t \pccomment{swap them and remember something changed} \\
   \t\t\t     \mathsf{swap}( A[i-1], A[i] ) \\
   \t\t\t       s \gets true \\
   until \neg s }
```

You can also define additional keywords using the "addkeywords" option. This would allow us to specify "length" and "swap" in the above example. Combined with automatic spacing we could thus get

Bubblesort(A : list of items)

$n \leftarrow$ **length**(A)
repeat
  $s \leftarrow$ **false**
  for $i = 1$ **to** $n - 1$ **do**
    // **if** this pair is out of order
    if $A[i-1] > A[i]$ **then**
      // **swap** them and remember something changed
      swap$(A[i-1], A[i])$
      $s \leftarrow$ **true**
until $\neg s$

Simply by writing (note the `\neg{}` in order to not have a space before s):

```
\procedure[space=keep,syntaxhighlight=auto,addkeywords={swap,length}]{Bubblesort(A : list of
    items)}{
n \gets length(A) \\
repeat \\
```

```
    s \gets false \\
    for i=1 to n-1 do \\
        \pccomment{if this pair is out of order} \\
        if A[i-1]>A[i] then \\
            \pccomment{swap them and remember something changed} \\
            swap(A[i-1], A[i]) \\
            s \gets true \\
until \neg{}s }
```

Also note that a simple \fbox around the above \procedure command has the effect that the automatic spacing fails. For this also see Section 8.1. As an alternative we could use automatic spacing and insert "group end" keywords such as "fi":

Bubblesort(A : list of items)

$n \leftarrow$ **length**(A)
repeat
  $s \leftarrow$ **false**
  for $i = 1 to n - 1 do$
    // assuming this pair is out of order
    if $A[i-1] > A[i]$ **then**
      // **swap** them and remember something changed
      swap$(A[i-1], A[i])$
      $s \leftarrow$ **true**
    endif
  endfor
until $\neg s$

The last example is generated as (note that here fbox is fine.)

```
\fbox{\procedure[space=auto,syntaxhighlight=auto,addkeywords={swap,length}]{Bubblesort(A :
    list of items)}{
n \gets length(A) \\
repeat \\
    s \gets false \\
    for i=1 to n-1 do \\
        \pccomment{assuming this pair is out of order} \\
        if A[i-1]>A[i] then \\
            \pccomment{swap them and remember something changed} \\
            swap(A[i-1], A[i]) \\
            s \gets true \\
        endif \\
     endfor \\
until \neg s }}
```

3.4.1 Alternative Keywords

There is a second keyword list that you can add keywords to which are highlighted not via \highlightkeyword but via \highlightaltkeyword where alt stands for alternate. This allows you to have two different keyword styles which are by default defined as

```
\newcommand{\highlightkeyword}[2][\ ]{\ensuremath{\mathbf{#2}}#1}
\newcommand{\highlightaltkeyword}[1]{\ensuremath{\mathsf{#1}}}
```

This allows you to rewrite the above example and highlight the different nature of swap and length.

Bubblesort(A : list of items)

$n \leftarrow \mathsf{length}(A)$
repeat
  $s \leftarrow$ **false**
  for $i = 1 to n - 1 do$
    // assuming this pair is out of order
    if $A[i-1] > A[i]$ **then**
      // swap them and remember something changed
      $\mathsf{swap}(A[i-1], A[i])$
      $s \leftarrow$ **true**
    endif
  endfor
until $\neg s$

```
\procedure[space=auto,syntaxhighlight=auto,addkeywords={swap,length}]{Bubblesort(A : list of
    items)}{
n \gets length(A) \\
repeat \\
    s \gets false \\
    for i=1 to n-1 do \\
        \pccomment{assuming this pair is out of order} \\
        if A[i-1]>A[i] then \\
            \pccomment{swap them and remember something changed} \\
            swap(A[i-1], A[i]) \\
            s \gets true \\
        endif \\
     endfor \\
until \neg s }
```

3.4.2 Draft Mode

Automatic syntax highlighting is a somewhat expensive operation as it requires several rounds of regular expression matching. In order to speed up compilation the pseudocode command will not attempt automatic highlighting when the document is in draft mode. When in draft mode and you want to force a specific instance of `\pseudocode` to render the code with automatic syntax highlighting you can use the option `nodraft`.

3.5 Line Numbering

The pseudocode command allows to insert line numbers into pseudocode. You can either manually control line numbering or simply turn on the option "linenumbering".

$\text{IND-CPA}^{\mathcal{A}}_{\mathsf{Enc}}$

1 : $b \leftarrow\!\!\$\ \{0,1\}$
2 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\!\$\ \mathsf{KGen}1^n)$
3 : $(m_0, m_1) \leftarrow\!\!\$\ \mathcal{A}(1^n, \mathsf{pk}, c)$
4 : $c \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{pk}, m_b)$
5 : $b' \leftarrow\!\!\$\ \mathcal{A}(1^n, \mathsf{pk}, c)$
6 : **return** $b = b'$

is generated by

```
\procedure[linenumbering]{$\indcpa_\enc^\adv$}{%
```

```
  b \sample \bin \\
 (\pk,\sk) \sample \kgen\secparam)  \\
  \label{tmp:line:label}   (m_0,m_1) \sample \adv(\secparam, \pk, c)  \\
  c \sample \enc(\pk,m_b)  \\
  b' \sample \adv(\secparam, \pk, c)  \\
 \pcreturn b = b' }
```

Note how you can use labels such as \label{tmp:line:label} which now points to 3.

3.5.1 Manually Inserting Line Numbers

In order to manually insert line numbers use the command \pcln.

$\text{IND-CPA}^{\mathcal{A}}_{\mathsf{Enc}}$

1 : $b \leftarrow_{\$} \{0,1\}$
2 : $(\mathsf{pk},\mathsf{sk}) \leftarrow_{\$} \mathsf{KGen}1^n)$
3 : $(m_0,m_1) \leftarrow_{\$} \mathcal{A}(1^n,\mathsf{pk},c)$
4 : $c \leftarrow_{\$} \mathsf{Enc}(\mathsf{pk},m_b)$
5 : $b' \leftarrow_{\$} \mathcal{A}(1^n,\mathsf{pk},c)$
6 : **return** $b = b'$

is generated by

```
\procedure{$\indcpa_\enc^\adv$}{%
 \pcln  b \sample \bin \\
 \pcln   (\pk,\sk) \sample \kgen\secparam)  \\
 \pcln\label{tmp:line:label2}   (m_0,m_1) \sample \adv(\secparam, \pk, c)  \\
 \pcln   c \sample \enc(\pk,m_b)  \\
 \pcln   b' \sample \adv(\secparam, \pk, c)  \\
 \pcln   \pcreturn b = b' }
```

Note that the label *tmp:line:label2* now points to line number 3.

3.5.2 Start Values

You can specify the start value (-1) of the counter by setting the option “lnstart”.

```
\procedure[lnstart=10,linenumbering]{Header}{Body}
```

$\text{IND-CPA}^{\mathcal{A}}_{\mathsf{Enc}}$

11 : $b \leftarrow_{\$} \{0,1\}$
12 : $(\mathsf{pk},\mathsf{sk}) \leftarrow_{\$} \mathsf{KGen}1^n)$
13 : $(m_0,m_1) \leftarrow_{\$} \mathcal{A}(1^n,\mathsf{pk},c)$
14 : $c \leftarrow_{\$} \mathsf{Enc}(\mathsf{pk},m_b)$
15 : $b' \leftarrow_{\$} \mathcal{A}(1^n,\mathsf{pk},c)$
16 : **return** $b = b'$

3.5.3 Separators

The commands \pclnseparator defines the separator between the pseudocode and the line numbering. By default the left separator is set to (:) colon. Also see Section 5.3.1.

3.6 Subprocedures

The pseudocode package allows the typesetting of sub procedures such as

$\text{IND-CPA}^{\mathcal{A}}_{\mathsf{Enc}}$

1 : $b \leftarrow_{\$} \{0,1\}$
2 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow_{\$} \mathsf{KGen}(1^n)$
3 : $(m_0, m_1) \leftarrow_{\$}$

$\mathcal{A}(1^n, \mathsf{pk}, c)$	
1 :	Step 1
2 :	Step 2
3 :	**return** m_0, m_1

4 : $c \leftarrow_{\$} \mathsf{Enc}(\mathsf{pk}, m_b)$
5 : $b' \leftarrow_{\$} \mathcal{A}(1^n, \mathsf{pk}, c)$
6 : **return** $b = b'$

To create a subprocedure use the *subprocedure* environment. The above example is generated via

```
\procedure[linenumbering]{$\indcpa_\enc^\adv$}{%
   b \sample \bin   \\
   (\pk,\sk) \sample \kgen(\secparam)   \\
   (m_0,m_1) \sample \begin{subprocedure}%
 \dbox{\procedure{$\adv(\secparam, \pk, c)$}{%
     \text{Step 1}   \\
     \text{Step 2}   \\
   \pcreturn m_0, m_1 }}
  \end{subprocedure}   \\
   c \sample \enc(\pk,m_b)   \\
   b' \sample \adv(\secparam, \pk, c)   \\
   \pcreturn b = b' }
```

Here the dbox command (from the dashbox package) is used to generate a dashed box around the sub procedure.

3.6.1 Numbering in Subprocedures

Subprocedures as normal pseudocode allow you to create line numbers. By default the line numbering starts with 1 in a subprocedure while ensuring that the outer numbering remains intact. Also note that the linenumbering on the outer procedure in the above example is inherited by the subprocedure. For more control, either use manual numbering or set the option "linenumbering=off" on the subprocedure.

$\text{IND-CPA}^{\mathcal{A}}_{\mathsf{Enc}}$

1 : $b \leftarrow_{\$} \{0,1\}$
2 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow_{\$} \mathsf{KGen}(1^n)$
3 : $(m_0, m_1) \leftarrow_{\$}$

$\mathcal{A}(1^n, \mathsf{pk}, c)$	
1 :	Step 1
2 :	Step 2
3 :	**return** m_0, m_1

4 : $c \leftarrow_{\$} \mathsf{Enc}(\mathsf{pk}, m_b)$
5 : $b' \leftarrow_{\$} \mathcal{A}(1^n, \mathsf{pk}, c)$
6 : **return** $b = b'$

```
\procedure{$\indcpa_\enc^\adv$}{%
  \pcln   b \sample \bin   \\
  \pcln   (\pk,\sk) \sample \kgen(\secparam)   \\
  \pcln   (m_0,m_1) \sample \begin{subprocedure}%
```

```
  \dbox{\procedure{$\adv(\secparam, \pk, c)$}{%
    \pcln \text{Step 1}  \\
    \pcln \text{Step 2}  \\
  \pcln \pcreturn m_0, m_1 }}
  \end{subprocedure}  \\
\pcln   c \sample \enc(\pk,m_b)  \\
\pcln   b' \sample \adv(\secparam, \pk, c)  \\
\pcln   \pcreturn b = b' }
```

3.7 Stacking Procedures

You can stack procedures horizontally or vertically using the environments "pchstack" and "pcvstack".

```
\begin{pchstack}[center] body \end{pchstack}
\begin{pcvstack}[center] body \end{pcvstack}
```

The following example displays two procedures next to one another. As a spacing between two horizontally outlined procedures use \pchspace which takes an optional length as a parameter.

$\text{IND-CPA}^{\mathcal{A}}_{\mathsf{Enc}}$

1 : $b \leftarrow\!\!\$\ \{0,1\}$
2 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\!\$\ \mathsf{KGen}(1^n)$
3 : $(m_0, m_1) \leftarrow\!\!\$\ \mathcal{A}^{O}(1^n, \mathsf{pk})$
4 : $c \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{pk}, m_b)$
5 : $b' \leftarrow\!\!\$\ \mathcal{A}(1^n, \mathsf{pk}, c)$
6 : **return** $b = b'$

Oracle O

1 : line 1
2 : line 2

```
\begin{pchstack}[center]
\procedure{$\indcpa_\enc^\adv$}{%
  \pcln  b \sample \bin  \\
  \pcln  (\pk,\sk) \sample \kgen(\secparam)  \\
  \pcln  (m_0,m_1) \sample \adv^O(\secparam, \pk)  \\
\pcln   c \sample \enc(\pk,m_b)  \\
\pcln   b' \sample \adv(\secparam, \pk, c)  \\
\pcln   \pcreturn b = b' }

\pchspace

\procedure{Oracle $O$}{%
  \pcln  \text{line 1}  \\
  \pcln  \text{line 2}
}
\end{pchstack}
```

Similarly you can stack two procedures vertically using the "pcvstack" environment. As a spacing between two vertically stacked procedures use \pcvspace which takes an optional length as a parameter.

$\text{IND-CPA}^{\mathcal{A}}_{\mathsf{Enc}}$

1 : $b \leftarrow_{\$} \{0,1\}$
2 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow_{\$} \mathsf{KGen}(1^n)$
3 : $(m_0, m_1) \leftarrow_{\$} \mathcal{A}^{O}(1^n, \mathsf{pk})$
4 : $c \leftarrow_{\$} \mathsf{Enc}(\mathsf{pk}, m_b)$
5 : $b' \leftarrow_{\$} \mathcal{A}(1^n, \mathsf{pk}, c)$
6 : **return** $b = b'$

Oracle O

1 : line 1
2 : line 2

```
\begin{pcvstack}[center]
\procedure{$\indcpa_\enc^\adv$}{%
  \pcln  b \sample \bin  \\
  \pcln  (\pk,\sk) \sample \kgen(\secparam)  \\
  \pcln  (m_0,m_1) \sample \adv^O(\secparam, \pk)  \\
  \pcln  c \sample \enc(\pk,m_b)  \\
  \pcln  b' \sample \adv(\secparam, \pk, c)  \\
  \pcln  \pcreturn b = b' }

\pcvspace

  \procedure{Oracle $O$}{%
  \pcln  \text{line 1}  \\
  \pcln  \text{line 2}
}
\end{pcvstack}
```

Horizontal and vertical stacking can be combined

$\text{IND-CPA}^{\mathcal{A}}_{\mathsf{Enc}}$

1 : $b \leftarrow_{\$} \{0,1\}$
2 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow_{\$} \mathsf{KGen}(1^n)$
3 : $(m_0, m_1) \leftarrow_{\$} \mathcal{A}^{O,H_1,H_2}(1^n, \mathsf{pk})$
4 : $c \leftarrow_{\$} \mathsf{Enc}(\mathsf{pk}, m_b)$
5 : $b' \leftarrow_{\$} \mathcal{A}(1^n, \mathsf{pk}, c)$
6 : **return** $b = b'$

$\text{IND-CPA}^{\mathcal{A}}_{\mathsf{Enc}}$

1 : $b \leftarrow_{\$} \{0,1\}$
2 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow_{\$} \mathsf{KGen}(1^n)$
3 : $(m_0, m_1) \leftarrow_{\$} \mathcal{A}^{O}(1^n, \mathsf{pk})$
4 : $c \leftarrow_{\$} \mathsf{Enc}(\mathsf{pk}, m_b)$
5 : $b' \leftarrow_{\$} \mathcal{A}(1^n, \mathsf{pk}, c)$
6 : **return** $b = b'$

Oracle O	Oracle H_1	Oracle H_2
1 : line 1	1 : line 1	1 : line 1
2 : line 2	2 : line 2	2 : line 2

```
\begin{pchstack}[center]
\begin{pcvstack}
\procedure{$\indcpa_\enc^\adv$}{%
  \pcln  b \sample \bin  \\
  \pcln  (\pk,\sk) \sample \kgen(\secparam)  \\
  \pcln  (m_0,m_1) \sample \adv^{O,H_1,H_2}(\secparam, \pk)  \\
  \pcln  c \sample \enc(\pk,m_b)  \\
  \pcln  b' \sample \adv(\secparam, \pk, c)  \\
  \pcln  \pcreturn b = b' }

\pcvspace

\begin{pchstack}
  \procedure{Oracle $O$}{%
  \pcln  \text{line 1}  \\
  \pcln  \text{line 2}
```

```
}

  \procedure{Oracle $H_1$}{%
    \pcln   \text{line 1}   \\
    \pcln   \text{line 2}
}

  \procedure{Oracle $H_2$}{%
    \pcln   \text{line 1}   \\
    \pcln   \text{line 2}
}
\end{pchstack}
\end{pcvstack}

\pchspace

\procedure{$\indcpa_\enc^\adv$}{%
    \pcln   b \sample \bin   \\
    \pcln   (\pk,\sk) \sample \kgen(\secparam)   \\
    \pcln   (m_0,m_1) \sample \adv^O(\secparam, \pk)   \\
  \pcln     c \sample \enc(\pk,m_b)   \\
  \pcln     b' \sample \adv(\secparam, \pk, c)   \\
  \pcln     \pcreturn b = b' }

\end{pchstack}
```

3.8 Divisions and Linebreaks

Within the pseudocode command you generate linebreaks as
. In order to specify the linewidth you can add an optional argument

```
\\[height]
```

Furthermore, you can add, for example a horizontal line by using the second optional argument and write

```
\\[][\hline]
```

$\text{IND-CPA}^{\mathcal{A}}_{\mathsf{Enc}}$

1 : $b \leftarrow_{\$} \{0,1\}$

2 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow_{\$} \mathsf{KGen}(1^n)$
3 : $(m_0, m_1) \leftarrow_{\$} \mathcal{A}^O(1^n, \mathsf{pk})$
4 : $c \leftarrow_{\$} \mathsf{Enc}(\mathsf{pk}, m_b)$
5 : $b' \leftarrow_{\$} \mathcal{A}(1^n, \mathsf{pk}, c)$
6 : **return** $b = b'$

```
\procedure{$\indcpa_\enc^\adv$}{%
    \pcln   b \sample \bin   \\[2\baselineskip][\hline\hline]
    \pcln   (\pk,\sk) \sample \kgen(\secparam)   \\
    \pcln   (m_0,m_1) \sample \adv^O(\secparam, \pk)   \\
  \pcln     c \sample \enc(\pk,m_b)   \\
  \pcln     b' \sample \adv(\secparam, \pk, c)   \\
  \pcln     \pcreturn b = b' }
```

3.9 Fancy Code with Overlays

Consider the IND-CPA game. Here we have a single adversary $\mathcal{A}$ that is called twice, first to output two messages then given the ciphertext of one of the messages to "guess" which one was encrypted. Often this is not visualized. Sometimes an additional state state is passed as we have in the following example on the left. On the right, we visualize the same thing in a bit more fancy way.

$\mathrm{IND\text{-}CPA}^{\mathcal{A}}_{\mathsf{Enc}}$

1 : $b \leftarrow\!\!\$\ \{0,1\}$
2 : $(\mathsf{pk},\mathsf{sk}) \leftarrow\!\!\$\ \mathsf{KGen}(1^n)$
3 : $(\mathsf{state}, m_0, m_1) \leftarrow\!\!\$\ \mathcal{A}(1^n, \mathsf{pk}, c)$
4 : $c \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{pk}, m_b)$
5 : $b' \leftarrow\!\!\$\ \mathcal{A}(1^n, \mathsf{pk}, c, \mathsf{state})$
6 : **return** $b = b'$

$\mathrm{IND\text{-}CPA}^{\mathcal{A}}_{\mathsf{Enc}}$

1 : $b \leftarrow\!\!\$\ \{0,1\}$
2 : $(\mathsf{pk},\mathsf{sk}) \leftarrow\!\!\$\ \mathsf{KGen}(1^n)$
3 : $(m_0, m_1) \leftarrow\!\!\$\ \mathcal{A}(1^n, \mathsf{pk}, c)$
4 : $c \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{pk}, m_b)$
5 : $b' \leftarrow\!\!\$\ \mathcal{A}(1^n, \mathsf{pk}, c, \mathsf{state})$
6 : **return** $b = b'$

The image on the right is generated by:

```
\begin{pcimage}
\procedure{$\indcpa_\enc^\adv$}{%
  \pcln  b \sample \bin  \\
  \pcln  (\pk,\sk) \sample \kgen (\secparam)  \\
  \pcln  (m_0,m_1) \sample \adv(\secparam, \pk, c) \pcnode{start}  \\
  \pcln   c \sample \enc(\pk,m_b)  \\
  \pcln   b' \sample \adv(\secparam, \pk, c, \state) \pcnode{end} \\
  \pcln   \pcreturn b = b' }

\pcdraw{
  \path[->] (start) edge[bend left=50] node[right]{$\state$} (start|-end);
}
\end{pcimage}
```

In order to achieve the above effect cryptocode utilizes TIKZ underneath. The `pcnode` command generates TIKZ nodes and additionally we wrapped the pseudocode (or procedure) command in an `\begin{pcimage}\end{pcimage}` environment which allows us to utilize these nodes later, for example using the `\pcdraw` command. We can achieve a similar effect without an additional pcimage environment as

```
\procedure{$\indcpa_\enc^\adv$}{%
  \pcln  b \sample \bin  \\
  \pcln  (\pk,\sk) \sample \kgen (\secparam)  \\
  \pcln  (m_0,m_1) \sample \adv(\secparam, \pk, c) \pcnode{start}  \\
  \pcln   c \sample \enc(\pk,m_b)  \\
  \pcln   b' \sample \adv(\secparam, \pk, c, \state) \pcnode{end}[draw={
    \path[->] (start) edge[bend left=50] node[right]{$\state$} (start|-end);
  }] \\
  \pcln   \pcreturn b = b' }
```

3.9.1 Example: Explain your Code

As an exmaple of what you can do with this, let us put an explanation to a line of the code.

$\mathrm{IND\text{-}CPA}^{\mathcal{A}}_{\mathsf{Enc}}$

1 : $b \leftarrow\!\!\$\ \{0,1\}$
2 : $(\mathsf{pk},\mathsf{sk}) \leftarrow\!\!\$\ \mathsf{KGen}(1^n)$
3 : $(m_0, m_1) \leftarrow\!\!\$\ \mathcal{A}(1^n, \mathsf{pk}, c)$
4 : $c \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{pk}, m_b)$
5 : $b' \leftarrow\!\!\$\ \mathcal{A}(1^n, \mathsf{pk}, c, \mathsf{state})$
6 : **return** $b = b'$

$\mathsf{KGen}(1^n)$ samples a public key pk and a private key sk.

```
\begin{center}
\begin{pcimage}
\procedure{$\indcpa_\enc^\adv$}{%
  \pcln  b \sample \bin  \\
  \pcln  (\pk,\sk) \sample \kgen (\secparam)\pcnode{kgen}  \\
  \pcln  (m_0,m_1) \sample \adv(\secparam, \pk, c)  \\
  \pcln   c \sample \enc(\pk,m_b)  \\
  \pcln   b' \sample \adv(\secparam, \pk, c, \state) \\
  \pcln   \pcreturn b = b' }

\pcdraw{
  \node[rectangle callout,callout absolute pointer=(kgen),fill=orange]
      at ([shift={(+3,+1)}]kgen) {
    \begin{varwidth}{3cm}
      $\kgen(\secparam)$ samples a public key $\pk$ and a private key $\sk$.
    \end{varwidth}
  };
}
\end{pcimage}
\end{center}
```

Chapter 4

Tabbing Mode

In the following chapter we discuss how to create multiple columns within a pseudocode command. Within a pseudocode command you can switch to a new column by inserting a \>. This is similar to using an align environment and placing a tabbing character (&). Also, similarly to using align you should ensure that the number of \> are identical on each line.

First	**SecondThird**	**Fourth**
$b \leftarrow_{\$} \{0,1\}$	$b \leftarrow_{\$} \{0,1\} b \leftarrow_{\$} \{0,1\}$	$b \leftarrow_{\$} \{0,1\}$

```
\pseudocode{%
  \textbf{First} \> \textbf{Second} \> \textbf{Third} \> \textbf{Fourth} \\
   b \sample \bin \> b \sample \bin \> b \sample \bin \> b \sample \bin}
```

As you can see the first column is left aligned the second right, the third left and so forth. In order to get only left aligned columns you could thus simply always skip a column by using \>\>. You can also use \< a shorthand for \>\>.

First	**Second**	**Third**	**Fourth**
$b \leftarrow_{\$} \{0,1\}$	$b \leftarrow_{\$} \{0,1\}$	$b \leftarrow_{\$} \{0,1\}$	$b \leftarrow_{\$} \{0,1\}$

```
\pseudocode{%
  \textbf{First} \< \textbf{Second} \< \textbf{Third} \< \textbf{Fourth} \\
   b \sample \bin \< b \sample \bin \< b \sample \bin \< b \sample \bin}
```

Column Spacing You can control the space between columns using the option "colsep=2em". Note that when doing so you should additionally use "addtolength=5em" (where 5em depends on the number of columns) in order to avoid having overfull hboxes.

First	**Second**	**Third**	**Fourth**
$b \leftarrow_{\$} \{0,1\}$	$b \leftarrow_{\$} \{0,1\}$	$b \leftarrow_{\$} \{0,1\}$	$b \leftarrow_{\$} \{0,1\}$

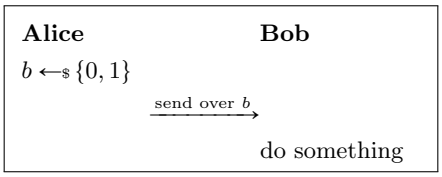
```
\pseudocode{%
\pseudocode[colsep=1em,addtolength=10em]{%
  \textbf{First} \< \textbf{Second} \< \textbf{Third} \< \textbf{Fourth} \\
   b \sample \bin \< b \sample \bin \< b \sample \bin \< b \sample \bin}
```

This is basically all you need to know in order to go on to writing protocols with the cryptocode package. So unless you want to know a bit more about tabbing (switching columns) and learn some of the internals, feel free to proceed to Chapter 5.

4.1 Tabbing in Detail

At the heart of the pseudocode package is an align (or rather a flalign*) environment which allows you to use basic math writing. Usually an align (or flalign) environment uses & as tabbing characters. The pseudocode comes in two modes the first of which changes the default align behavior. That is, it automatically adds a tabbing character to the beginning and end of each line and changes the tabbing character to \>. This mode is called mintabmode and is active by default.

In mintabmode in order to make use of extra columns in the align environment (which we will use shortly in order to write protocols) you can use \> as you would use & normally. But, don't forget that there is an alignment tab already placed at the beginning and end of each line. So the following example



is generated by

```
\pseudocode{%
 \textbf{Alice} \> \> \textbf{Bob}  \\
 b \sample \bin \> \> \\
 \> \xrightarrow{\text{send over } b} \>  \\
 \> \> \text{do something}}
```

In Chapter 5 we'll discuss how to write protocols in detail. The next two sections are rather technical, so feel free to skip them.

4.1.1 Overriding The Tabbing Character

If you don't like \> as the tabbing character you can choose a custom command by overwriting \pctabname. For example

```
\renewcommand{\pctabname}{\myTab}

\pseudocode{%
 \textbf{Alice} \myTab \myTab \textbf{Bob}  \\
 b \sample \bin \myTab \myTab \\
 \myTab \xrightarrow{\text{send over } b} \myTab  \\
 \myTab \myTab \text{do something}}
```

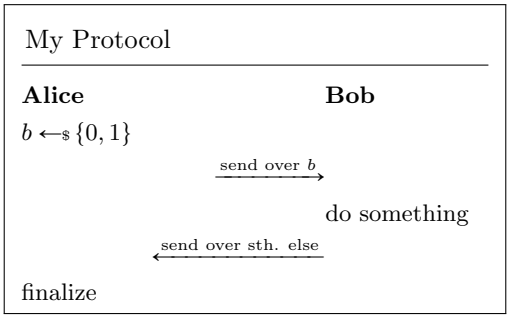
4.1.2 Custom Line Spacing and Horizontal Rules

As explained underlying the pseudocode command is an flalign environment. This would allow the use of \\[spacing] to specifiy the spacing between two lines or of [\\\hline] to insert a horizontal rule. In order to achieve the same effect within the pseudocode command you can use \\[spacing][\hline]. You can also use \pclb to get a line break which does not insert the additional alignment characters.

Chapter 5

Protocols

The pseudocode package can also be used to write protocols such as

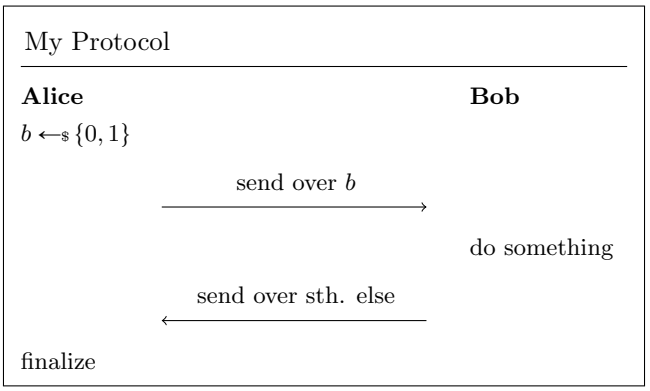


which uses the tabbing feature of align and is generated as

```
\procedure{My Protocol}{%
 \textbf{Alice} \> \> \textbf{Bob}  \\
 b \sample \bin \> \> \\
 \> \xrightarrow{\text{send over } b} \>  \\
 \> \> \text{do something}     \\
 \> \xleftarrow{\text{send over sth. else}} \>   \\
 \text{finalize} \> \>}
}
```

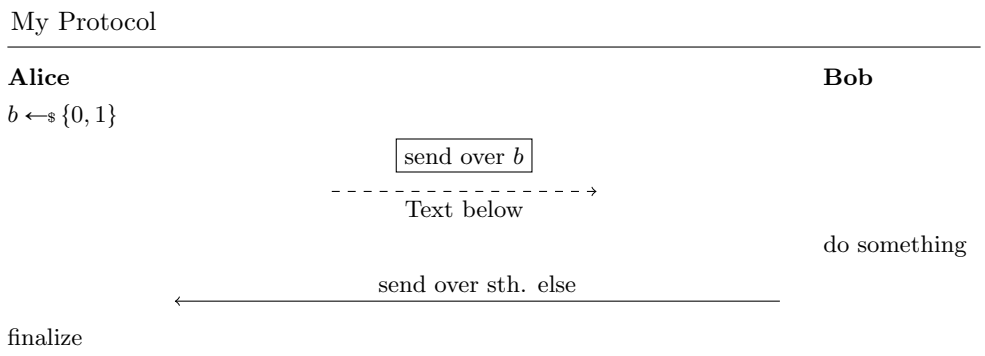
In order to get nicer message arrows use the commands \sendmessageright*{message} and \sendmessageleft*{message}. Both take an additional optional argument specifying the length of the arrow and both are run in math mode.

```
\sendmessageright*[3.5cm]{message}
\sendmessageleft*[3.5cm]{message}
```



```
\procedure{My Protocol}{%
 \textbf{Alice} \> \> \textbf{Bob}  \\
 b \sample \bin \> \> \\
 \> \sendmessageright*{\text{send over } b} \>  \\
 \> \> \text{do something}    \\
 \> \sendmessageleft*{\text{send over sth. else}} \>   \\
 \text{finalize} \> \> }
```

Besides the starred version there is also the unstarred version which allows more flexibility. Note that a crucial difference between the starred and unstarred versions are that `\sendmessageleft*{message}` wraps an aligned environment around the message.




```
\procedure{My Protocol}{%
 \textbf{Alice} \> \> \textbf{Bob}  \\
 b \sample \bin \> \> \\
 \> \sendmessageright{centercol=3,top=send over $b$,bottom=Text below,topstyle={draw,solid,
    yshift=0.25cm},style={dashed}} \>  \\
 \> \> \text{do something}    \\
 \> \sendmessageleft{length=8cm,top=send over sth. else} \>   \\
 \text{finalize} \> \> }
```

The unstarred commands take key-value pairs. The following keys are available:

top The content to display on top of the arrow.

bottom The content to display below the arrow.

left The content to display on the left of the arrow.

right The content to display on the right of the arrow.

topstyle The TIKZ style to be used for the top node.

bottomstyle The TIKZ style to be used for the bottom node.

rightstyle The TIKZ style to be used for the right node.

leftstyle The TIKZ style to be used for the left node.

length The length of the arrow.

style The style of the arrow.

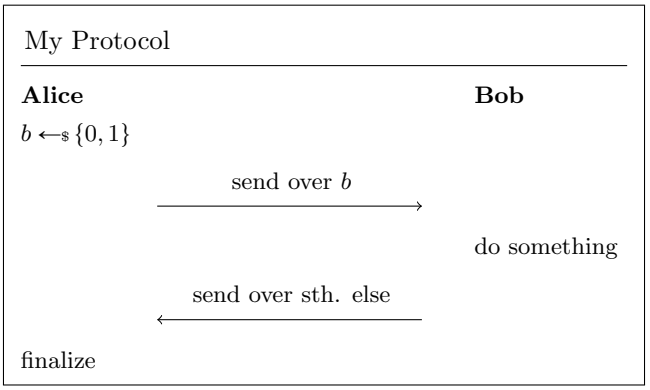
width The width of the column

centercol Can be used to ensure that the message is displayed in the center. This should be set to the column index. In the above example, the message column is the third column (note that there is a column left of alice that is automatically inserted.).

5.1 Tabbing

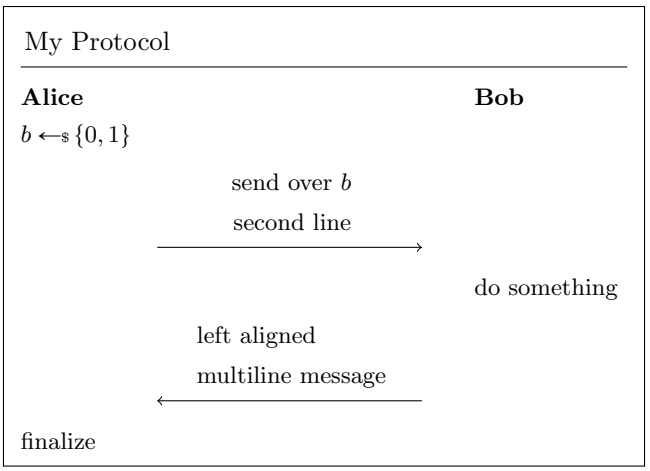
When typesetting protocols you might find that using two tabs instead of a single tab usually provides a better result as this ensures that all columns are left aligned. For this you can use \< instead of \> (see Chapter 4).

Following is once more the example from before but now with double tapping.



5.2 Multiline Messages

Using the send message commands you can easily generate multiline messages as the command wraps an *aligned* environment around the message.

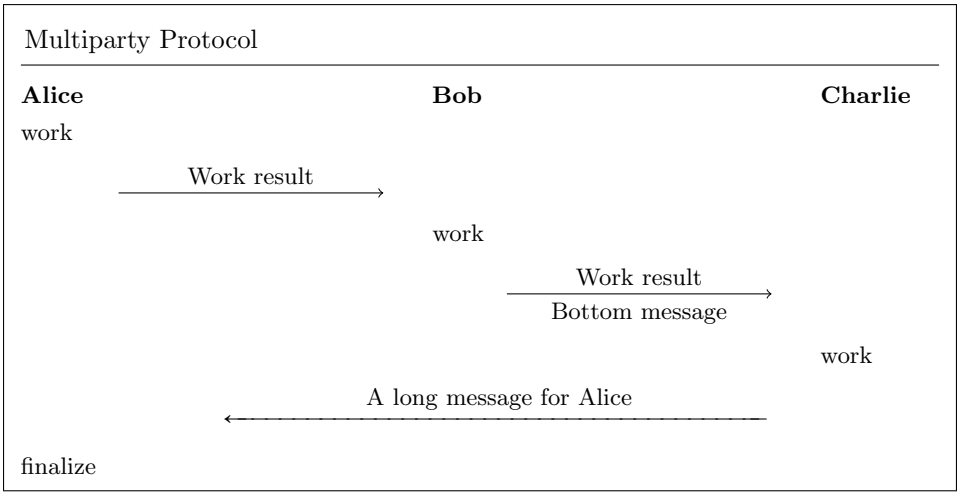



```
\procedure{My Protocol}{%
 \textbf{Alice} \< \< \textbf{Bob}  \\
 b \sample \bin \< \< \\
 \< \sendmessageright*{\text{send over } b\\ \text{second line}} \<  \\
 \< \< \text{do something}    \\
 \< \sendmessageleft*{&\text{left aligned}\\ &\text{multiline message}} \<   \\
 \text{finalize} \< \<}
```

5.2.1 Multiplayer Protocols

You are not limited to two players. In order to send messages skipping players use \sendmessagerightx and \sendmessageleftx.

```
\sendmessagerightx[width]{columnspan}{Text}
\sendmessageleftx[width]{columnspan}{Text}
```




```
\begin{center}
\procedure{Multiparty Protocol}{%
 \textbf{Alice} \< \< \textbf{Bob} \< \< \textbf{Charlie} \\
 \text{work} \< \< \< \< \\
 \< \sendmessageright{top=Work result} \< \< \< \\
 \< \< \text{work} \< \< \\
 \< \< \<  \sendmessageright{top=Work result,bottom=Bottom message} \< \\
 \< \<  \< \< \text{work} \\
 \< \sendmessageleftx[7cm]{8}{\text{A long message for Alice}} \< \\
 \text{finalize} \< \< \< \< }
```

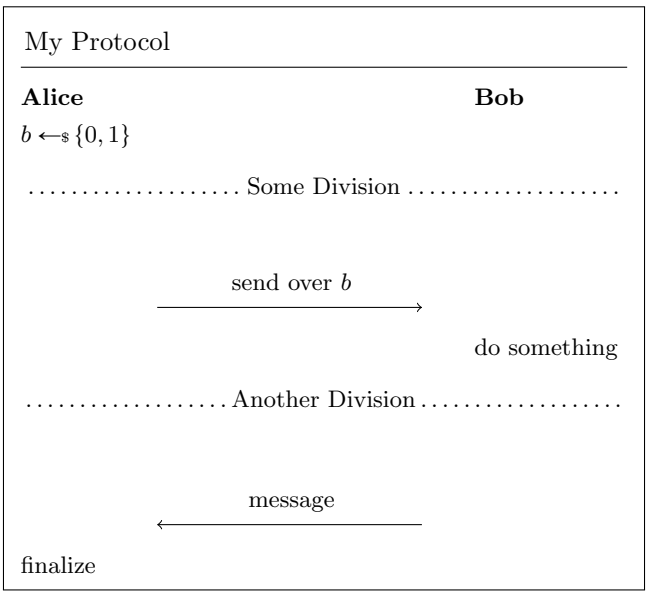
Note that for the last message from Charlie to Alice we needed to specify the number of passed over colums (\sendmessageleftx[7cm]{8}{message}). As we were passing 4 \< where each creates 2 columns, the total was 8 columns.

5.2.2 Divisions

You can use \pcintertext in order to divide protocols (or other pseudocode for that matter).

```
\pcintertext[dotted|center]{Division Text}
```

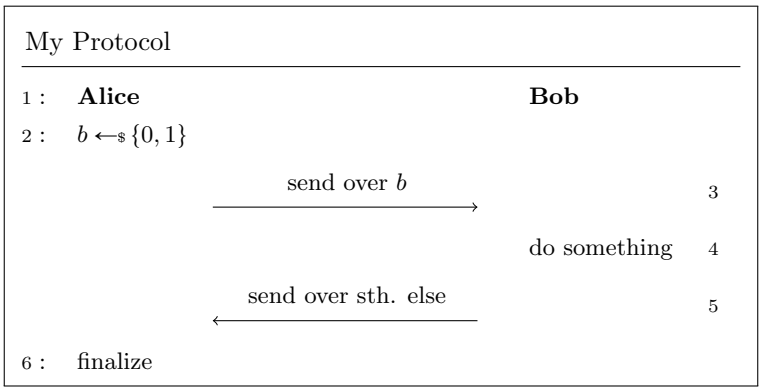
Note that in order to use the \pcintertext you need to use \pclb as the line break for the line before. Also see Chapter 4.



```
\procedure{My Protocol}{%
\textbf{Alice} \< \< \textbf{Bob}  \\
 b \sample \bin \< \<  \pclb
 \pcintertext[dotted]{Some Division} \\
 \< \sendmessageright*{\text{send over } b} \<  \\
 \< \< \text{do something} \pclb
 \pcintertext[dotted]{Another Division} \\
 \< \sendmessageleft*{\text{message}} \<  \\
 \text{finalize} \< \< }
```

5.3 Line Numbering in Protocols

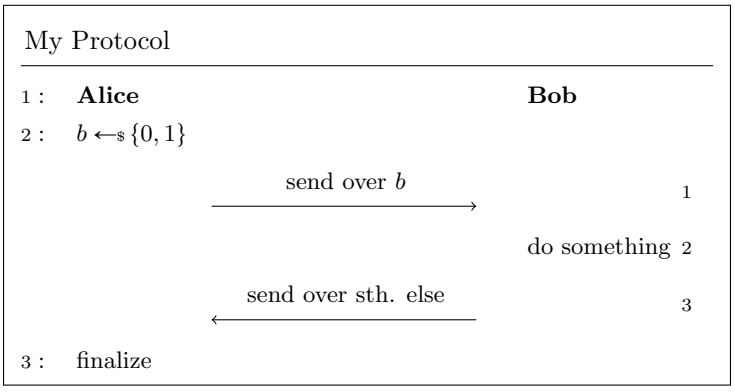
Protocols can be numbered similarly to plain pseudocode. Additionally to the \pcln there are the commands \pclnr and \pcrln. The first allows you to right align line numbers but uses the same counter as \pcln. The second uses a different counter.



Which is generated as

```
\procedure{My Protocol}{%
\pcln  \textbf{Alice} \< \< \textbf{Bob} \< \\
\pcln b \sample \bin \< \< \< \\
 \< \sendmessageright*{\text{send over } b} \< \< \pclnr\\
 \< \< \text{do something} \<  \pclnr \\
 \< \sendmessageleft*{\text{send over sth. else}} \<\<  \pclnr \\
\pcln \text{finalize} \< \< \<}
```

And using \pcrln:



Which is generated as

```
\procedure{My Protocol}{%
\pcln  \textbf{Alice} \< \< \textbf{Bob}  \\
\pcln b \sample \bin \< \< \\
\< \sendmessageright*{\text{send over } b} \<  \pcrln\\
```

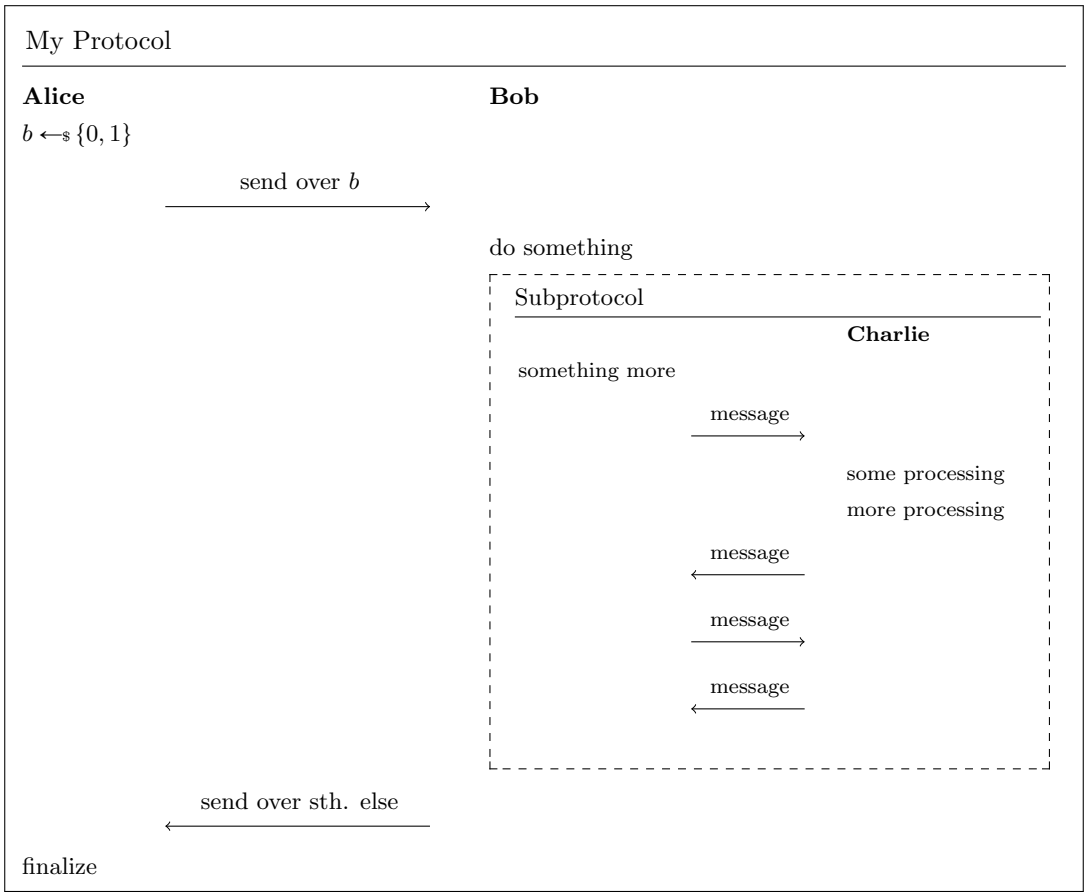
```
\< \< \text{do something}   \pcrln \\
\< \sendmessageleft*{\text{send over sth. else}} \<  \pcrln \\
\pcln \text{finalize} \< \< }
```

5.3.1 Separators

The commands \pclnseparator and \pcrlnseparator define the separators between the pseudocode and line numbering. By default the left separator is set to (:) colon and the right separator is set to a space of 3 pt.

5.4 Sub Protocols

Use the "subprocedure" function also to create sub protocols.




```
\procedure{My Protocol}{%
 \textbf{Alice} \< \< \textbf{Bob}  \\
 b \sample \bin \< \< \\
 \< \sendmessageright*{\text{send over } b} \<  \\
 \< \< \text{do something}    \\
 \<\< \dbox{\begin{subprocedure}\procedure{Subprotocol}{
 \< \< \textbf{Charlie}  \\
\text{something more} \< \<  \\
 \< \sendmessageright*[1.5cm]{\text{message}} \<  \\
 \< \< \text{some processing}  \\
 \< \< \text{more processing}  \\
 \< \sendmessageleft*[1.5cm]{\text{message}} \<  \\
 \< \sendmessageright*[1.5cm]{\text{message}} \<  \\
 \< \sendmessageleft*[1.5cm]{\text{message}} \<  \\
}\end{subprocedure}}  \\
 \< \sendmessageleft*{\text{send over sth. else}} \<  \\
```

```
17  \text{finalize} \< \< }
```

Chapter 6

Game Based Proofs

6.1 Basics

Besides displaying pseudocode the package also comes with commands to display game based proofs. A proof is wrapped in the *gameproof* environment.

```
\begin{gameproof}
proof goes here
\end{gameproof}
```

Within the proof environment you can use the command \gameprocedure which works similarly to the pseudocode command and produces a heading of the form $\mathsf{Game}_{\text{counter}}$ where counter is a consecutive counter. Thus, we can create the following setup

$\mathsf{Game}_1(n)$	$\mathsf{Game}_2(n)$
1 : Step 1	Step 1
2 : Step 2	Step 2

by using

```
\begin{gameproof}
\gameprocedure[linenumbering,mode=text]{%
  Step 1  \\
  Step 2
  }
\gameprocedure[mode=text]{%
  Step 1  \\
  Step 2
  }
\end{gameproof}
```

6.1.1 Highlight Changes

In order to highlight changes from one game to the next use \gamechange.

$\mathsf{Game}_1(n)$	$\mathsf{Game}_2(n)$
1 : Step 1	Step 1
2 : Step 2	Step 2

```
\begin{gameproof}
\gameprocedure[linenumbering,mode=text]{%
  Step 1  \\
  Step 2
```

```
  }
\gameprocedure[mode=text]{%
  Step 1   \\
  \gamechange{Step 2}
  }
\end{gameproof}
```

6.1.2 Boxed games

Use \tbxgameprocedure in order to create two consecutive games where the second game is *boxed*. Use \pcbox to create boxed statements.

	$\mathsf{Game}_1(n)$	$\mathsf{Game}_2(n)$ $\boxed{\mathsf{Game}_3(n)}$	$\mathsf{Game}_4(n)$
1 :	Step 1	Step 1; Alternative step 1	Step 1
2 :	Step 2	Step 2 is different	Step 2

```
\begin{gameproof}
\gameprocedure{%
  \pcln   \text{Step 1}   \\
  \pcln   \text{Step 2}
  }
\tbxgameprocedure{%
    \text{Step 1}; \pcbox{\text{Alternative step 1}}   \\
    \gamechange{\text{Step 2 is different}}
  }
\gameprocedure{%
  \pcln   \text{Step 1}   \\
  \pcln   \text{\gamechange{Step 2}}
  }
\end{gameproof}
```

6.1.3 Reduction Hints

In a game based proof in order to go from one game to the next we usually give a reduction, for example, we show that the difference between two games is bound by the security of some pseudorandom generator PRG. To give a hint within the pseudocode that the difference between two games is down to "something" you can use the \addgamehop command.

```
\addgamehop{startgame}{endgame}{options}
```

Here options allows you to specify the hint as well as the style. The following options are available

hint The hint text

nodestyle A TIKZ style to be used for the node.

pathstyle A TIKZ style to be used for the path.

edgestyle A TIKZ style to be used for the edge. This defaults to "bend left".

some hint

	$\mathsf{Game}_1(n)$	$\mathsf{Game}_2(n)$
1 :	Step 1	Step 1
2 :	Step 2	Step 2 is different

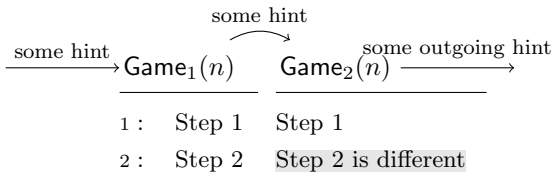
```
\begin{gameproof}
\gameprocedure{%
  \pcln  \text{Step 1}  \\
  \pcln  \text{Step 2}
  }
\gameprocedure{%
    \text{Step 1}  \\
    \gamechange{\text{Step 2 is different}}
  }
\addgamehop{1}{2}{hint=\footnotesize some hint,nodestyle=red}
\end{gameproof}
```

The edgestyle allows you to specify how the hint is displayed. If you, for example want a straight line, rather than the curved arrow simply use

```
\addgamehop{1}{2}{hint=\footnotesize some hint,edgestyle=}
```

If game proofs do not fit into a single picture you can specify start and end hints using the commands

```
\addstartgamehop[first game]{options}
\addendgamehop[last game]{options}
```

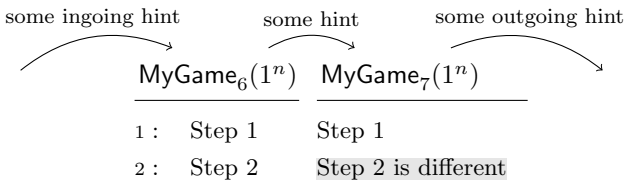



```
\begin{gameproof}
\gameprocedure{%
  \pcln  \text{Step 1}  \\
  \pcln  \text{Step 2}
  }
\gameprocedure{%
    \text{Step 1}  \\
    \gamechange{\text{Step 2 is different}}
  }
  \addstartgamehop{hint=\footnotesize some hint,edgestyle=}
\addgamehop{1}{2}{hint=\footnotesize some hint}
  \addendgamehop{hint=\footnotesize some outgoing hint,edgestyle=}
\end{gameproof}
```

6.1.4 Numbering and Names

By default the *gameproof* environment starts to count from 1 onwards. Its optional parameters allow you to specify a custom name for your game and the starting number.

```
\begin{gameproof}[options]
```




```
\begin{gameproof}[nr=5,name=$\mathsf{MyGame}$,arg=$(1^n)$]
\gameprocedure{%
  \pcln  \text{Step 1}  \\
  \pcln  \text{Step 2}
  }
```

```
\gameprocedure{%
   \text{Step 1}  \\
   \gamechange{\text{Step 2 is different}}
 }
\addstartgamehop{hint=\footnotesize some ingoing hint}
\addgamehop{6}{7}{hint=\footnotesize some hint}
 \addendgamehop{hint=\footnotesize some outgoing hint}
\end{gameproof}
```

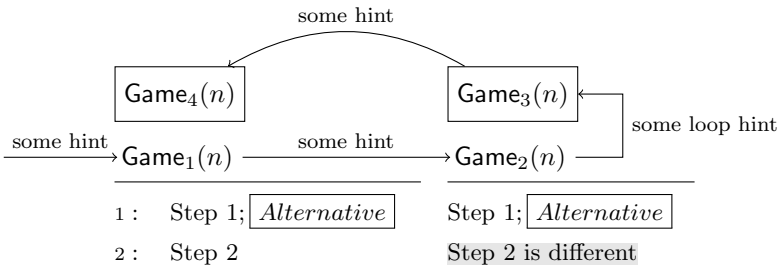
6.1.5 Default Name and Argument

The default name and argument are controlled via the commands \pcgamename and \gameprocedurearg.

Command	Default
\pcgamename	\mathsf{Game}
\gameprocedurearg	(\secpar)

6.1.6 Two Directional Games

You can use the \bxgameprocedure to generate games for going in two directions. Use the \addloopgamehop to add the gamehop in the middle.




```
\begin{gameproof}
\bxgameprocedure{4}{%
 \pcln  \text{Step 1}; \pcbox{Alternative} \\
 \pcln  \text{Step 2}
 }
\bxgameprocedure{3}{%
   \text{Step 1}; \pcbox{Alternative}  \\
   \gamechange{\text{Step 2 is different}}
 }
 \addstartgamehop{hint=\footnotesize some hint,edgestyle=}
\addgamehop{1}{2}{hint=\footnotesize some hint,edgestyle=}
 \addloopgamehop{hint=\footnotesize some loop hint}
 \addgamehop{2}{1}{hint=\footnotesize some hint}
\end{gameproof}
```

Chapter 7

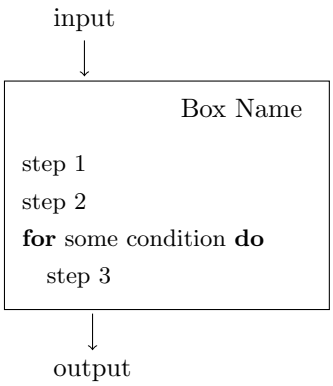
Black-box Reductions

The cryptocode package comes with support for drawing basic black box reductions. A reduction is always of the following form.

```
\begin{bbrenv}{A}
\begin{bbrbox}[name=Box Name]
% The Box's content
\end{bbrbox}
% Commands to display communication, input output etc
\end{bbrenv}
```

That is, a “bbrenv” (where bbr is short for black-box reduction) environment which takes a single “bbrbox” environment and some additional commands.

The following is a simple example drawing one (black)box with some code and input output:



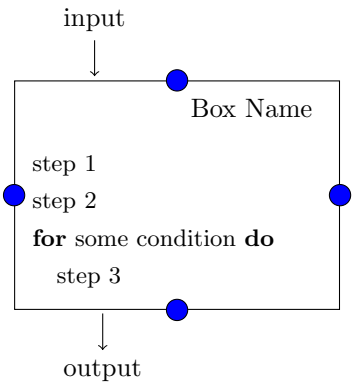
This box is generated as

```
\begin{bbrenv}{A}
  \begin{bbrbox}[name=Box Name]
  \pseudocode{
    \text{step 1} \\
    \text{step 2} \\
    \pcfor \text{some condition} \pcdo \\
    \pcind\text{step 3}
  }
  \end{bbrbox}
  \bbrinput{input}
  \bbroutput{output}
\end{bbrenv}
```

The commands bbrinput and bbroutput allow to specify input and output for the latest ”bbrenv environment. The single argument to the bbrenv environment needs to specify a unique identifier (unique for the current reduction). This id is used as an internal TIKZ node name (`http://www.ctan.org/tex-archive/graphics/pgf/`).

```
\begin{bbrenv}{UNIQUE IDENTIFIER}
```

As we are drawing a TIKZ image, note that we can easily later customize the image using the labels that we have specified on the way.




```
\begin{bbrenv}{A}
  \begin{bbrbox}[name=Box Name]
  \pseudocode{
    \text{step 1} \\
    \text{step 2} \\
    \pcfor \text{some condition} \pcdo \\
    \pcind\text{step 3}
  }
  \end{bbrbox}
  \bbrinput{input}
  \bbroutput{output}

  \filldraw[fill=blue] (A.north) circle (4pt);
  \filldraw[fill=blue] (A.west) circle (4pt);
  \filldraw[fill=blue] (A.east) circle (4pt);
  \filldraw[fill=blue] (A.south) circle (4pt);
\end{bbrenv}
```

The “bbrbox” takes as single argument a comma separated list of key value pairs. In the example we have used

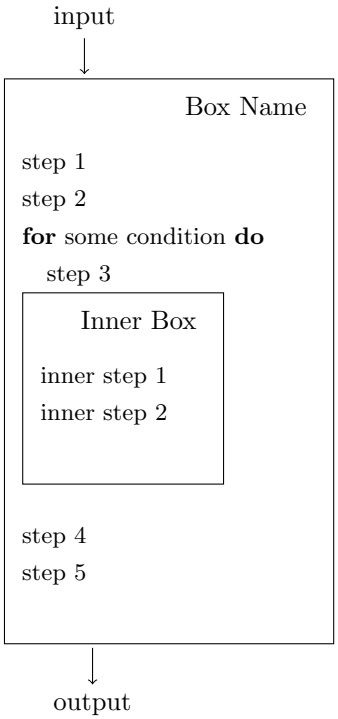
```
name=Box Name
```

to specify the label. The following options are available

Option	Description
name	Specifies the box’ label
minheight	The minimal height
xshift	Allows horizontal positioning
style	allows to customize the node

7.1 Nesting of Boxes

Boxes can be nested. For this simply insert a bbrenv (together with a single bbrbox) environment into an existing bbrbox.




```
\begin{bbrenv}{A}
  \begin{bbrbox}[name=Box Name]
  \pseudocode{
    \text{step 1} \\
    \text{step 2} \\
    \pcfor \text{some condition} \pcdo \\
    \pcind\text{step 3}
  }

  \begin{bbrenv}{B}
    \begin{bbrbox}[name=Inner Box]
    \pseudocode{
      \text{inner step 1} \\
      \text{inner step 2} \\
    }
    \end{bbrbox}
  \end{bbrenv}

  \pseudocode{
    \text{step 4} \\
    \text{step 5} \\
  }
  \end{bbrbox}
  \bbrinput{input}
  \bbroutput{output}
\end{bbrenv}
```

7.2 Messages and Queries

You can send messages and queries to boxes. For this use the commands

```
\bbrmsgto{options}
\bbrmsgfrom{options}
\bbrqryto{options}
\bbrqryfrom{options}
```

By convention messages are on the left of boxes and queries on the right. Commands ending on to make an arrow to the right while commands ending on from make an arrow to the left. The *options* define how the message is drawn and consists of a key-value pairs separated by ",".

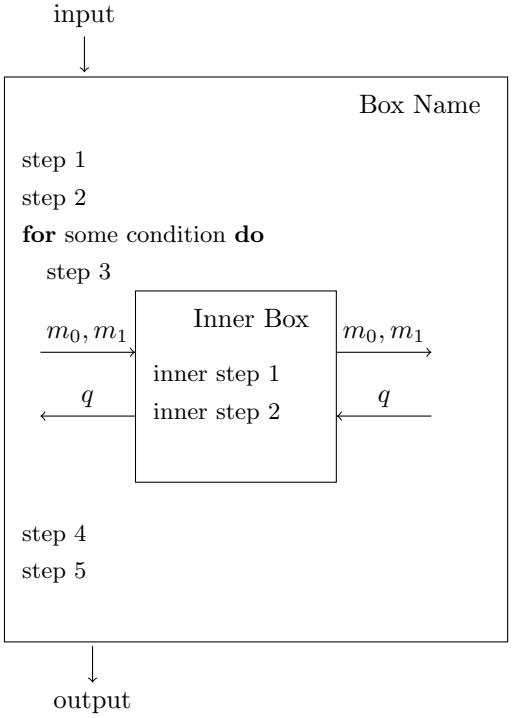
For example, to draw a message with a label on top and on the side use

```
\bbrmsgto{top=Top Label, side=Side Label}
```

If your label contains a "," (comma), then group the label in {} (curly brackets).

```
\bbrmsgto{top=Top Label, side={Side, Label}}
```

Following is a complete example. Notice that cryptocode takes care of the vertical positioning.




```
\begin{bbrenv}{A}
  \begin{bbrbox}[name=Box Name]
  \pseudocode{
    \text{step 1} \\
    \text{step 2} \\
    \pcfor \text{some condition} \pcdo \\
    \pcind\text{step 3}
  }

  \begin{bbrenv}{B}
    \begin{bbrbox}[name=Inner Box]
    \pseudocode{
      \text{inner step 1} \\
      \text{inner step 2} \\
    }
    \end{bbrbox}

    \bbrmsgto{top={$m_0,m_1$}}
    \bbrmsgfrom{top=$q$}


    \bbrqryto{top={$m_0,m_1$}}
    \bbrqryfrom{top=$q$}

  \end{bbrenv}

  \pseudocode{
    \text{step 4} \\
    \text{step 5} \\
  }
  \end{bbrbox}
  \bbrinput{input}
  \bbroutput{output}
```

```
\end{bbrenv}
```

7.2.1 Options

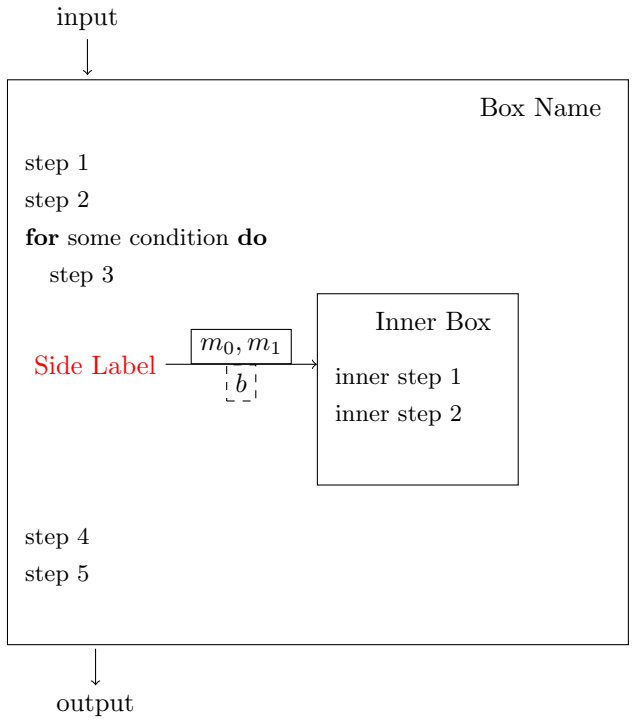
Besides specifying labels for top, side and bottom you can further specify how cryptocode renders the message. Remember that tnderneath the reduction commands is a TIKZ image (`http://www.ctan.org/tex-archive/graphics/pgf/`). For each label position (top, side, bottom) a node is generated. You can provide additional properties for this node using the options:

- topstyle
- sidestyle
- bottomstyle

You can additionally provide custom names for the nodes for later reference using

- topname
- sidename
- osidename
- bottomname

The "osidename" allows you to provide a name for the "other side".
Via the option "length" you can specifiy the length of the arrow.

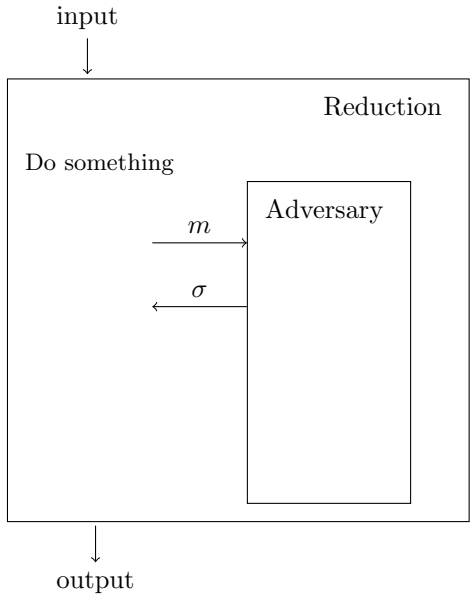



```
\begin{bbrenv}{A}
  \begin{bbrbox}[name=Box Name]
  \pseudocode{
    \text{step 1} \\
    \text{step 2} \\
    \pcfor \text{some condition} \pcdo \\
    \pcind\text{step 3}
  }

  \begin{bbrenv}{B}
```

```
    \begin{bbrbox}[name=Inner Box]
    \pseudocode{
      \text{inner step 1} \\
      \text{inner step 2} \\
    }
    \end{bbrbox}

    \bbrmsgto{top={$m_0,m_1$},side=Side Label, bottom=$b$, length=2cm,
          topstyle={draw, solid}, sidestyle={red}, bottomstyle={draw, dashed}}

  \end{bbrenv}

  \pseudocode{
    \text{step 4} \\
    \text{step 5} \\
  }
  \end{bbrbox}
  \bbrinput{input}
  \bbroutput{output}
\end{bbrenv}
```

7.2.2 Loops

Often an adversary may send poly many queries to an oracle, or a reduction sends many queries to an adversary. Consider the following setting

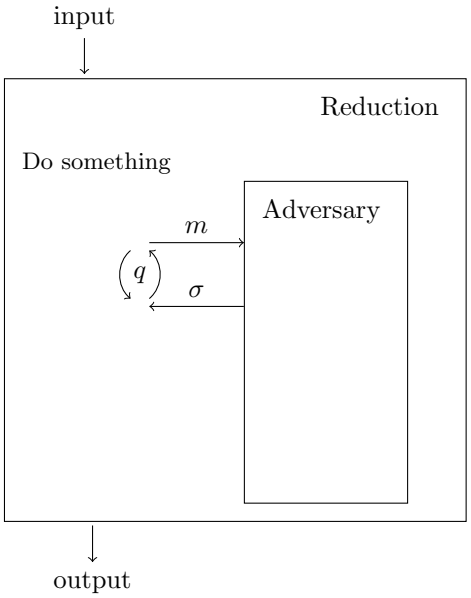



```
\begin{bbrenv}{A}
  \begin{bbrbox}[name=Reduction]
  \pseudocode{
    \text{Do something}
  }

  \begin{bbrenv}{B}

    \begin{bbrbox}[name=Adversary,minheight=3cm,xshift=4cm]

    \end{bbrbox}

    \bbrmsgto{top=$m$}
    \bbrmsgfrom{top=$\sigma$}

  \end{bbrenv}

  \end{bbrbox}
  \bbrinput{input}
  \bbroutput{output}
\end{bbrenv}
```

First note that by specifying the minheight and xshift option we shifted the adversary box a bit to the right and enlarged its box. Further we specified custom names for the node on the side of the two messages. We can now use the bbrloop command to visualize that these two messages are exchanged q many times

```
\bbrloop{BeginLoop}{EndLoop}{center=$q$}
```

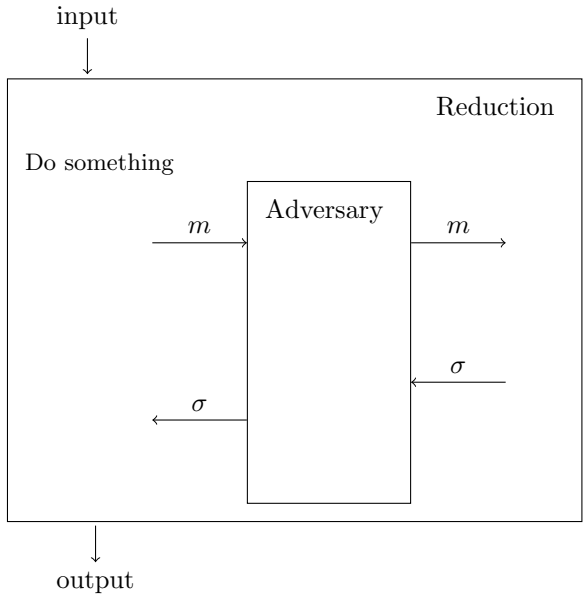
The bbrloop command takes two node names and a config which allows you to specify if the label is to be shown on the left, center or right. Here is the result.




```
\begin{bbrenv}{A}
  \begin{bbrbox}[name=Reduction]
  \pseudocode{
    \text{Do something}
  }

  \begin{bbrenv}{B}

    \begin{bbrbox}[name=Adversary,minheight=3cm,xshift=4cm]

    \end{bbrbox}

    \bbrmsgto{top=$m$,sidename=BeginLoop}
    \bbrmsgfrom{top=$\sigma$,sidename=EndLoop}
    \bbrloop{BeginLoop}{EndLoop}{center=$q$}

  \end{bbrenv}

  \end{bbrbox}
  \bbrinput{input}
  \bbroutput{output}
\end{bbrenv}
```

7.2.3 Add Space

If the spacing between messages is not sufficient you can use the bbrmsgspace and bbrqryspace commands to add additional space.

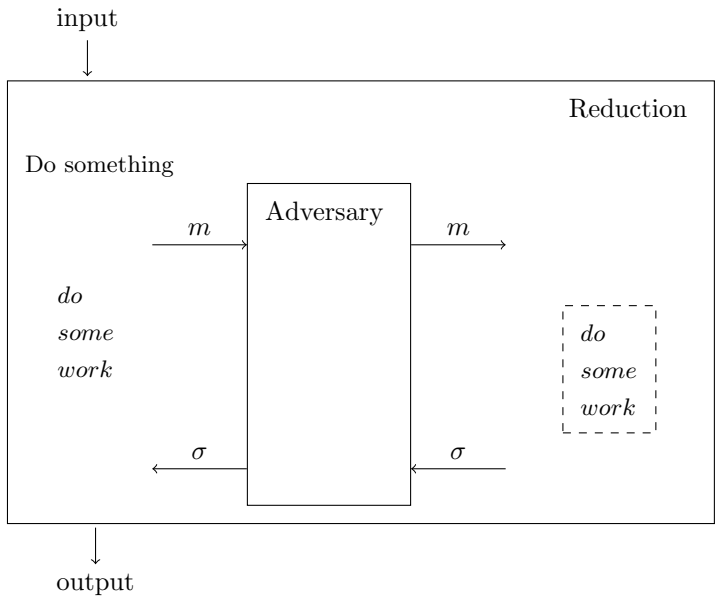



```
\begin{bbrenv}{A}
  \begin{bbrbox}[name=Reduction]
  \pseudocode{
    \text{Do something}
  }

  \begin{bbrenv}{B}

    \begin{bbrbox}[name=Adversary, minheight=3cm, xshift=4cm]

    \end{bbrbox}

    \bbrmsgto{top=$m$}
    \bbrmsgspace{1.5cm}
    \bbrmsgfrom{top=$\sigma$}

    \bbrqryto{top=$m$}
    \bbrqryspace{1cm}
    \bbrqryfrom{top=$\sigma$}

  \end{bbrenv}

  \end{bbrbox}
  \bbrinput{input}
  \bbroutput{output}
\end{bbrenv}
```

7.2.4 Intertext

If your reduction needs to do some extra work between queries use the \bbrmsgtxt and \bbrqrytxt commands.

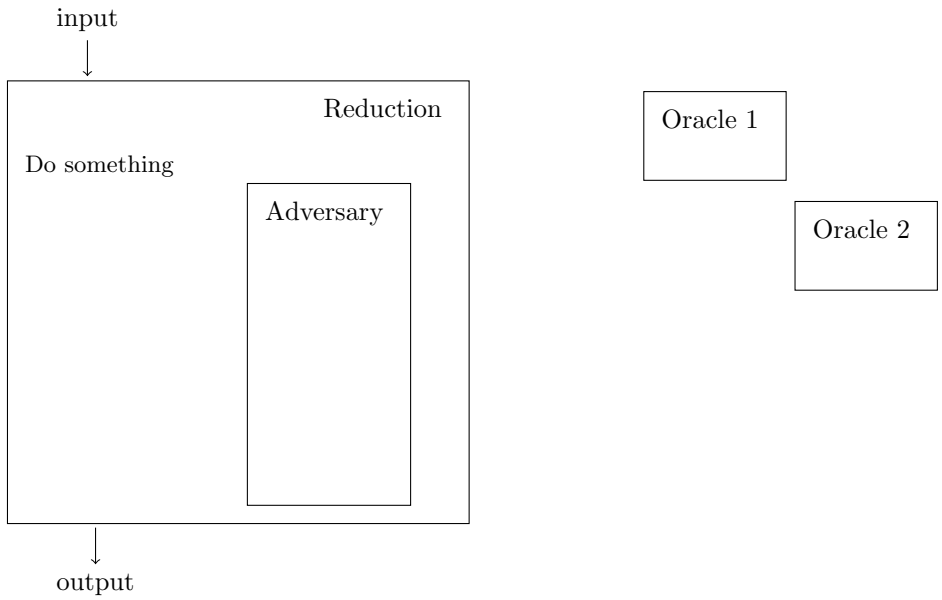
```
\bbrmsgtxt[options]{Text}
\bbrqrytxt[options]{Text}
```




```
\begin{bbrenv}{A}
  \begin{bbrbox}[name=Reduction]
  \pseudocode{
    \text{Do something}
  }

  \begin{bbrenv}{B}

    \begin{bbrbox}[name=Adversary, minheight=3cm, xshift=4cm]

    \end{bbrbox}

    \bbrmsgto{top=$m$}
    \bbrmsgtxt{\pseudocode{
      do \\
      some \\
      work
    }}
    \bbrmsgfrom{top=$\sigma$}

    \bbrqryto{top=$m$}
    \bbrqrytxt[beforeskip=0.5cm, nodestyle={draw, dashed}, xshift=2cm]{\pseudocode{
      do \\
      some \\
      work
    }}
    \bbrqryfrom{top=$\sigma$}

  \end{bbrenv}

  \end{bbrbox}
  \bbrinput{input}
  \bbroutput{output}
\end{bbrenv}
```

7.3 Oracles

Each box can have one or more oracles which are drawn on the right hand side of the box. An oracle is created similarly to a *bbrenv* environment using the *bbroracle* environment. Oracles go behind the single *bbrbox* environment within an *bbrenv* enviornment.




```
\begin{bbrenv}{A}
  \begin{bbrbox}[name=Reduction]
  \pseudocode{
    \text{Do something}
  }

  \begin{bbrenv}{B}
    \begin{bbrbox}[name=Adversary, minheight=3cm, xshift=4cm]
    \end{bbrbox}

  \end{bbrenv}

  \end{bbrbox}
  \bbrinput{input}
  \bbroutput{output}

  \begin{bbroracle}{OraA}
    \begin{bbrbox}[name=Oracle 1]
    \end{bbrbox}
  \end{bbroracle}

  \begin{bbroracle}{OraB}
    \begin{bbrbox}[name=Oracle 2]
    \end{bbrbox}
  \end{bbroracle}
\end{bbrenv}
```

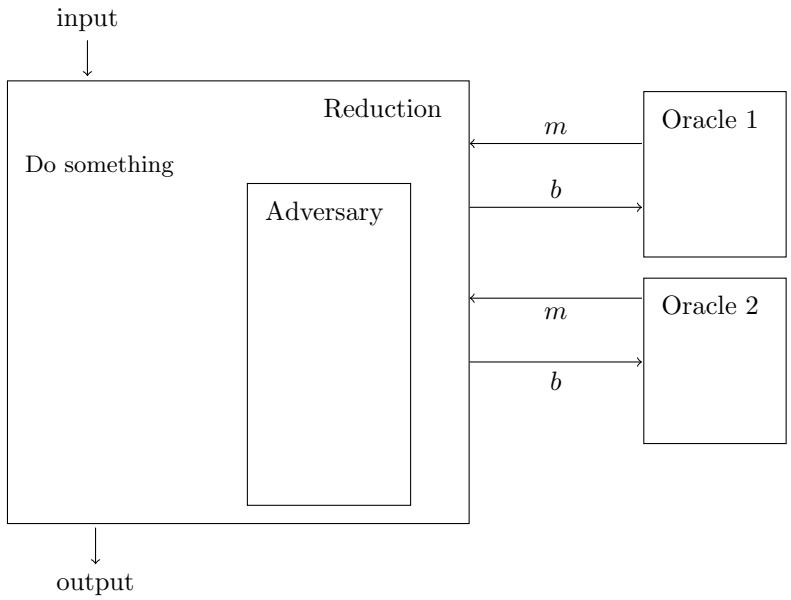
Via the option "distance=length" you can control the horizontal position of the oracle. By default this value is set to 1cm.

7.3.1 Communicating with Oracles

As oracles use the *bbrbox* environment we can directly use the established ways to send messages and queries to oracles. In addition you can use the \bbroraclequeryfrom and \bbroraclequeryto.

```
\bbroraclequeryfrom{options}
\bbroraclequeryto{options}
```

Here options allow you to specify where the label goes (top, bottom). In addition you can use \bbroracleqryspace to generate extra space between oracle messages. Note that oracle messages need to be added after the closing \end{bbroracle} command.




```
\begin{bbrenv}{A}
  \begin{bbrbox}[name=Reduction]
  \pseudocode{
    \text{Do something}
  }

  \begin{bbrenv}{B}
    \begin{bbrbox}[name=Adversary,minheight=3cm,xshift=4cm]
    \end{bbrbox}

  \end{bbrenv}

  \end{bbrbox}
  \bbrinput{input}
  \bbroutput{output}

  \begin{bbroracle}{OraA}
    \begin{bbrbox}[name=Oracle 1,minheight=1cm]
    \end{bbrbox}
  \end{bbroracle}
  \bbroraclequeryfrom{top=$m$}
  \bbroraclequeryto{top=$b$}

  \begin{bbroracle}{OraB}
    \begin{bbrbox}[name=Oracle 2,minheight=1cm]
    \end{bbrbox}
  \end{bbroracle}
  \bbroraclequeryfrom{bottom=$m$}
  \bbroraclequeryto{bottom=$b$}
\end{bbrenv}
```

Chapter 8

Known Issues

8.1 Pseudocode KeepSpacing within Commands

The "space=keep" option of pseudocode which should output spacing identical to that of the input will fail, if the pseudocode command is called from within another command. An example is to wrap the \pseudocode command with an \fbox. As a workaround for generating frame boxes you should hence use a package such as *mdframed* (`https://www.ctan.org/pkg/mdframed`) which provides a frame environment.

Pseudocode with - spaces -

```
\pseudocode[space=keep,mode=text]{ Pseudocode with - spaces -}
```

As an alternative you could use a *savebox* (in combination with the lrbox environment):

Pseudocode with - spaces -

```
\newsavebox{\mypcbox}
\begin{lrbox}{\mypcbox}%
\pseudocode[space=keep,mode=text]{ Pseudocode with - spaces -}%
\end{lrbox}
\fbox{\usebox{\mypcbox}}
```

8.2 AMSFonts

Some packages are not happy with the "amsfonts" package. Cryptocode will attempt to load amsfonts if it is loaded with either the "sets" or the "probability" option. In order to not load amsfonts you can additionally add the "noamsfonts" at the very end. Note that in this case you should ensure that the command \mathbb is defined as this is used by most of the commands in "sets" and some of the commands in "probability".

Index