

# Cryptocode

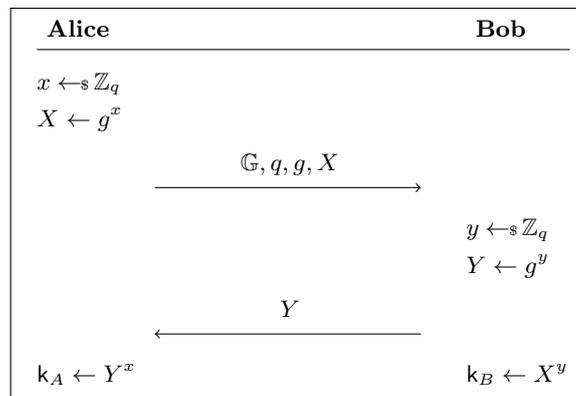
TYPESETTING CRYPTOGRAPHY

Arno Mittelbach  
mail@arno-mittelbach.de

April 22, 2015

### Abstract

The cryptocode package is targeted at cryptographers typesetting their results in L<sup>A</sup>T<sub>E</sub>X. 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

<b>1</b>	<b>Cryptocode by Example</b>	<b>1</b>
1.1	Pseudocode . . . . .	1
1.2	Columns . . . . .	3
1.3	Protocols . . . . .	3
1.4	Game-based Proofs . . . . .	4
1.5	Black-box Reductions . . . . .	5
<b>2</b>	<b>Cryptographic Notation</b>	<b>7</b>
2.1	Security Parameter . . . . .	7
2.2	Advantage Terms . . . . .	7
2.3	Math Operators . . . . .	8
2.4	Adversaries . . . . .	8
2.5	Landau . . . . .	8
2.6	Probabilities . . . . .	9
2.7	Sets . . . . .	10
2.8	Crypto Notions . . . . .	10
2.9	Logic . . . . .	10
2.10	Function Families . . . . .	11
2.11	Machine Model . . . . .	11
2.12	Crypto Primitives . . . . .	11
2.13	Events . . . . .	12
2.14	Complexity . . . . .	12
2.15	Asymptotics . . . . .	13
2.16	Keys . . . . .	13
<b>3</b>	<b>Pseudocode</b>	<b>14</b>
3.1	Basics . . . . .	14
3.1.1	Customizing Pseudocode . . . . .	14
3.1.2	Customized Pseudocode Commands . . . . .	16
3.2	Indentation . . . . .	16
3.3	Textmode . . . . .	19
3.4	Syntax Highlighting . . . . .	19
3.4.1	Alternative Keywords . . . . .	22
3.4.2	Draft Mode . . . . .	22
3.5	Line Numbering . . . . .	22
3.5.1	Manually Inserting Line Numbers . . . . .	23
3.5.2	Start Values . . . . .	23
3.5.3	Separators . . . . .	24
3.6	Subprocedures . . . . .	24
3.6.1	Numbering in Subprocedures . . . . .	24
3.7	Stacking Procedures . . . . .	25
3.8	Divisions and Linebreaks . . . . .	27
3.9	Fancy Code with Overlays . . . . .	28
3.9.1	Example: Explain your Code . . . . .	29

<b>4</b>	<b>Tabbing Mode</b>	<b>30</b>
4.1	Tabbing in Detail . . . . .	31
4.1.1	Overriding The Tabbing Character . . . . .	31
4.1.2	Custom Line Spacing and Horizontal Rules . . . . .	31
<b>5</b>	<b>Protocols</b>	<b>32</b>
5.1	Tabbing . . . . .	34
5.2	Multiline Messages . . . . .	34
5.2.1	Multiplayer Protocols . . . . .	34
5.2.2	Divisions . . . . .	35
5.3	Line Numbering in Protocols . . . . .	36
5.3.1	Separators . . . . .	37
5.4	Sub Protocols . . . . .	37
<b>6</b>	<b>Game Based Proofs</b>	<b>38</b>
6.1	Basics . . . . .	38
6.1.1	Highlight Changes . . . . .	38
6.1.2	Boxed games . . . . .	39
6.1.3	Reduction Hints . . . . .	39
6.1.4	Numbering and Names . . . . .	40
6.1.5	Default Name and Argument . . . . .	41
6.1.6	Two Directional Games . . . . .	41
<b>7</b>	<b>Black-box Reductions</b>	<b>42</b>
7.1	Nesting of Boxes . . . . .	43
7.2	Messages and Queries . . . . .	44
7.2.1	Options . . . . .	46
7.2.2	Add Space . . . . .	47
7.2.3	Loops . . . . .	47
7.2.4	Intertext . . . . .	49
7.3	Oracles . . . . .	50
7.3.1	Communicating with Oracles . . . . .	51
7.4	Challengers . . . . .	52
7.4.1	Communicating with Challengers . . . . .	53
<b>8</b>	<b>Known Issues</b>	<b>55</b>
8.1	Pseudocode KeepSpacing within Commands . . . . .	55
8.2	AMSFonTS . . . . .	55
8.3	Hyperref . . . . .	55

# Chapter 1

## Cryptocode by Example

Cryptocode is a  $\LaTeX$  package 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

```
1 \usepackage[
2   n,
3   advantage,
4   operators,
5   sets,
6   adversary,
7   landau,
8   probability,
9   notions,
10  logic,
11  ff,
12  mm,
13  primitives,
14  events,
15  complexity,
16  asymptotics,
17  keys
18 ]{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  $\backslash\text{pk}$  or  $\backslash\text{sk}$  (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  $\backslash\text{pseudocode}$  command takes a single parameter where you can start writing code in mathmode using  $\backslash\backslash$  as line breaks. Following is an IND-CPA game definition using various commands from cryptocode to ease writing keys ( $\backslash\text{pk}$ ,  $\backslash\text{sk}$ ), sampling ( $\backslash\text{sample}$ ), and more:

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

The above code is generated by (the code is actually wrapped in an `\fbbox`).

```

1 \pseudocode[linenumbering,syntaxhighlight=auto]{%
2   b \sample \bin \\\
3   (\pk,\sk) \sample \kgen (\seccparam) \\\
4   (\state,m_0,m_1) \sample \adv(\seccparam, \pk, c) \\\
5   c \sample \enc(\pk,m-b) \\\
6   b' \sample \adv(\seccparam, \pk, c, \state) \\\
7   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.

IND-CPA <sub>Enc</sub> <sup>A</sup>
1: $b \leftarrow \{0, 1\}$
2: $(pk, sk) \leftarrow \text{KGen}(1^n)$
3: $(state, m_0, m_1) \leftarrow \mathcal{A}(1^n, pk, c)$
4: $c \leftarrow \text{Enc}(pk, m_b)$
5: $b' \leftarrow \mathcal{A}(1^n, pk, c, state)$
6: <b>return</b> $b = b'$

```

1 \procedure[linenumbering]{\indcpa_\enc^\adv}{%
2   b \sample \bin \\\
3   (\pk,\sk) \sample \kgen (\seccparam) \\\
4   (\state,m_0,m_1) \sample \adv(\seccparam, \pk, c) \\\
5   c \sample \enc(\pk,m-b) \\\
6   b' \sample \adv(\seccparam, \pk, c, \state) \\\
7   \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.

IND-CPA <sub>Enc</sub> <sup>A</sup>
1: $b \leftarrow \{0, 1\}$
2: $(pk, sk) \leftarrow \text{KGen}(1^n)$
..... Setup Completed .....
3: $(m_0, m_1) \leftarrow \mathcal{A}(1^n, pk, c)$
4: $c \leftarrow \text{Enc}(pk, m_b)$
5: $b' \leftarrow \mathcal{A}(1^n, pk, c, state)$
6: <b>return</b> $b = b'$

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

```

1 \begin{pcimage}
2 \procedure[linenumbering]{\indcpa_\enc^\adv}{%
3   b \sample \bin \\\
4   (\pk,\sk) \sample \kgen (\seccparam)\pcnode{kgen} \pclb
5   \pcintertext[dotted]{Setup Completed}
6   (m_0,m_1) \sample \adv(\seccparam, \pk, c) \\\
7   c \sample \enc(\pk,m-b) \\\
8   b' \sample \adv(\seccparam, \pk, c, \state) \\\
9   \pcreturn b = b' }

```

```

10 \pcdraw{
11   \node[rectangle callout,callout absolute pointer=(kgen),fill=orange]
12     at ([shift={(+3,-1)}]kgen) {
13       \begin{varwidth}{3cm}
14         $\kgen(\secpam)$ samples a public key $\pk$ and a private key $\sk$.
15       \end{varwidth}
16     };
17 }
18 \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.<sup>1</sup>

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

```

1 \pseudocode{%
2   \textbf{First} \> \textbf{Second} \> \textbf{Third} \> \textbf{Fourth} \\\
3   b \sample \bin \> b \sample \bin \> b \sample \bin \> b \sample \bin}

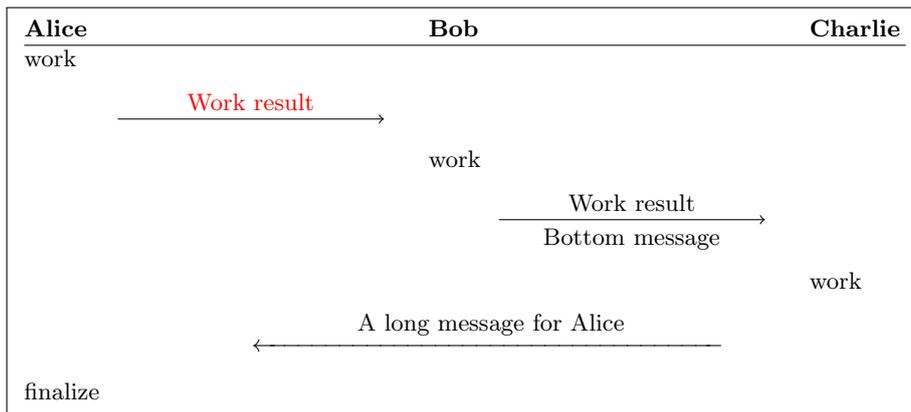
```

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
$b \leftarrow \{0, 1\}$			

## 1.3 Protocols

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



```

1 \pseudocode{%
2   \textbf{Alice} \< \< \textbf{Bob} \< \< \textbf{Charlie} \\\[ \hline ]
3   \text{work} \< \< \< \< \\\
4   \< \sendmessageright{top=Work result,topstyle=red} \< \< \< \\\
5   \< \< \text{work} \< \< \\\
6   \< \< \< \sendmessageright{top=Work result,bottom=Bottom message} \< \\\
7   \< \< \< \< \text{work} \\\

```

<sup>1</sup>In fact, the *pseudocode* command is based on `amsmath`'s `flalign` environment.



```

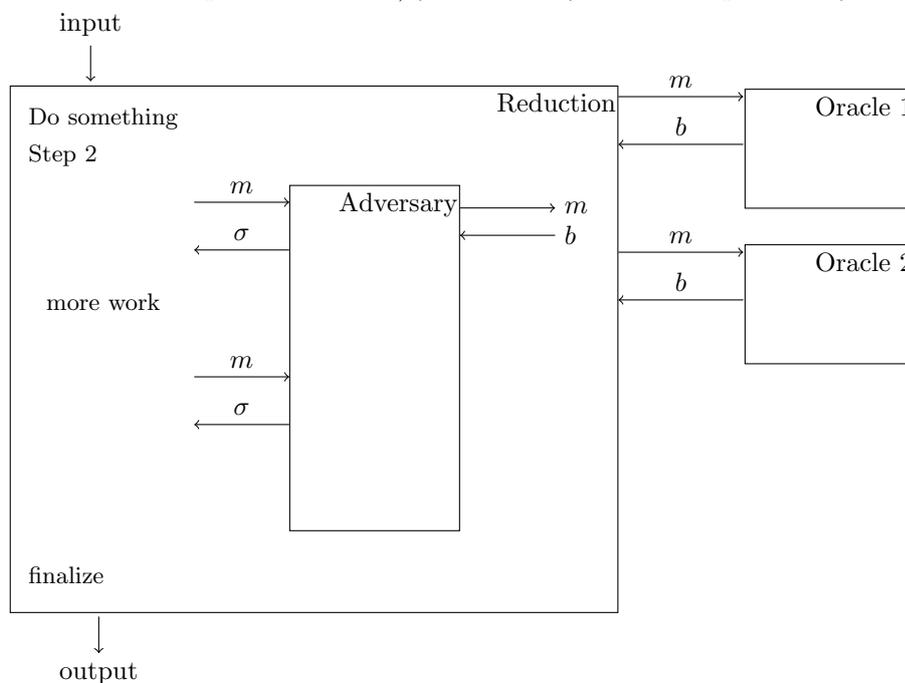
5 Step 3
6 }
7 \gameprocedure [mode=text] {%
8 Step 1 \\
9 \gamechange {Step 2 is different} \\
10 Step 3
11 }
12 \addgamehop {1}{2} {hint={\footnotesize some hint}}
13 \end {gameproof}

```

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 structured 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.



```

1 \begin {bbenv} {A}
2 \begin {bbrbox} [name=Reduction]
3 \pseudocode {
4 \text {Do something} \\
5 \text {Step 2}
6 }
7
8 \begin {bbenv} {B}
9 \begin {bbrbox} [name=Adversary , minheight=4cm]
10 \end {bbrbox}
11
12 \bbrmsgto {top=$n$}
13 \bbrmsgfrom {top=$\sigma$}
14 \bbrmsgtxt {\pseudocode {%
15 \text {more work}
16 }}
17 \bbrmsgto {top=$n$}
18 \bbrmsgfrom {top=$\sigma$}
19
20 \bbrqryto {side=$m$}

```

```

21 \bbrqryfrom{side=$b$}
22 \end{bbrenv}
23
24 \pseudocode{
25   \text{finalize}
26 }
27
28 \end{bbrbox}
29 \bbrinput{input}
30 \bbroutput{output}
31
32 \begin{bbroracle}{OraA}
33   \begin{bbrbox}[name=Oracle 1,minheight=1cm]
34   \end{bbrbox}
35 \end{bbroracle}
36 \bbroracleqryto{top=$m$}
37 \bbroracleqryfrom{top=$b$}
38
39 \begin{bbroracle}{OraB}
40   \begin{bbrbox}[name=Oracle 2,minheight=1cm]
41   \end{bbrbox}
42 \end{bbroracle}
43 \bbroracleqryto{top=$m$}
44 \bbroracleqryfrom{top=$b$}
45 \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.

```
1 \usepackage[
2   n,
3   advantage,
4   operators,
5   sets,
6   adversary,
7   landau,
8   probability,
9   notions,
10  logic,
11  ff,
12  mm,
13  primitives,
14  events,
15  complexity,
16  asymptotics,
17  keys
18 ]{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  $1^\lambda$ . The cryptocode package, when loading either option “n” or option “lambda” will define the commands

```
1 \secpa
2 \secpa
```

The first command provides the “letter”, i.e., either  $n$  or  $\lambda$ , whereas `\secpa` 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:

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

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

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

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

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

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

## 2.3 Math Operators

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

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

## 2.4 Adversaries

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

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

The style in which an adversary is rendered is controlled via

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

## 2.5 Landau

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

Command	Description	Result
<code>\bigO{n^2}</code>	Big O notation	$\mathcal{O}(n^2)$
<code>\smallO{n^2}</code>	small o notation	$\mathcal{o}(n^2)$
<code>\bigOmega{n^2}</code>	Big Omega notation	$\mathcal{\Omega}(n^2)$
<code>\bigsmallO{n^2}</code>	Big and small O notation	$\mathcal{\Theta}(n^2)$

## 2.6 Probabilities

The “probability” option provides commands for writing probabilities. Use

```
1 \prob{X=x}
2 \probsub{x\sample{\bin^n}}{x=5}
3 \condprob{X=x}{A=b}
4 \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

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

For expectations you can use

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

yielding

$$\mathbb{E}[X]$$

$$\mathbb{E}_{x,y \leftarrow \{1,\dots,6\}}[x + y]$$

$$\mathbb{E}[X + Y \mid Y > 3]$$

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

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

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

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

```
1 \supp{X}
```

where again the name can be controlled via

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

For denoting entropy and min-entropy use

```
1 \entropy{X}
2 \minentropy{X}
3 \condminentropy{X}{Y=5}
```

This yields

$$H(X)$$

$$H_\infty(X)$$

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

## 2.7 Sets

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

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

which is typeset as

$$\{1, \dots, 10\}$$

$$(1, \dots, 10)$$

In addition the following commands are provided

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

## 2.8 Crypto Notions

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

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

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

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

## 2.9 Logic

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

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

## 2.10 Function Families

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

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

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

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

## 2.11 Machine Model

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

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

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

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

## 2.12 Crypto Primitives

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

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

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

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

## 2.13 Events

The “events” option provides the following list of commands.

To classify an event use

```
1 \event{Event}
2 \nevent{Event}
```

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

$$\text{Event}$$

$$\overline{\text{Event}}$$

For bad events, use `\bad (bad)`.

## 2.14 Complexity

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

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

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

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

## 2.15 Asymptotics

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

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

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

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

## 2.16 Keys

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

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

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

```
1 \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

```

$$b \leftarrow \{0, 1\}$$

$$(\mathbf{pk}, \mathbf{sk}) \leftarrow \text{KGen}(1^n)$$

$$(m_0, m_1) \leftarrow \mathcal{A}(1^n, \mathbf{pk}, c)$$

$$c \leftarrow \text{Enc}(\mathbf{pk}, m_b)$$

$$b' \leftarrow \mathcal{A}(1^n, \mathbf{pk}, c)$$

$$\text{return } b = b'$$

```

which is generated as

```
1 \pseudocode{%
2   b \sample \bin \\
3   (\pk, \sk) \sample \kgen (\seccparam) \\
4   (m_0, m_1) \sample \adv(\seccparam, \pk, c) \\
5   c \sample \enc(\pk, m_b) \\
6   b' \sample \adv(\seccparam, \pk, c) \\
7   \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

```
1 \begin{center}
2 \fbox{%
3 Code
4 }
5 \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 (,).

```
1 \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 numbers when using line numbering.

**linenumbering** Enables line numbering.

**syntaxhighlight** When set to “auto” cryptocode will attempt to automatically highlight 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”.

**codesize** Allows to specify the fontsize for the pseudocode. Set to `\scriptsize` for a smaller size.

**xshift** Allows horizontal shifting

**yshift** Allows horizontal shifting

**headlinesep** Specifies the distance between header and the line.

**bodylinesep** Specifies the distance between body and the line.

**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

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

creates

Header
1: <b>return null</b>

### 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

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

#### Examples

IND-CPA<sub>Enc</sub><sup>A</sup>

$$b \leftarrow \{0, 1\}$$

$$(\mathbf{pk}, \mathbf{sk}) \leftarrow \text{KGen}(1^n)$$

$$(m_0, m_1) \leftarrow \mathcal{A}(1^n, \mathbf{pk}, c)$$

$$c \leftarrow \text{Enc}(\mathbf{pk}, m_b)$$

$$b' \leftarrow \mathcal{A}(1^n, \mathbf{pk}, c)$$

**return**  $b = b'$

which is generated as

```
1 \procedure{\$ \indcpa_ \enc ^ \adv \$}{%
2   b \sample \bin \
3   (\pk, \sk) \sample \kgen(\secpam) \
4   (m_0, m_1) \sample \adv(\secpam, \pk, c) \
5   c \sample \enc(\pk, m_b) \
6   b' \sample \adv(\secpam, \pk, c) \
7   \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

```
1 \createprocedurecommand{mypseudocode}{}{}{linenumbering}
2 \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 use customized 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

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

```

4 \pcfor i = 1..10 \pcdo \\
5 \t T[i] \sample \bin^n }

```

You can specify multiple levels via the optional first argument

```

1 \pcind[level]

```

<pre> <b>for</b> <math>i = 1..10</math> <b>do</b>   <math>T[i] \leftarrow \{0, 1\}^n</math>   <math>T[i] \leftarrow \{0, 1\}^n</math>   <math>T[i] \leftarrow \{0, 1\}^n</math>   <math>T[i] \leftarrow \{0, 1\}^n</math>   <math>T[i] \leftarrow \{0, 1\}^n</math> </pre>
--

```

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

```

You can customize the indentation shortcut by redefining

```

1 \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

```

1 \pcif , \pcfor , \pcwhile , \pcrepeat , \pcforeach

```

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

```

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

```

Additionally, on seeing

```

1 \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:

```

1 \pseudocode[space=auto]{%
2 \pcfor a \in [10] \pcdo \\
3   \pcfor a \in [10] \pcdo \\
4     \pcfor a \in [10] \pcdo \\
5       \pcif a = b \pcthen \\
6         \text{some operation} \\
7       \pcelseif a = c \pcthen \\
8         \text{some operation} \\
9       \pcelse \\
10        \text{some default operation} \\
11      \pcfi \\
12    \pcendfor \\
13  \pcendfor \\
14 \pcendfor \\
15 \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

```

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

```

```

1 \pseudocode[space=keep]{%
2 \pcfor i = 1..10 \pcdo \\
3   T[i] \sample \bin^n \\
4   T[i] \sample \bin^n \\
5   T[i] \sample \bin^n \\
6   T[i] \sample \bin^n \\
7   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 L<sup>A</sup>T<sub>E</sub>X' 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

```
1 \pseudocode[mode=text]{%
2 This is \\
3 \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
<code>pccontinue</code>	<code>\pccontinue</code>	<b>continue</b>
<code>pccomment</code>	<code>\pccomment{comment}</code>	<b>// comment</b>
<code>pcdo</code>	<code>\pcdo</code>	<b>do</b>
<code>pcdone</code>	<code>\pcdone</code>	<b>done</b>
<code>pcffalse</code>	<code>\pcffalse</code>	<b>false</b>
<code>pcif</code>	<code>\pcif</code>	<b>if</b>
<code>pcfi</code>	<code>\pcfi</code>	<b>fi</b>
<code>pcendif</code>	<code>\pcendif</code>	<b>endif</b>
<code>pcelse</code>	<code>\pcelse</code>	<b>else</b>
<code>pcelseif</code>	<code>\pcelseif</code>	<b>elseif</b>
<code>pcfor</code>	<code>\pcfor</code>	<b>for</b>
<code>pcendfor</code>	<code>\pcendfor</code>	<b>endfor</b>
<code>pcforeach</code>	<code>\pcforeach</code>	<b>foreach</b>
<code>pcendforeach</code>	<code>\pcendforeach</code>	<b>endforeach</b>
<code>pcglobvar</code>	<code>\pcglobvar</code>	<b>gbl</b>
<code>pcin</code>	<code>\pcin</code>	<b>in</b>
<code>pcnew</code>	<code>\pcnew</code>	<b>new</b>
<code>pcnull</code>	<code>\pcnull</code>	<b>null</b>
<code>pcparse</code>	<code>\pcparse</code>	<b>parse</b>
<code>pcrepeat</code>	<code>\pcrepeat{10}</code>	<b>repeat 10 times</b>
<code>pcuntil</code>	<code>\pcuntil</code>	<b>until</b>
<code>pcreturn</code>	<code>\pcreturn</code>	<b>return</b>
<code>pcthen</code>	<code>\pcthen</code>	<b>then</b>
<code>pctrue</code>	<code>\pctrue</code>	<b>true</b>
<code>pcwhile</code>	<code>\pcwhile</code>	<b>while</b>
<code>pcendwhile</code>	<code>\pcendwhile</code>	<b>endwhile</b>

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

**for *i* in{1, ..., 10}**

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

**for** *i* **in** {1, ..., 10}

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

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

```
1 \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

```
1 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](http://en.wikipedia.org/wiki/Bubble_sort))

```
Bubblesort(A : list of items)
-----
n ← length(A)
repeat
  s ← 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 ← true
until ¬s
```

can be typeset as

```
1 \procedure[syntaxhighlight=auto]{Bubblesort(A : list of items)}{
2   n \gets \mathsf{length}(A) \\\
3   repeat \\\
4     \t s \gets false \\\
5     \t for i = 1 to n-1 do \\\
6       \t\t \pcomment{if this pair is out of order} \\\
7       \t\t if A[i-1] > A[i] then \\\
8       \t\t\t \pcomment{swap them and remember something changed} \\\
9       \t\t\t \mathsf{swap}( A[i-1], A[i] ) \\\
10      \t\t\t s \gets true \\\
11      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)

---

```
 $n \leftarrow \text{length}(A)$ 
repeat
   $s \leftarrow \text{false}$ 
  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( $A[i - 1]$ ,  $A[i]$ )
       $s \leftarrow \text{true}$ 
  until  $\neg s$ 
```

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

```
1 \procedure [space=keep , syntaxhighlight=auto , addkeywords={swap , length }]{ Bubblesort(A : list of
2   items) }{
3    $n \leftarrow \text{length}(A)$  \\\
4   repeat \\\
5      $s \leftarrow \text{false}$  \\\
6     for  $i=1$  to  $n-1$  do \\\
7       \pcomment{if this pair is out of order} \\\
8       if  $A[i-1]>A[i]$  then \\\
9         \pcomment{swap them and remember something changed} \\\
10        swap( $A[i-1]$ ,  $A[i]$ ) \\\
11         $s \leftarrow \text{true}$  \\\
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)


---


 $n \leftarrow \text{length}(A)$ 
repeat
   $s \leftarrow \text{false}$ 
  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( $A[i - 1]$ ,  $A[i]$ )
       $s \leftarrow \text{true}$ 
    endif
  endfor
until  $\neg s$ 
```

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

```
1 \fbox{\procedure [space=auto , syntaxhighlight=auto , addkeywords={swap , length }]{ Bubblesort(A :
2   list of items) }{
3    $n \leftarrow \text{length}(A)$  \\\
4   repeat \\\
5      $s \leftarrow \text{false}$  \\\
6     for  $i=1$  to  $n-1$  do \\\
7       \pcomment{assuming this pair is out of order} \\\
8       if  $A[i-1]>A[i]$  then \\\
9         \pcomment{swap them and remember something changed} \\\
10        swap( $A[i-1]$ ,  $A[i]$ ) \\\
11         $s \leftarrow \text{true}$  \\\
12        endif \\\
      endfor \\\
```

```
13 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

```
1 \newcommand{\highlightkeyword}[2][\ ]{\ensuremath{\mathbf{#2}}#1}
2 \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 ← length(A)
repeat
  s ← 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 ← true
    endif
  endfor
until ¬s
```

```
1 \procedure[space=auto,syntaxhighlight=auto,addkeywords={swap,length}]{Bubblesort(A : list of
  items)}{
2 n \gets length(A) \\\
3 repeat \\\
4   s \gets false \\\
5   for i=1 to n-1 do \\\
6     \pcomment{assuming this pair is out of order} \\\
7     if A[i-1]>A[i] then \\\
8       \pcomment{swap them and remember something changed} \\\
9       swap(A[i-1], A[i]) \\\
10      s \gets true \\\
11     endif \\\
12   endfor \\\
13 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 “`linenumbers`”.

### IND-CPA<sub>Enc</sub><sup>A</sup>

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

is generated by

```
1 \procedure [linenumbering] {\$ \indcpa_ \enc ^ \adv \$} {%
2   b \sample \bin \\\
3   (\pk, \sk) \sample \kgen \secpam) \\\
4   \label {tmp:line:label} (m_0, m_1) \sample \adv (\secpam, \pk, c) \\\
5   c \sample \enc (\pk, m_b) \\\
6   b' \sample \adv (\secpam, \pk, c) \\\
7   \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`.

### IND-CPA<sub>Enc</sub><sup>A</sup>

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

is generated by

```
1 \procedure {\$ \indcpa_ \enc ^ \adv \$} {%
2   \pcln b \sample \bin \\\
3   \pcln (\pk, \sk) \sample \kgen \secpam) \\\
4   \pcln \label {tmp:line:label2} (m_0, m_1) \sample \adv (\secpam, \pk, c) \\\
5   \pcln c \sample \enc (\pk, m_b) \\\
6   \pcln b' \sample \adv (\secpam, \pk, c) \\\
7   \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`”.

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

IND-CPA <sub>Enc</sub> <sup>A</sup>	
11 :	$b \leftarrow_{\$} \{0, 1\}$
12 :	$(pk, sk) \leftarrow_{\$} \text{KGen}(1^n)$
13 :	$(m_0, m_1) \leftarrow_{\$} \mathcal{A}(1^n, pk, c)$
14 :	$c \leftarrow_{\$} \text{Enc}(pk, m_b)$
15 :	$b' \leftarrow_{\$} \mathcal{A}(1^n, pk, c)$
16 :	<b>return</b> $b = b'$

### 3.5.3 Separators

The commands `\pclseparator` 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

IND-CPA <sub>Enc</sub> <sup>A</sup>							
1 :	$b \leftarrow_{\$} \{0, 1\}$						
2 :	$(pk, sk) \leftarrow_{\$} \text{KGen}(1^n)$						
3 :	$(m_0, m_1) \leftarrow_{\$} \mathcal{A}(1^n, pk, c)$						
	<table border="1"> <tbody> <tr> <td>1 :</td> <td>Step 1</td> </tr> <tr> <td>2 :</td> <td>Step 2</td> </tr> <tr> <td>3 :</td> <td><b>return</b> <math>m_0, m_1</math></td> </tr> </tbody> </table>	1 :	Step 1	2 :	Step 2	3 :	<b>return</b> $m_0, m_1$
1 :	Step 1						
2 :	Step 2						
3 :	<b>return</b> $m_0, m_1$						
4 :	$c \leftarrow_{\$} \text{Enc}(pk, m_b)$						
5 :	$b' \leftarrow_{\$} \mathcal{A}(1^n, pk, c)$						
6 :	<b>return</b> $b = b'$						

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

```

1 \procedure [linenumbering]{\$ \indcpa_\enc^{\adv\$}}{\%
2   b \sample \bin \\\
3   (\pk, \sk) \sample \kgen(\secpam) \\\
4   (m_0, m_1) \sample \begin{subprocedure}\%
5   \dbox{\procedure{\$ \adv(\secpam, \pk, c)\$}}{\%
6     \text{Step 1} \\\
7     \text{Step 2} \\\
8     \pcreturn m_0, m_1 }}
9   \end{subprocedure} \\\
10  c \sample \enc(\pk, m_b) \\\
11  b' \sample \adv(\secpam, \pk, c) \\\
12  \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.

### IND-CPA<sub>Enc</sub><sup>A</sup>

---

```

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

```

```

1 \procedure{\$ \indcpa_\enc^{\adv}}{%
2   \pcln b \sample \bin \\\
3   \pcln (\pk, \sk) \sample \kgen(\secpam) \\\
4   \pcln (m_0, m_1) \sample \begin{subprocedure}%
5   \dbox{\procedure{\$ \adv(\secpam, \pk, c)}{%
6     \pcln \text{Step 1} \\\
7     \pcln \text{Step 2} \\\
8     \pcln \pcreturn m_0, m_1 }}
9   \end{subprocedure} \\\
10  \pcln c \sample \enc(\pk, m_b) \\\
11  \pcln b' \sample \adv(\secpam, \pk, c) \\\
12  \pcln \pcreturn b = b' }

```

## 3.7 Stacking Procedures

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

```

1 \begin{pchstack}[center] body \end{pchstack}
2 \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.

IND-CPA <sub>Enc</sub> <sup>A</sup>	Oracle $O$
1: $b \leftarrow \{0, 1\}$	1: line 1
2: $(pk, sk) \leftarrow \text{KGen}(1^n)$	2: line 2
3: $(m_0, m_1) \leftarrow \mathcal{A}^O(1^n, pk)$	
4: $c \leftarrow \text{Enc}(pk, m_b)$	
5: $b' \leftarrow \mathcal{A}(1^n, pk, c)$	
6: <b>return</b> $b = b'$	

```

1 \begin{pchstack}[center]
2 \procedure{\$ \indcpa_\enc^{\adv}}{%
3   \pcln b \sample \bin \\\
4   \pcln (\pk, \sk) \sample \kgen(\secpam) \\\
5   \pcln (m_0, m_1) \sample \adv^O(\secpam, \pk) \\\
6   \pcln c \sample \enc(\pk, m_b) \\\
7   \pcln b' \sample \adv(\secpam, \pk, c) \\\
8   \pcln \pcreturn b = b' }
9
10 \pchspace
11
12 \procedure{Oracle \$O}{%
13   \pcln \text{line 1} \\\
14   \pcln \text{line 2} \\\
15 }
16 \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.

### IND-CPA<sub>Enc</sub><sup>A</sup>

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

### Oracle $O$

1 : line 1  
 2 : line 2

```

1 \begin{pcvstack}[center]
2 \procedure{\$ \indcpa_ \enc ^ \adv \$}{%
3 \pcln b \sample \bin \ \
4 \pcln (\pk, \sk) \sample \kgen(\secpam) \ \
5 \pcln (m_0, m_1) \sample \adv^O(\secpam, \pk) \ \
6 \pcln c \sample \enc(\pk, m_b) \ \
7 \pcln b' \sample \adv(\secpam, \pk, c) \ \
8 \pcln \pcreturn b = b' }
9
10 \pcvspace
11
12 \procedure{Oracle \$O\$}{%
13 \pcln \text{line 1} \ \
14 \pcln \text{line 2}
15 }
16 \end{pcvstack}

```

Horizontal and vertical stacking can be combined

### IND-CPA<sub>Enc</sub><sup>A</sup>

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

### IND-CPA<sub>Enc</sub><sup>A</sup>

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

```

1 \begin{pchstack}[center]
2 \begin{pcvstack}
3 \procedure{\$ \indcpa_ \enc ^ \adv \$}{%
4 \pcln b \sample \bin \ \
5 \pcln (\pk, \sk) \sample \kgen(\secpam) \ \
6 \pcln (m_0, m_1) \sample \adv^{O, H_1, H_2}(\secpam, \pk) \ \
7 \pcln c \sample \enc(\pk, m_b) \ \
8 \pcln b' \sample \adv(\secpam, \pk, c) \ \
9 \pcln \pcreturn b = b' }
10
11 \pcvspace

```

```

12
13 \begin{pchstack}
14 \procedure{Oracle $O$}{%
15   \pcln \text{line 1} \\
16   \pcln \text{line 2}
17 }
18
19 \procedure{Oracle $H_1$}{%
20   \pcln \text{line 1} \\
21   \pcln \text{line 2}
22 }
23
24 \procedure{Oracle $H_2$}{%
25   \pcln \text{line 1} \\
26   \pcln \text{line 2}
27 }
28 \end{pchstack}
29 \end{pcvstack}
30
31 \pchspace
32
33 \procedure{$\text{indcpa\_enc}^{\text{adv}}$}{%
34   \pcln b \sample \bin \\
35   \pcln (\pk,\sk) \sample \kgen(\seccparam) \\
36   \pcln (m_0,m_1) \sample \adv^O(\seccparam, \pk) \\
37   \pcln c \sample \enc(\pk,m_b) \\
38   \pcln b' \sample \adv(\seccparam, \pk, c) \\
39   \pcln \pcreturn b = b' }
40
41 \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

```
1 \\\[height]
```

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

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

$\text{IND-CPA}_{\text{Enc}}^{\mathcal{A}}$ <hr style="width: 50%; margin: 0 auto;"/> 1: $b \leftarrow_{\mathcal{S}} \{0, 1\}$  <hr style="width: 50%; margin: 0 auto;"/> 2: $(\text{pk}, \text{sk}) \leftarrow_{\mathcal{S}} \text{KGen}(1^n)$ 3: $(m_0, m_1) \leftarrow_{\mathcal{S}} \mathcal{A}^O(1^n, \text{pk})$ 4: $c \leftarrow_{\mathcal{S}} \text{Enc}(\text{pk}, m_b)$ 5: $b' \leftarrow_{\mathcal{S}} \mathcal{A}(1^n, \text{pk}, c)$ 6: <b>return</b> $b = b'$
---

```

1 \procedure{$\text{indcpa\_enc}^{\text{adv}}$}{%
2   \pcln b \sample \bin \\[2\baselineskip][\hline\hline]
3   \pcln (\pk,\sk) \sample \kgen(\seccparam) \\
4   \pcln (m_0,m_1) \sample \adv^O(\seccparam, \pk) \\
5   \pcln c \sample \enc(\pk,m_b) \\
6   \pcln b' \sample \adv(\seccparam, \pk, c) \\
7   \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.

IND-CPA <sub>Enc</sub> <sup>A</sup>	IND-CPA <sub>Enc</sub> <sup>A</sup>
1 : $b \leftarrow_{\$} \{0, 1\}$	1 : $b \leftarrow_{\$} \{0, 1\}$
2 : $(pk, sk) \leftarrow_{\$} \text{KGen}(1^n)$	2 : $(pk, sk) \leftarrow_{\$} \text{KGen}(1^n)$
3 : $(state, m_0, m_1) \leftarrow_{\$} \mathcal{A}(1^n, pk, c)$	3 : $(m_0, m_1) \leftarrow_{\$} \mathcal{A}(1^n, pk, c)$
4 : $c \leftarrow_{\$} \text{Enc}(pk, m_b)$	4 : $c \leftarrow_{\$} \text{Enc}(pk, m_b)$
5 : $b' \leftarrow_{\$} \mathcal{A}(1^n, pk, c, state)$	5 : $b' \leftarrow_{\$} \mathcal{A}(1^n, pk, c, state)$
6 : <b>return</b> $b = b'$	6 : <b>return</b> $b = b'$

The image on the right is generated by:

```

1 \begin{pcimage}
2 \procedure{\$\indcpa_\enc^adv\$}{%
3   \pcln b \sample \bin \
4   \pcln (\pk,\sk) \sample \kgen (\secpam) \
5   \pcln (m_0,m_1) \sample \adv(\secpam, \pk, c) \pcnode{start} \
6   \pcln c \sample \enc(\pk,m_b) \
7   \pcln b' \sample \adv(\secpam, \pk, c, \state) \pcnode{end} \
8   \pcln \pcreturn b = b' }
9
10 \pcdraw{
11   \path[->] (start) edge[bend left=50] node[right]{\state\$} (start|-end);
12 }
13 \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

```

1 \procedure{\$\indcpa_\enc^adv\$}{%
2   \pcln b \sample \bin \
3   \pcln (\pk,\sk) \sample \kgen (\secpam) \
4   \pcln (m_0,m_1) \sample \adv(\secpam, \pk, c) \pcnode{start} \
5   \pcln c \sample \enc(\pk,m_b) \
6   \pcln b' \sample \adv(\secpam, \pk, c, \state) \pcnode{end}[draw={
7     \path[->] (start) edge[bend left=50] node[right]{\state\$} (start|-end);
8   }] \
9   \pcln \pcreturn b = b' }

```

### 3.9.1 Example: Explain your Code

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

IND-CPA<sub>Enc</sub><sup>A</sup>

---

```

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

```

**KGen**( $1^n$ ) samples a public key  $pk$  and a private key  $sk$ .

```

1  \begin{center}
2  \begin{pcimage}
3  \procedure{\$indcpa_\enc^adv\$}{%
4  \pcln  b \sample \bin  \\\
5  \pcln  (\pk,\sk) \sample \kgen (\seccparam)\pcnode{kgen}  \\\
6  \pcln  (m_0,m_1) \sample \adv(\seccparam, \pk, c)  \\\
7  \pcln  c \sample \enc(\pk,m_b)  \\\
8  \pcln  b' \sample \adv(\seccparam, \pk, c, \state)  \\\
9  \pcln  \pcreturn b = b' }
10
11 \pcdraw{
12   \node[rectangle callout,callout absolute pointer=(kgen),fill=orange]
13   at ([shift={(+3,+1)}]kgen) {
14     \begin{varwidth}{3cm}
15       \$\kgen(\seccparam)\$ samples a public key \$\pk\$ and a private key \$\sk$.
16     \end{varwidth}
17   };
18 }
19 \end{pcimage}
20 \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	Second	Third	Fourth
$b \leftarrow \{0, 1\}$			

```
1 \pseudocode{%
2 \textbf{First} \> \textbf{Second} \> \textbf{Third} \> \textbf{Fourth} \\\
3 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
$b \leftarrow \{0, 1\}$			

```
1 \pseudocode{%
2 \textbf{First} \< \textbf{Second} \< \textbf{Third} \< \textbf{Fourth} \\\
3 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
$b \leftarrow \{0, 1\}$			

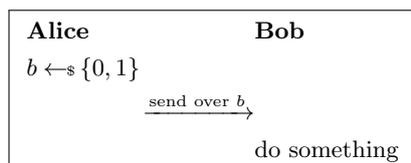
```
1 \pseudocode{%
2 \pseudocode[colsep=1em, addtolength=10em]{%
3 \textbf{First} \< \textbf{Second} \< \textbf{Third} \< \textbf{Fourth} \\\
4 b \sample \bin \< b \sample \bin \< b \sample \bin \< b \sample \bin}
```

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

```
1 \pseudocode{%
2 \textbf{Alice} \> \> \textbf{Bob} \ \ \
3 b \sample \bin \> \> \ \ \
4 \> \xrightarrow{\text{send over } b} \> \ \ \
5 \> \> \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

```
1 \renewcommand{\pctabname}{\myTab}
2
3 \pseudocode{%
4 \textbf{Alice} \myTab \myTab \textbf{Bob} \ \ \
5 b \sample \bin \myTab \myTab \ \ \
6 \myTab \xrightarrow{\text{send over } b} \myTab \ \ \
7 \myTab \myTab \text{do something}}
```

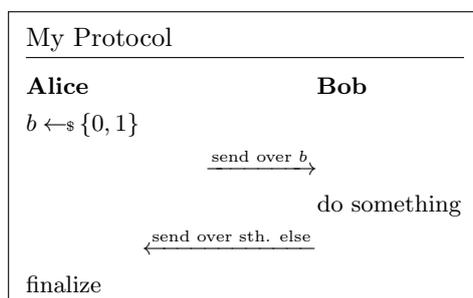
### 4.1.2 Custom Line Spacing and Horizontal Rules

As explained underlying the pseudocode command is an flalign environment. This would allow the use of \\[spacing] to specify 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

```

1 \procedure{My Protocol}{%
2   \textbf{Alice} \> \> \textbf{Bob}  \> \>
3   b \sample \bin \> \> \> \>
4   \> \xrightarrow{\text{send over } b} \> \>
5   \> \> \text{do something} \> \>
6   \> \xleftarrow{\text{send over sth. else}} \> \>
7   \text{finalize} \> \>
8 }

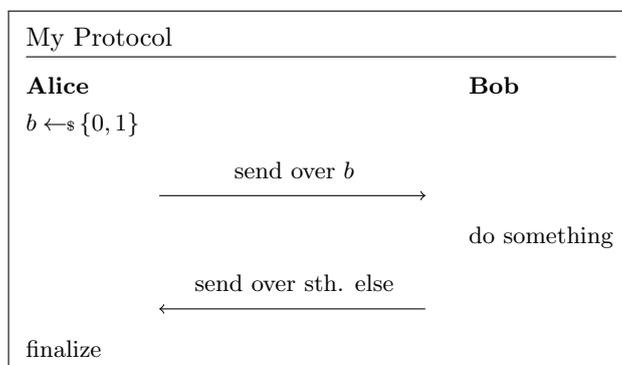
```

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.

```

1 \sendmessageright*[3.5cm]{message}
2 \sendmessageleft*[3.5cm]{message}

```



```

1 \procedure{My Protocol}{%
2 \textbf{Alice} \> \> \textbf{Bob} \& \&
3 b \sample \bin \> \> \& \&
4 \> \sendmessageright*{\text{send over } b} \> \& \&
5 \> \> \text{do something} \& \&
6 \> \sendmessageleft*{\text{send over sth. else}} \> \& \&
7 \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.

My Protocol

---

**Alice**

$b \leftarrow \{0, 1\}$

send over  $b$

----->  
Text below

**Bob**

do something

← send over sth. else

finalize

```

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

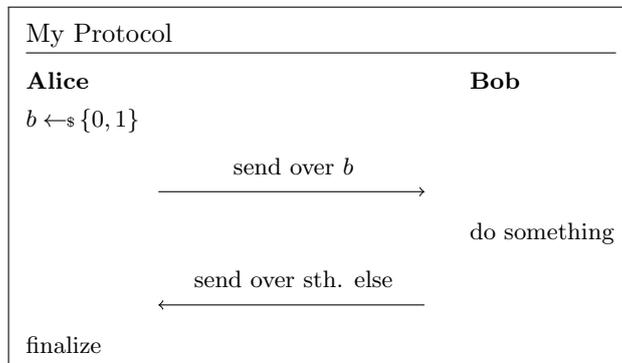
**width** The width of the column

**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 Tabbng

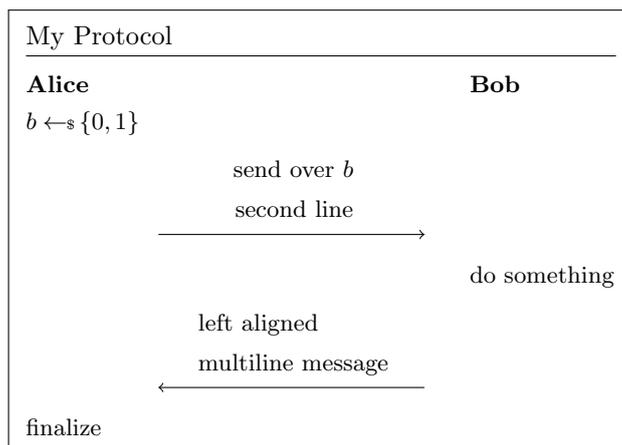
When typesetting protocols you might find that using two tabs instead of a single tab usually provides a better result as this ensures that all columns are left aligned. For this you can use `\<` instead of `\>` (see Chapter 4).

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.



```

1 \procedure{My Protocol}{%
2   \textbf{Alice} \< \< \textbf{Bob}   \> \>
3   b \sample \bin \< \< \> \>
4   \< \sendmessageright*{\text{send over } b\ \&\text{second line}} \< \>
5   \< \< \text{do something} \> \>
6   \< \sendmessageleft*{\&\text{left aligned}} \&\text{multiline message} \< \>
7   \text{finalize} \< \<

```

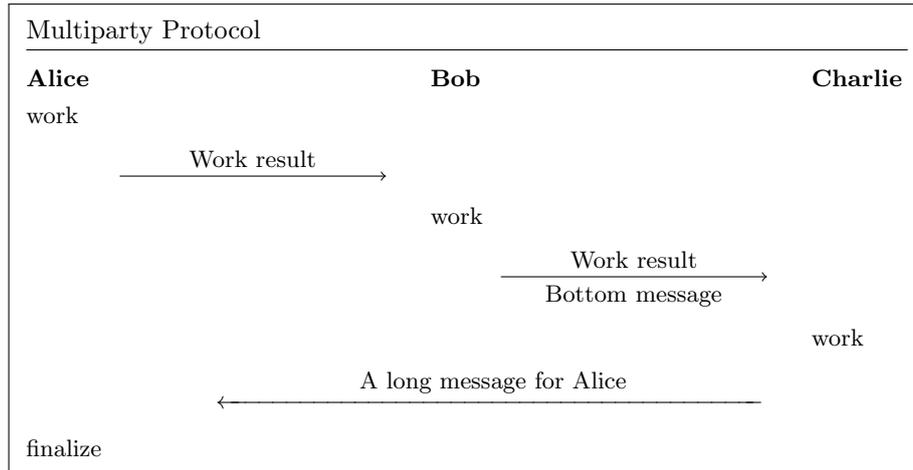
### 5.2.1 Multiplayer Protocols

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

```

1 \sendmessagerightx [width]{columnspan}{Text}
2 \sendmessageleftx [width]{columnspan}{Text}

```



```

1 \begin{center}
2 \procedure{Multiparty Protocol}{%
3 \textbf{Alice} \< \< \textbf{Bob} \< \< \textbf{Charlie} \< \<
4 \text{work} \< \< \< \< \< \<
5 \< \sendmessageright{top=Work result} \< \< \< \< \< \<
6 \< \< \text{work} \< \< \< \< \< \<
7 \< \< \< \sendmessageright{top=Work result ,bottom=Bottom message} \< \< \< \<
8 \< \< \< \< \text{work} \< \< \< \< \< \<
9 \< \sendmessageleftx[7cm]{8}{\text{A long message for Alice}} \< \< \< \< \< \<
10 \text{finalize} \< \< \< \< \< \<

```

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

### 5.2.2 Divisions

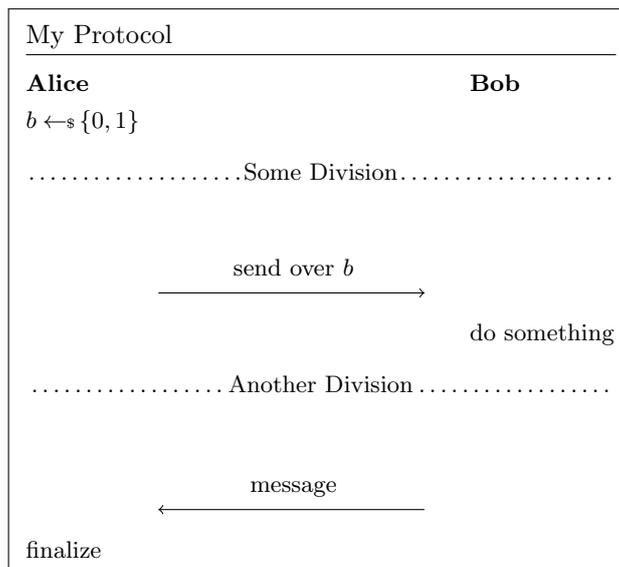
You can use `\pintertext` in order to divide protocols (or other pseudocode for that matter).

```

1 \pintertext[dotted|center]{Division Text}

```

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



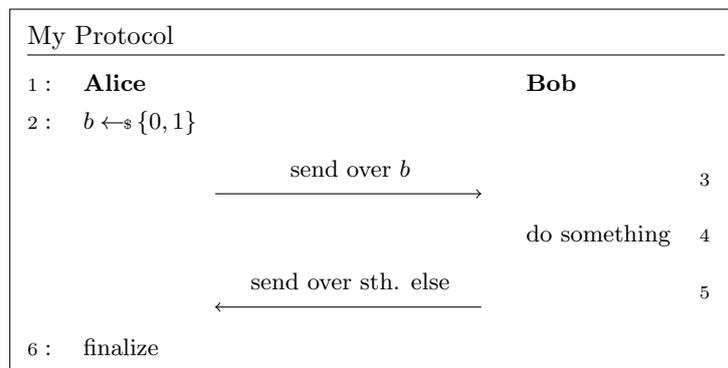
```

1 \procedure{My Protocol}{%
2 \textbf{Alice} \< \< \textbf{Bob} \< \<
3 b \sample \bin \< \< \pclb
4 \pcintertext[dotted]{Some Division} \< \<
5 \< \sendmessengeright*{\text{send over } b} \< \< \< \<
6 \< \< \text{do something} \< \< \< \<
7 \pcintertext[dotted]{Another Division} \< \<
8 \< \sendmessageleft*{\text{message}} \< \< \< \<
9 \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.



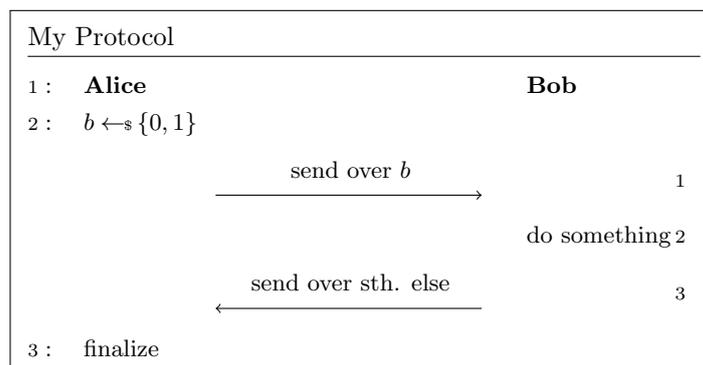
Which is generated as

```

1 \procedure{My Protocol}{%
2 \pcln \textbf{Alice} \< \< \textbf{Bob} \< \<
3 \pcln b \sample \bin \< \< \< \<
4 \< \sendmessengeright*{\text{send over } b} \< \< \< \< \pclnr \< \<
5 \< \< \text{do something} \< \< \< \< \pclnr \< \<
6 \< \sendmessageleft*{\text{send over sth. else}} \< \< \< \< \pclnr \< \<
7 \pcln \text{finalize} \< \< \< \<

```

And using `\pcrln`:



Which is generated as

```

1 \procedure{My Protocol}{%
2 \pcln \textbf{Alice} \< \< \textbf{Bob} \< \<
3 \pcln b \sample \bin \< \< \< \<
4 \< \sendmessengeright*{\text{send over } b} \< \< \< \< \pcrln \< \<
5 \< \< \text{do something} \< \< \< \< \pcrln \< \<
6 \< \sendmessageleft*{\text{send over sth. else}} \< \< \< \< \pcrln \< \<

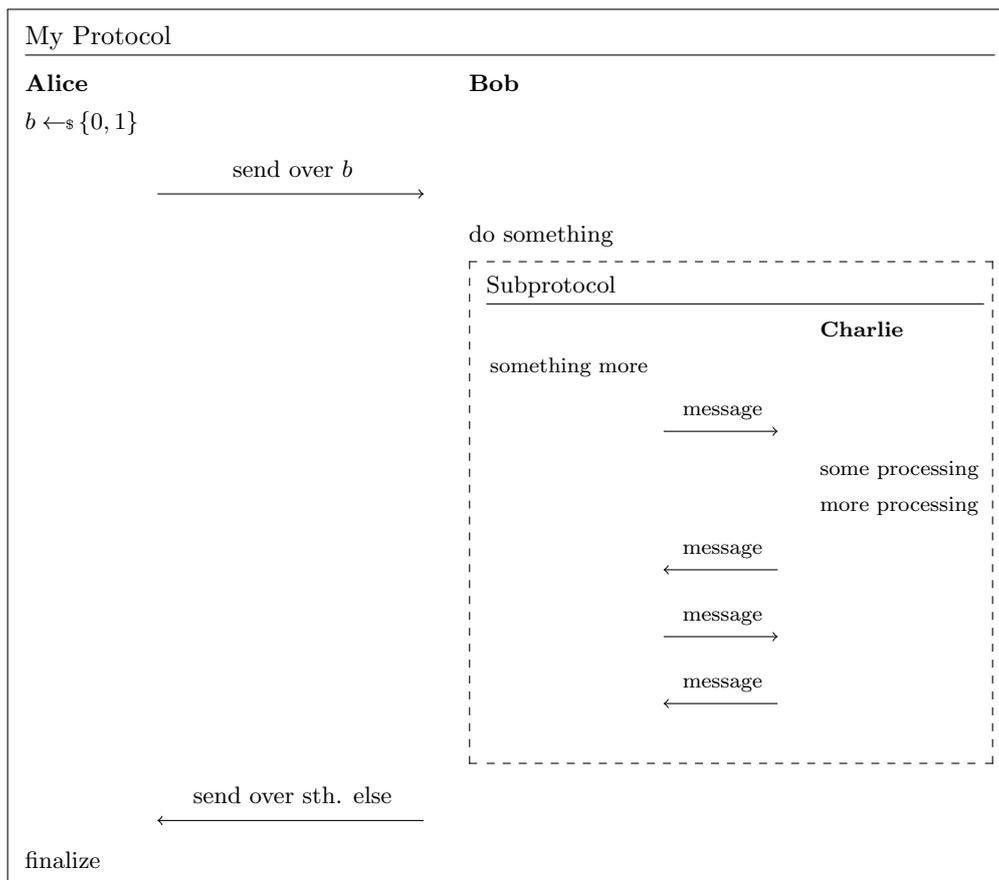
```

### 5.3.1 Separators

The commands `\pclnseparator` and `\pclnseparator` 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.



```

1 \procedure{My Protocol}{%
2   \textbf{Alice} \< \< \textbf{Bob}  \> \>
3   b \sample \bin \< \< \> \>
4   \< \< \sendmessengeright*{\text{send over } b} \< \< \> \>
5   \< \< \text{do something} \> \>
6   \< \< \dbox{\begin{subprocedure}\procedure{Subprotocol}{
7     \< \< \textbf{Charlie} \> \>
8     \text{something more} \< \< \> \>
9     \< \< \sendmessengeright*[1.5cm]{\text{message}} \< \< \> \>
10    \< \< \text{some processing} \> \>
11    \< \< \text{more processing} \> \>
12    \< \< \sendmessageleft*[1.5cm]{\text{message}} \< \< \> \>
13    \< \< \sendmessengeright*[1.5cm]{\text{message}} \< \< \> \>
14    \< \< \sendmessageleft*[1.5cm]{\text{message}} \< \< \> \>
15  }\end{subprocedure}} \> \>
16  \< \< \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.

```
1 \begin{gameproof}
2 proof goes here
3 \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 `Gamecounter` where counter is a consecutive counter. Thus, we can create the following setup

	<u>Game<sub>1</sub>(n)</u>	<u>Game<sub>2</sub>(n)</u>
1 :	Step 1	Step 1
2 :	Step 2	Step 2

by using

```
1 \begin{gameproof}
2 \gameprocedure[linenumbering,mode=text]{%
3 Step 1 \\
4 Step 2
5 }
6 \gameprocedure[mode=text]{%
7 Step 1 \\
8 Step 2
9 }
10 \end{gameproof}
```

#### 6.1.1 Highlight Changes

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

	<u>Game<sub>1</sub>(n)</u>	<u>Game<sub>2</sub>(n)</u>
1 :	Step 1	Step 1
2 :	Step 2	<u>Step 2</u>

```
1 \begin{gameproof}
2 \gameprocedure[linenumbering,mode=text]{%
3 Step 1 \\
4 Step 2
5 }
```

```

6 \gameprocedure [mode=text] {%
7   Step 1 \\
8   \gamechange{Step 2}
9 }
10 \end{gameproof}

```

### 6.1.2 Boxed games

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

$\text{Game}_1(n)$	$\text{Game}_2(n)$	$\text{Game}_3(n)$	$\text{Game}_4(n)$
1 : Step 1	Step 1; <span style="border: 1px solid black; padding: 2px;">Alternative step 1</span>		Step 1
2 : Step 2	Step 2 is different		Step 2

```

1 \begin{gameproof}
2 \gameprocedure{%
3   \pcln \text{Step 1} \\
4   \pcln \text{Step 2}
5 }
6 \tbxgameprocedure{%
7   \text{Step 1}; \pobox{\text{Alternative step 1}} \\
8   \gamechange{\text{Step 2 is different}}
9 }
10 \gameprocedure{%
11  \pcln \text{Step 1} \\
12  \pcln \text{\gamechange{Step 2}}
13 }
14 \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.

```

1 \addgamehop{startgame}{endgame}{options}

```

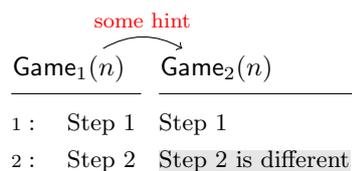
Here options allows you to specify the hint as well as the style. The following options are available

**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”.



```

1 \begin{gameproof}
2 \gameprocedure{%
3 \pcln \text{Step 1}  \\\
4 \pcln \text{Step 2}
5 }
6 \gameprocedure{%
7 \text{Step 1}  \\\
8 \gamechange{\text{Step 2 is different}}
9 }
10 \addgamehop{1}{2}{hint=\footnotesize some hint , nodestyle=red}
11 \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

```

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

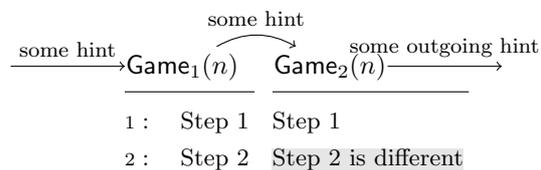
```

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

```

1 \addstartgamehop[first game]{options}
2 \addendgamehop[last game]{options}

```



```

1 \begin{gameproof}
2 \gameprocedure{%
3 \pcln \text{Step 1}  \\\
4 \pcln \text{Step 2}
5 }
6 \gameprocedure{%
7 \text{Step 1}  \\\
8 \gamechange{\text{Step 2 is different}}
9 }
10 \addstartgamehop{hint=\footnotesize some hint , edgestyle=}
11 \addgamehop{1}{2}{hint=\footnotesize some hint}
12 \addendgamehop{hint=\footnotesize some outgoing hint , edgestyle=}
13 \end{gameproof}

```

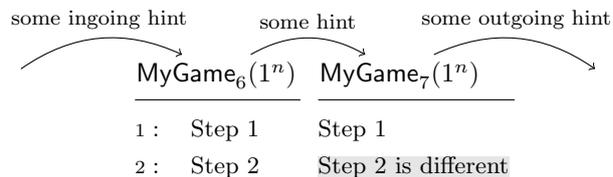
### 6.1.4 Numbering and Names

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.

```

1 \begin{gameproof}[options]

```



```

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

```



# Chapter 7

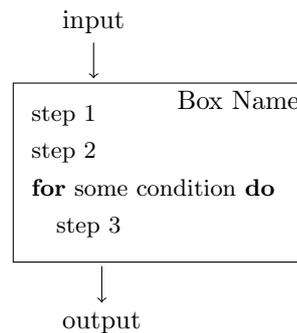
## Black-box Reductions

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

```
1 \begin{bbrenv}{A}
2 \begin{bbrbox}[name=Box Name]
3 % The Box's content
4 \end{bbrbox}
5 % Commands to display communication, input output etc
6 \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:



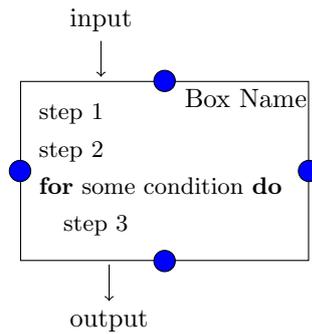
This box is generated as

```
1 \begin{bbrenv}{A}
2 \begin{bbrbox}[name=Box Name]
3 \pseudocode{
4 \text{step 1} \\
5 \text{step 2} \\
6 \pcfor \text{some condition} \pcdo \\
7 \pcind \text{step 3}
8 }
9 \end{bbrbox}
10 \bbrinput{input}
11 \bboutput{output}
12 \end{bbrenv}
```

The commands `bbrinput` and `bboutput` 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 (<https://www.ctan.org/pkg/pgf>).

```
1 \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.

```

1 \begin{bbrenv}{A}
2   \begin{bbrbox}[name=Box Name]
3     \pseudocode{
4       \text{step 1} \\
5       \text{step 2} \\
6       \pcfor \text{some condition} \pcdo \\
7       \pcind \text{step 3}
8     }
9   \end{bbrbox}
10  \bbrinput{input}
11  \bbroutput{output}
12
13  \filldraw[fill=blue] (A.north) circle (4pt);
14  \filldraw[fill=blue] (A.west) circle (4pt);
15  \filldraw[fill=blue] (A.east) circle (4pt);
16  \filldraw[fill=blue] (A.south) circle (4pt);
17 \end{bbrenv}

```

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

```

1 name=Box Name

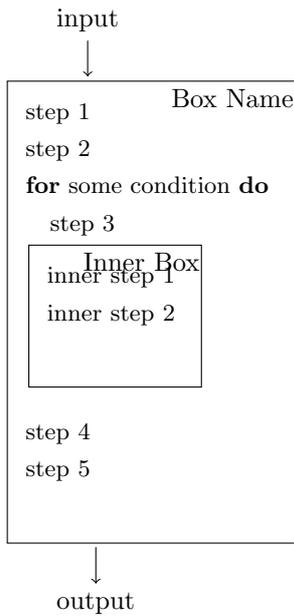
```

to specify the label. The following options are available

Option	Description
name	Specifies the box' label
namepos	Specifies the position (left, center, right, top left, top center, top right)
namestyle	Specifies the style of the name
minheight	The minimal height
xshift	Allows horizontal positioning
yshift	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.



```

1 \begin{bbrenv}{A}
2   \begin{bbrbox}[name=Box Name]
3     \pseudocode{
4       \text{step 1} \\
5       \text{step 2} \\
6       \pcfor \text{some condition} \pcdo \\
7       \pcind\text{step 3}
8     }
9
10    \begin{bbrenv}{B}
11      \begin{bbrbox}[name=Inner Box]
12        \pseudocode{
13          \text{inner step 1} \\
14          \text{inner step 2}
15        }
16      \end{bbrbox}
17    \end{bbrenv}
18
19    \pseudocode{
20      \text{step 4} \\
21      \text{step 5}
22    }
23  \end{bbrbox}
24  \bbrinput{input}
25  \bbroutput{output}
26 \end{bbrenv}

```

## 7.2 Messages and Queries

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

```

1 \bbrmsgto{options}
2 \bbrmsgfrom{options}
3 \bbrqryto{options}
4 \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

```

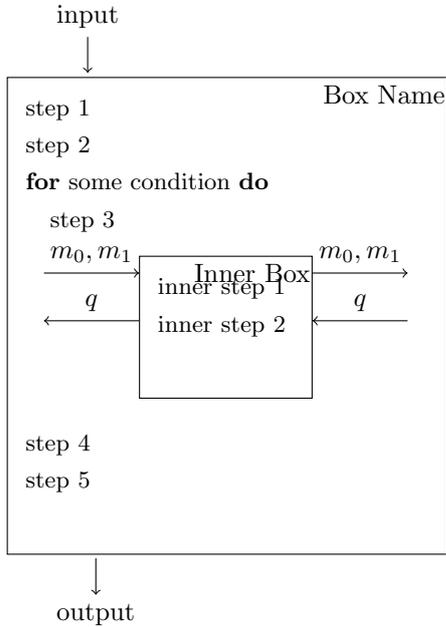
1 \bbrmsgto{top=Top Label, side=Side Label}

```

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

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

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



```

1 \begin{bbrenv}{A}
2   \begin{bbrbox}[name=Box Name]
3     \pseudocode{
4       \text{step 1} \\
5       \text{step 2} \\
6       \pcfor \text{some condition} \pcdo \\
7         \pcind\text{step 3}
8     }
9
10    \begin{bbrenv}{B}
11      \begin{bbrbox}[name=Inner Box]
12        \pseudocode{
13          \text{inner step 1} \\
14          \text{inner step 2}
15        }
16      \end{bbrbox}
17
18      \bbrmsgto{top={\$m_0 ,m_1\$}}
19      \bbrmsgfrom{top={\$q\$}}
20
21
22      \bbrqryto{top={\$m_0 ,m_1\$}}
23      \bbrqryfrom{top={\$q\$}}
24
25    \end{bbrenv}
26
27    \pseudocode{
28      \text{step 4} \\
29      \text{step 5}
30    }
31  \end{bbrbox}
32  \bbrinput{input}
33  \bbroutput{output}
34 \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 underneath the reduction commands is a TIKZ image (<https://www.ctan.org/pkg/pgf/>). For each label position (top, side, bottom) a node is generated. You can provide additional properties for this node using the options:

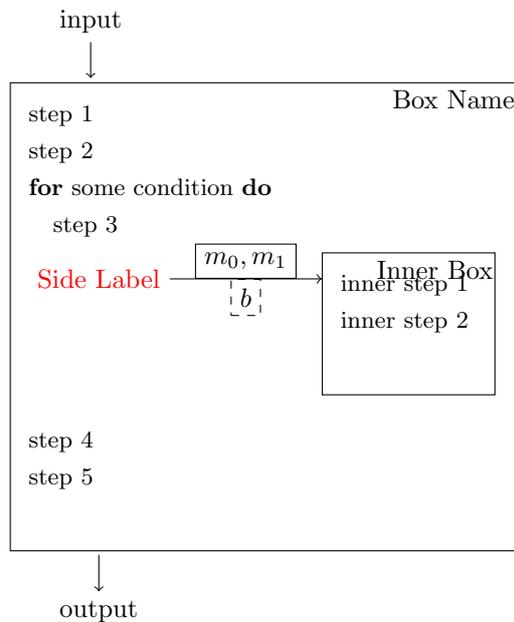
- topstyle
- sidestyle
- bottomstyle

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 specify the length of the arrow.



```

1 \begin{bbrenv}{A}
2   \begin{bbrbox}[name=Box Name]
3     \pseudocode{
4       \text{step 1} \\
5       \text{step 2} \\
6       \pcfor \text{some condition} \pcdo \\
7       \pcind \text{step 3}
8     }
9
10    \begin{bbrenv}{B}
11      \begin{bbrbox}[name=Inner Box]
12        \pseudocode{
13          \text{inner step 1} \\
14          \text{inner step 2}
15        }
16      \end{bbrbox}
17
18      \bbrmsgto{top={m_0, m_1}, side=Side Label, bottom=b$, length=2cm,
19               topstyle={draw, solid}, sidestyle={red}, bottomstyle={draw, dashed}}
20    
```

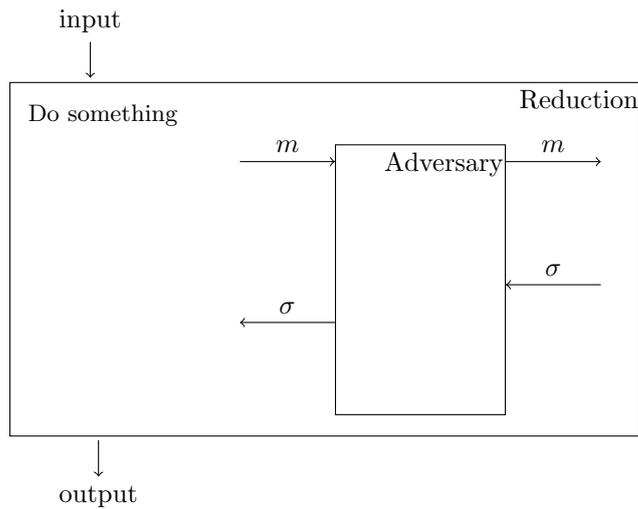
```

21 \end{bbrenv}
22
23 \pseudocode{
24   \text{step 4} \\
25   \text{step 5} \\
26 }
27 \end{bbrbox}
28 \bbrinput{input}
29 \bbroutput{output}
30 \end{bbrenv}

```

## 7.2.2 Add Space

If the spacing between messages is not sufficient you can use the `bbrmsgspace` and `bbrqryspace` commands to



add additional space.

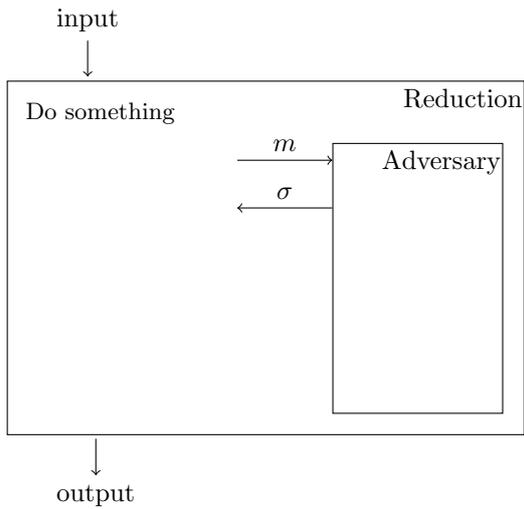
```

1 \begin{bbrenv}{A}
2   \begin{bbrbox}[name=Reduction]
3     \pseudocode{
4       \text{Do something}
5     }
6
7   \begin{bbrenv}{B}
8
9     \begin{bbrbox}[name=Adversary , minheight=3cm, xshift=4cm]
10
11     \end{bbrbox}
12
13     \bbrmsgto{top=$m$}
14     \bbrmsgspace{1.5cm}
15     \bbrmsgfrom{top=$\sigma$}
16
17     \bbrqryto{top=$m$}
18     \bbrqryspace{1cm}
19     \bbrqryfrom{top=$\sigma$}
20
21   \end{bbrenv}
22
23 \end{bbrbox}
24 \bbrinput{input}
25 \bbroutput{output}
26 \end{bbrenv}

```

## 7.2.3 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



```

1 \begin{bbrenv}{A}
2   \begin{bbrbox}[name=Reduction]
3     \pseudocode{
4       \text{Do something}
5     }
6
7   \begin{bbrenv}{B}
8
9     \begin{bbrbox}[name=Adversary, minheight=3cm, xshift=4cm]
10
11       \end{bbrbox}
12
13       \bbrmsgto{top=$m$}
14       \bbrmsgfrom{top=$\sigma$}
15
16     \end{bbrenv}
17
18   \end{bbrbox}
19   \bbrinput{input}
20   \bbroutput{output}
21 \end{bbrenv}

```

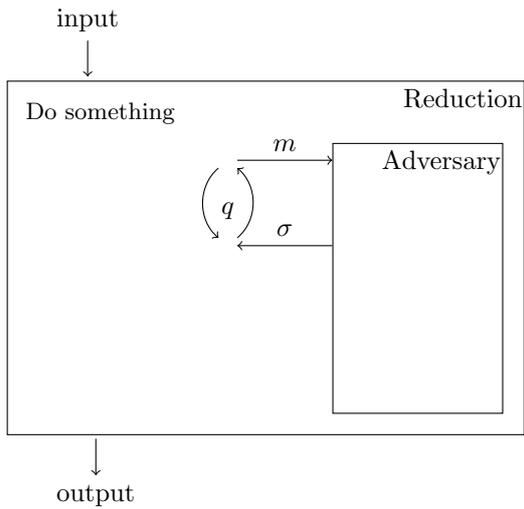
First note that by specifying the `minheight` and `xshift` option we shifted the adversary box a bit to the right and enlarged its box. Further we specified custom names for the node on the side of the two messages. We can now use the `bbrloop` command to visualize that these two messages are exchanged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

```

1 \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.



```

1 \begin{bbrenv}{A}
2   \begin{bbrbox}[name=Reduction]
3     \pseudocode{
4       \text{Do something}
5     }
6
7   \begin{bbrenv}{B}
8
9     \begin{bbrbox}[name=Adversary, minheight=3cm, xshift=4cm]
10
11     \end{bbrbox}
12
13     \bbrmsgto{top=$n$, sidename=BeginLoop}
14     \bbrmsgspace{0.5cm}
15     \bbrmsgfrom{top=$\sigma$, sidename=EndLoop}
16     \bbrloop{BeginLoop}{EndLoop}{center=$q$}
17
18   \end{bbrenv}
19
20 \end{bbrbox}
21 \bbrinput{input}
22 \bbroutput{output}
23 \end{bbrenv}

```

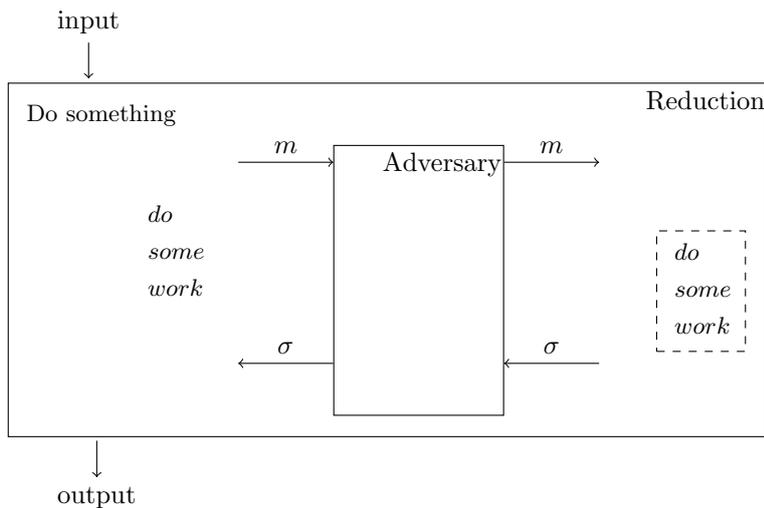
## 7.2.4 Intertext

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

```

1 \bbrmsgtxt[options]{Text}
2 \bbrqrytxt[options]{Text}

```



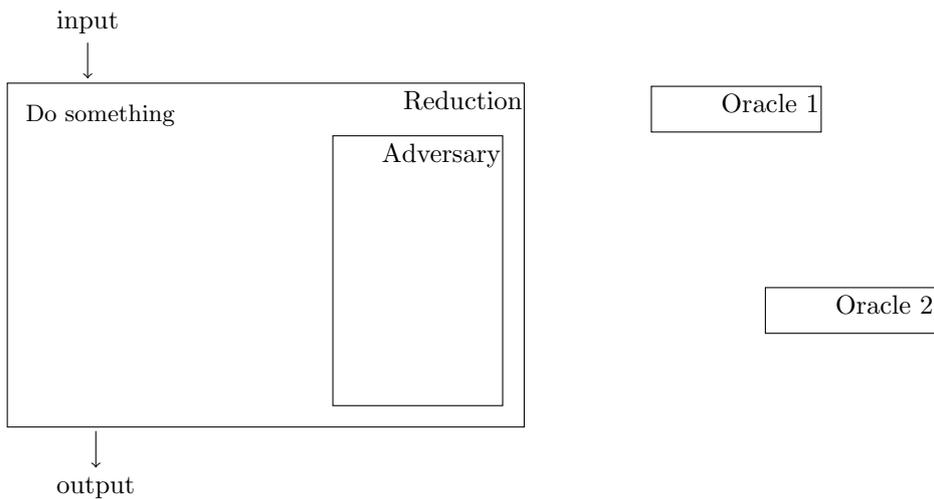
```

1 \begin{bbrenv}{A}
2   \begin{bbrbox}[name=Reduction]
3     \pseudocode{
4       \text{Do something}
5     }
6
7   \begin{bbrenv}{B}
8
9     \begin{bbrbox}[name=Adversary , minheight=3cm, xshift=4cm]
10
11       \end{bbrbox}
12
13       \bbrmsgto{top=$m$}
14       \bbrmsgtxt{\pseudocode{
15         do \\
16         some \\
17         work
18       }}
19       \bbrmsgfrom{top=$\sigma$}
20
21       \bbrqryto{top=$m$}
22       \bbrqrytxt[before skip=0.5cm, nodestyle={draw, dashed}, xshift=2cm]{\pseudocode{
23         do \\
24         some \\
25         work
26       }}
27       \bbrqryfrom{top=$\sigma$}
28
29     \end{bbrenv}
30
31   \end{bbrbox}
32   \bbrinput{input}
33   \bbroutput{output}
34 \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* environment.



```

1 \begin{bbrenv}{A}
2   \begin{bbrbox}[name=Reduction]
3     \pseudocode{
4       \text{Do something}
5     }
6
7   \begin{bbrenv}{B}
8     \begin{bbrbox}[name=Adversary, minheight=3cm, xshift=4cm]
9       \end{bbrbox}
10
11   \end{bbrenv}
12
13 \end{bbrbox}
14 \bbrinput{input}
15 \bbroutput{output}
16
17 \begin{bbroracle}{OraA}
18   \begin{bbrbox}[name=Oracle 1]
19     \end{bbrbox}
20 \end{bbroracle}
21
22 \begin{bbroracle}{OraB}[vdistance=2cm, hdistance=3cm]
23   \begin{bbrbox}[name=Oracle 2]
24     \end{bbrbox}
25 \end{bbroracle}
26 \end{bbrenv}

```

Via the option “`hdistance=length`” and “`vdistance=length`” you can control the horizontal and vertical position of the oracle. By default this value is set to 1.5cm and `\baselineskip`.

### 7.3.1 Communicating with Oracles

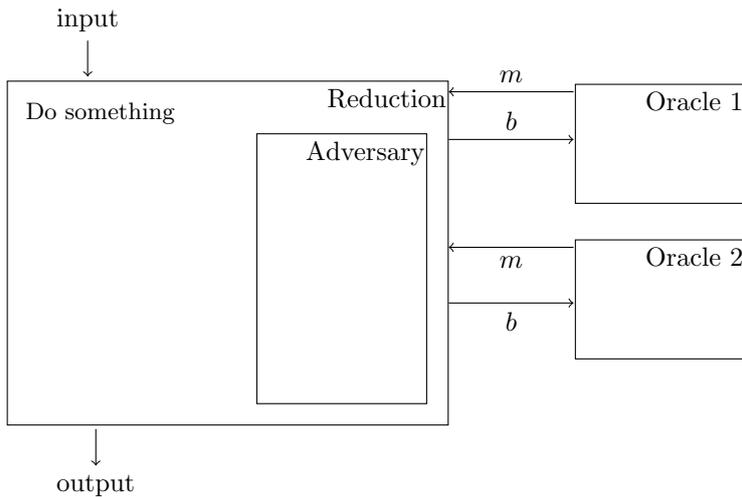
As oracles use the `bbrbox` environment we can directly use the established ways to send messages and queries to oracles. In addition you can use the `\bbroracleqryfrom` and `\bbroracleqryto`.

```

1 \bbroracleqryfrom{options}
2 \bbroracleqryto{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.



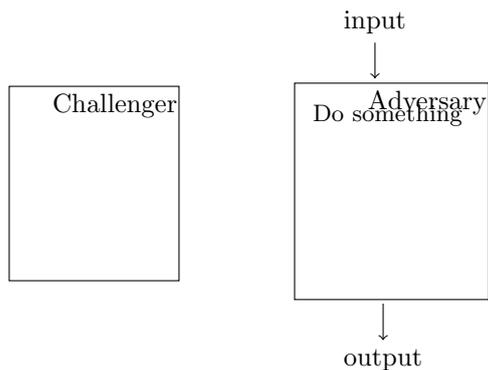
```

1 \begin{bbrenv}{A}
2   \begin{bbrbox}[name=Reduction]
3     \pseudocode{
4       \text{Do something}
5     }
6
7   \begin{bbrenv}{B}
8     \begin{bbrbox}[name=Adversary, minheight=3cm, xshift=3cm]
9       \end{bbrbox}
10
11    \end{bbrenv}
12
13  \end{bbrbox}
14  \bbrinput{input}
15  \bbroutput{output}
16
17  \begin{bbroracle}{OraA}
18    \begin{bbrbox}[name=Oracle 1, minheight=1cm]
19      \end{bbrbox}
20    \end{bbroracle}
21    \bbroracleqryfrom{top=$n$}
22    \bbroracleqryto{top=$b$}
23
24  \begin{bbroracle}{OraB}
25    \begin{bbrbox}[name=Oracle 2, minheight=1cm]
26      \end{bbrbox}
27    \end{bbroracle}
28    \bbroracleqryfrom{bottom=$n$}
29    \bbroracleqryto{bottom=$b$}
30 \end{bbrenv}

```

## 7.4 Challengers

Each box can have one or more challengers which are drawn on the left hand side of the box. Challengers behave identically to oracles with the exception that they are to the left of the box. A challenger is created similarly to a *bbrenv* environment using the *bbrchallenger* environment. Challengers go behind the single *bbrbox* environment within an *bbrenv* environment.



```

1 \begin{bbrenv}{A}
2   \begin{bbrbox}[name=Adversary , minheight=2cm]
3     \pseudocode{
4       \text{Do something}
5     }
6
7   \end{bbrbox}
8   \bbrinput{input}
9   \bbroutput{output}
10
11  \begin{bbrchallenger}{ChaA}
12    \begin{bbrbox}[name=Challenger , minheight=2cm]
13
14    \end{bbrbox}
15  \end{bbrchallenger}
16 \end{bbrenv}

```

Via the option “`hdistance=length`” and “`vdistance=length`” you can control the horizontal and vertical position of the challenger. By default this value is set to 1.5cm and `\baselineskip`.

### 7.4.1 Communicating with Challengers

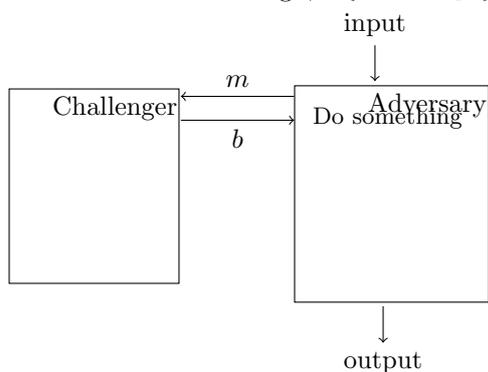
As challengers use the `bbrbox` environment we can directly use the established ways to send messages and queries to oracles. In addition you can use the `\bbrchallengerqryfrom` and `\bbrchallengerqryto`.

```

1 \bbrchallengerqryfrom{options}
2 \bbrchallengerqryto{options}

```

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



```

1 \begin{bbrenv}{A}
2   \begin{bbrbox}[name=Adversary , minheight=2cm]
3     \pseudocode{
4       \text{Do something}
5     }

```

```
6
7 \end{bbrbox}
8 \bbrinput{input}
9 \bbroutput{output}
10
11 \begin{bbrchallenger}{ChaA}
12   \begin{bbrbox}[name=Challenger , minheight=2cm]
13
14     \end{bbrbox}
15   \end{bbrchallenger}
16
17   \bbrchallengerqryfrom{top=$m$}
18   \bbrchallengerqryto{bottom=$b$}
19 \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 -
--

```
1 \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 -
--

```
1 \newsavebox { \mypcbox }
2 \begin { lrbox } { \mypcbox } %
3 \pseudocode [space=keep , mode=text ] {                      Pseudocode                      with                      - spaces - } %
4 \end { lrbox }
5 \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”.

### 8.3 Hyperref

The hyperref package (<https://www.ctan.org/pkg/hyperref>) should be loaded before cryptocode. If this is not possible call the `\pcfixhyperref` after `\begin{document}`.