IMPORTANT NOTICE
This documentation is furnished for informational use only and is subject to change without notice. Galois, Inc. assumes no responsibility or liability for any errors or inaccuracies that may appear in this documentation. [Note: This paragraph is NOT included for purposes of copyright protection. It is a liability reduction paragraph, and falls outside the scope of this document. This paragraph should be reviewed by qualified legal counsel to determine its appropriateness.]

Copyright 2003-2008 Galois, Inc. All rights reserved by Galois, Inc.
This documentation, or any part of it, may not be reproduced, displayed, photocopied, transmitted, or otherwise copied in any form or by any means now known or later developed, such as electronic, optical, or mechanical means, without express written authorization from Galois, Inc.
Warning: This computer program and its documentation are protected by copyright law and international treaties. Any unauthorized copying or distribution of this program, its documentation, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted under the maximum extent possible under the law.
The software installed in accordance with this documentation is copyrighted and licensed by Galois, Inc. under separate license agreement. This software may only be used pursuant to the terms and conditions of such license agreement. Any other use may be a violation of law.

TRADEMARKS
Cryptol is a registered trademark of Galois, Inc. in the United States and other countries.
UNIX is a registered trademark of The Open Group in the U. S. and other countries.
Linux is a registered trademark of Linus Torvalds.
Solaris, Java, Java Runtime Edition, JRE, and other Java-related marks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.
XILINX, the Xilinx logo, and other designated brands included herein are trademarks of Xilinx, Inc.
Other company or product names mentioned herein may be trademarks or registered trademarks of their respective owners.
Trademark specifications are subject to change without notice. All terms mentioned in this documentation that are known to be trademarks or service marks have been appropriately capitalized to the best of our knowledge; however, Galois cannot attest to the accuracy of all trademark information. Use of a term in this documentation should not be regarded as affecting the validity of any trademark or service mark.

Galois, Inc
421 SW Sixth Avenue, Suite 300
Portland, OR 97204
# Table of Contents

**SECTION 1**

| The Cryptol Toolset: An Overview | 1 |

**SECTION 2**

<table>
<thead>
<tr>
<th>The Cryptol Interpreter</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Expressions</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Interacting with the Interpreter</td>
<td>6</td>
</tr>
<tr>
<td>2.3 Programs</td>
<td>8</td>
</tr>
<tr>
<td>2.4 Starting Cryptol</td>
<td>9</td>
</tr>
<tr>
<td>2.5 Cryptol interpreter options and flags</td>
<td>11</td>
</tr>
<tr>
<td>2.6 Cryptol Modes</td>
<td>17</td>
</tr>
<tr>
<td>2.7 Cryptol interpreter commands</td>
<td>19</td>
</tr>
<tr>
<td>2.8 Using environment variables</td>
<td>25</td>
</tr>
</tbody>
</table>

**SECTION 3**

<table>
<thead>
<tr>
<th>Verifying Foreign VHDL Implementations</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
<td>28</td>
</tr>
<tr>
<td>3.2 Requirements</td>
<td>29</td>
</tr>
<tr>
<td>3.3 Getting the Example VHDL</td>
<td>30</td>
</tr>
<tr>
<td>3.4 Step #1: Examining the VHDL, Creating an Extern</td>
<td>31</td>
</tr>
<tr>
<td>3.5 Discovering the Interface and Delay</td>
<td>33</td>
</tr>
<tr>
<td>3.6 Generating Functions For Equivalence Checking</td>
<td>37</td>
</tr>
<tr>
<td>3.7 Equivalence Checking</td>
<td>39</td>
</tr>
<tr>
<td>3.8 Dealing With Overallocated Pins (a.k.a. IOBs)</td>
<td>41</td>
</tr>
</tbody>
</table>
The Cryptol Toolset: An Overview
This chapter provides a short tutorial for getting started with Cryptol, aimed at beginners. When you start Cryptol, you should see a banner that looks like this:

```plaintext
% cryptol
Cryptol v1.4, Copyright (c) 2001-2004 Galois Connections Inc.
www.cryptol.net

Type :? for help
Cryptol>
```

Exactly how you start Cryptol depends on the platform you are on. On Unix-like systems, just typing `cryptol` at a command line should get you started, assuming your system administrator has properly installed it. On Windows, you generally click on the Cryptol icon. Other than how you start Cryptol, it should behave the same way on all platforms.

To quit the interpreter, either press Ctrl-D (that is, hold down the Ctrl key and press ‘D’) on Unix machines, or Ctrl-Z on Windows machines, or, better yet, use the command `:quit` as in the following example:

```plaintext
Cryptol> :quit
[Leaving Cryptol]
% ...back to operating system (or window closes on Windows)
```

As we will see later, commands to the interpreter can always be abbreviated. For instance, `:quit` can be abbreviated all the way to `:q`. 
2.1 Expressions

Basically, using Cryptol is just like using a calculator: you type in an expression, and Cryptol evaluates it for you and prints the result.

```
cryptol> 0xa + 0b11
0xd
Cryptol> 7 - 0o2
0x5
```

As seen in the above example, Cryptol accepts literal numbers in 4 bases:
- binary (prefixed with 0b),
- octal (prefixed with 0o),
- decimal (with no prefix),
- hexadecimal (prefixed with 0x)

Cryptol, however, can print the results in any base from 2 to 36. See Section 2.5.15, "Setting output base (:set base)," on page 14 for details on how to set the output base.

By default, Cryptol uses hexadecimal for output, but that can be easily changed using the :set base command:

```
cryptol> :set base=7
cryptol> 12
0<7>15
cryptol> :set base=3
cryptol> 12
0<3>110
cryptol> :set base=20
cryptol> 12
0<20>c
```

As seen in the above examples, Cryptol prints a base prefix with any output. For binary, it is 0b; octal is 0o; decimal has no prefix; and hexadecimal is 0x. For all other bases, the prefix is 0<base>. You can check what the current base is with the command :set.

```
cryptol> :set
   Current output base : 20
   ...
```

Normally, :set prints a lot more information about the current state of the system than you see here, but for the time being we have just shown you the relevant bit.

There are no standards for printing numbers in different bases (say, base 17). So digits ‘10’ through ‘36’ are represented using the lower case alphabet ‘a’ through ‘z’:

```
cryptol> :set base=36
cryptol> [1..36]
[0<36>01 0<36>02 0<36>03 0<36>04 0<36>05 0<36>06 0<36>07 0<36>08
 0<36>09 0<36>0a 0<36>0b 0<36>0c 0<36>0d 0<36>0e 0<36>0f 0<36>0g
 0<36>0h 0<36>0i 0<36>0j 0<36>0k 0<36>0l 0<36>0m 0<36>0n 0<36>0o
 0<36>0p 0<36>0q 0<36>0r 0<36>0s 0<36>0t 0<36>0u 0<36>0v 0<36>0w
 0<36>0x 0<36>0y 0<36>0z 0<36>10]
```
Expressions

Cryptol> :set base=16
Cryptol> x where { x : [32]; x = 2; }
0x00000002
Cryptol> x where { x : [16]; x = 2; }
0x02
Cryptol> x where { x : [8]; x = 2; }
0x2
Cryptol> x where { x : [4]; x = 2; }
0x2

Here's a little Cryptol session:

Normally, Cryptol shows enough 0’s to convey size information. In this case, each digit becomes a nibble, i.e. 4 bits. Of course, if the size of the literal is not a multiple of 4 (in base 16), this heuristic fails. In this case, Cryptol upgrades the size to next multiple of 4 with regard to the number of 0’s to print. Notice that the type of the literal does not change. For instance:

Cryptol> x where { x : [7]; x = 2; }
0x2
Cryptol> x where { x : [13]; x = 2; }
0x0002
Cryptol> x where { x : [2]; x = 2; }
0x2

This same method applies to sizing of output literals in any base.

To verify that this only changes the way the literal is printed, but not its actual size, you can turn type printing on, using the command :set +t:

Cryptol> :set +t
Cryptol> x where { x : [2]; x = 2; }
0x2
: [2]
Cryptol> x where { x : [13]; x = 2; }
0x0002
: [13]

To turn type-printing off, simply use :set -t.

See Section 2.5.5, "Print types after evaluation (+/- t)," on page 12 for details.

Note: Top-level arithmetic expressions in Cryptol are subject to modular arithmetic. This might confuse the unsuspecting user. For instance:

Cryptol> 2*12
0x8
: [4]

Since 2 requires 2 bits, and 12 requires 4, Cryptol defaults 2 to 4 bits and passes them to multiplication. Now, the result is 24, but that is interpreted to fit in 4 bits, i.e. modulo 16. Since 24 is equivalent to 8 modulo 16, we see the result 8. See Section 2.9, "Defaulting," on page 15 in Cryptol: the Language of Cryptography programming guide for more information.
2.2 Interacting with the Interpreter

A line entered at the Cryptol prompt is classified as one of the following four things:

- A command, if the first non-whitespace character is `:`
  
The remainder of the line is interpreted as a command for the Cryptol interpreter.

- A shell escape, if the first non-whitespace character is `!`
  
The remainder of the line is passed to the shell for execution.

- A comment, if the first non-whitespace character is `#`
  
The line is ignored. (These comments are useful in batch processing; see Section 2.4.1, "Batch processing (-b)," on page 9 for more information.)

- An expression, if the first non-whitespace character is anything other than above.
  
The line is considered a Cryptol expression. The expression is evaluated and the result printed.

For example:

```
Cryptol> # comment
Cryptol> ! date
Wed Oct 17 12:12:49 PDT 2001
```

Notice that `#` should be the first non-whitespace character in order for the line to be considered a comment. Otherwise, it won’t be treated specially; that is, it will be considered just as any other character.

You can browse previous lines you have entered by using up/down arrow keys, and edit the input line as usual. In the Unix environment, Cryptol interpreter comes with the GNU Readline library binding. Hence, history search, tab-completion for file names, etc. are also available. The Windows version comes with a simpler readline facility; it can do tab-completion, arrow search, etc., but is rather limited in comparison to the GNU Readline.

The most important command to remember is `:?`, which gives a list of all available commands:

```
Cryptol> :?
Commands: Unambiguous abbreviations are OK, otherwise first match wins.

[ ] indicates the minimum prefix required for a command, if any.

<> shows optional arguments.

expr                      evaluate expression
!command                  shell escape
```
::load file   load program from file (Default extension: .cry)
::reload     re-load current file
::edit <file> edit the file and reload it
::set <option> set/display Cryptol options
::type expr  print type of expr
::browse <pat> list names (matching pat) in scope
::[tr]ace names add names to the trace list
::autotrace names add names to the automatic-trace list
::[sh]owtr <names> list names of trace variables and print values
::deltr pat   remove names matching pat from the trace list
::[def]inition <expr> pretty print definitions in scope, or expr
::print expr  pretty print expression
::fprint file expr pretty print expression to file
::[sc]ript file run script from file
::compile <outfile> compile definitions in scope to C
::[ru]nWith <file> compile and run C source <file> with [args],
  linking current defns
::cd dir      change working directory
::info topic  print online help about topic
::help <pat>  search online help using <pat>
::quit        quit Cryptol

For more info about a particular command cmd, try "::info :cmd" As indicated in the
help message, command names can be abbreviated, as long as no ambiguity
occurs. In case an abbreviation matches more than one command, then the first
match in the above list wins. (The shortest possible abbreviations for each
command are shown between [ ] marks.)

The online help facility can be used to obtain more info on these commands. Use
::info :quit to get help on the :quit command, for instance. Each command is also
explained in detail in this document; see Section 2.7, "Cryptol interpreter
commands," on page 19.
2.3 Programs

Although the Cryptol interpreter can be used to evaluate expressions, it cannot be used to define new functions. Cryptol programs are just like C/C++/Java programs, i.e. they are stored in ASCII files. The interpreter allows loading programs into the interpreter environment using the \texttt{:load} command.

As an example, create a text file. Call it \texttt{Try.cry} and put the following program in it:

\begin{verbatim}
mult : {a}[a][32] -> [32];
mult xs = mults @ last
  where {last = width xs;
    mults = [1] #\[ x * y
    x <- xs
    y <- mults
  ];}
\end{verbatim}

The function \texttt{mult} accepts a variable length vector, elements of which are 32-bit words, and multiplies all the elements of the given sequence. Here is a Cryptol session showing how we can load this file into the interpreter and use it:

\begin{verbatim}
Cryptol> :l Try.cry
Loading "Try.cry".. Checking types.. Compiling.. Done!...
Try> :set base=10
Try> mult [1 .. 5]
120
Try> mult [2 .. 3]
6
Try> mult [1 .. 10]
3628800
\end{verbatim}
2.4 Starting Cryptol

If you start Cryptol with the parameter `-?`, you will see the following command line options:

```
% cryptol -?
Usage: cryptol [OPTION...] [fname]

-b FILE   --batch=FILE  run in the batch mode, reading commands from FILE
-v         --version     show version info
-s         --skip        do not process .cryptolrc or CRYPTOLFLAGS
-?, -h     --help        show this help info
-f FLAGS   --flags=FLAGS pass the flags to the interpreter
```

The next section will explain what these options mean.

2.4.1 Batch processing (-b)

The batch mode is useful for regression testing, where one wants to load a number of files into Cryptol, run several tests, and collect the output for later processing. In this case, one prepares a text file, containing the commands to be run. Then, Cryptol is run with the `-b` option, to process the commands in this file.

As an example, assume we have a file called tests containing:

```
:echo Starting tests
!date
  # a comment in here

5+2

  # set to print types:
  :set +t

3-1
```

(Of course, a normal use of a batch mode would employ `:load` commands to load files into the interpreter.)

Then, we can invoke the batch compiler as:

```
% cryptol -b tests
Starting tests
0x7
0x2
  : [2]
  [:Leaving Cryptol]
```

The option `-b` is actually a shorthand for `- -batch`. Hence, the previous invocation is equivalent to:
% cryptol --batch=tests

In fact, the word batch can be abbreviated as well.
In a batch file, one can use any Cryptol command available; there are no restrictions.

2.4.2 Version (-v)
The command line argument -v prints the version information and quits:

% cryptol -v
Copyright (c) 2001 Galois Connections Inc.
Version: 1.0
%

The long version of this option uses --version, possibly abbreviated.

2.4.3 Skip (-s)
The command line argument --s skips the processing of the Cryptol .cryptolrc initialization script file and the use of any CRYPTOLFLAGS (see Using environment variables).

The .cryptolrc file is a batch file of Cryptol commands. If it exists in the current directory, the commands in .cryptolrc are executed when the Cryptol interpreter is invoked. The skip option prevents the execution of this file.

The long version uses --skip, possibly abbreviated.

2.4.4 Help (-?, -h)
The command line argument -? or -h prints usage help text and quits. The long version uses --help, possibly abbreviated.

2.4.5 Setting flags (-f)
The command line argument -f can be used to set Cryptol flags. The flags are discussed in detail in Section 2.5, "Cryptol interpreter options and flags," on page 11. For instance, to start Cryptol so that it will print types and set the default base to 23, use:

% cryptol -f "+t base=23"

The long version uses --flags=, possibly abbreviated.
All these flags can also be set while the interpreter is running.


## 2.5 Cryptol interpreter options and flags

When you type the command :set at the Cryptol command, you’ll see the current values of flags and options:

```
Cryptol> :set
Current settings: +dp -BOSabceotvw bit base=16 cols=0 depth=10 rtf=-K4m
```

A leading +/- indicate the respective option as being currently on/off:

- **-B** big-endian encoding of words.
- **-o** constant fold value definitions.
- **-O** optimize definitions.
- **-S** specialize polymorphic definitions.
- **-t** print types after evaluation.
- **+d** apply full defaulting to top-level expressions.
- **-c** generate test code (C backend).
- **-b** perform out-of-bounds checks in C code.
- **+p** print result sequences prettily.
- **-e** echo expression string.
- **-a** print bytes (and sequences of bytes) as ASCII.
- **-w** show warnings.
- **-v** verbosity flag.

Current output base : 16
Current output columns : 0
Current print depth : 10
Current mode : bit
Current output directory: /tmp/cryptol.-1393860447
Current run-time flags : -K4m

These options and flags can be set either while Cryptol is running, or by using the -f command line option to the Cryptol executable. (The command line option -f is described in Section 2.4.5, “Setting flags (-f),” on page 10.)

The following will explain what these options and flags are.
2.5.1 Little/Big-endian encoding of words (+/- B)

The B flag toggles whether words are encoded using the big-endian (+B) or little-endian (-B) convention. The default is little-endian. With big-endian, the most significant bit is at index zero. With little-endian, the least significant bit is at index zero.

This is an experimental feature, and only applies in the bit mode.

To toggle between these two modes, use the commands :set +B and :set -B.

2.5.2 Constant folding of value definitions (+/- o)

If this option is turned on, Cryptol will constant-fold monomorphic values after loading a program. If the right-hand side of a binding is determined to be a finite monomorphic constant at compile time, Cryptol will evaluate the definition at compile time, and replace the definition with its value. This is particularly useful when you have tables of values that you’d like to be pre-computed, but that you want to specify algorithmically, instead of going through the error-prone process of typing the tables by hand. (Cryptol will use the bit mode to do the computation at compile time; see Section 2.6.1, "Bit mode," on page 17 for more information.)

To toggle between these two modes, use the commands :set +o and :set -o.

2.5.3 Optimize definitions (+/- O)

This advanced option is useful for users interested in controlling optimizations such as in-lining and constant folding.

To toggle between these two modes, use the commands :set +O and :set -O.

2.5.4 Specialize polymorphic definitions (+/- S)

This advanced option is useful for code generation only (See C mode). To eliminate all polymorphic functions, a specialized concrete instance is generated at each use of the polymorphic function.

To toggle between these two modes, use the commands :set +S and :set -S.

2.5.5 Print types after evaluation (+/- t)

Cryptol can show you the types of results, after each evaluation, controlled by the t flag:

Cryptol> :set -t
Cryptol> 12
0xc
Cryptol> x where { x : {a} [a]; x = 0; }
polymorphic value
: {a} [a]
Cryptol> :set +t
Cryptol> 12
0xc
: [4]

As seen, types are always printed for polymorphic values, regardless of the t flag.

To toggle between these two modes, use the commands :set +t and :set -t.
2.5.6  **Apply full defaul ting to top-level expressions (+/- d)**

Cryptol applies full defaulting to top-level expressions:

```
Cryptol> [1..3]
[0x1 0x2 0x3]
```

The default type of this expression is [2], since the maximum number of bits required to represent the largest value (2) is 2. However, this default commitment to the type [2] can be suspended, and a less specific, polymorphic type assigned:

```
Cryptol> :set -d
Cryptol> [1..3]
<polymorphic value> : {a} (a >= 2, a >= 2) => [3][a]
```

See also Section 2.9, "Defaulting," on page 15 in *Cryptol: the Language of Cryptography* programming guide.

To toggle between these two modes, use the commands :set +d and :set - d.

2.5.7  **Generate test code (C-Backend) (+/- c)**

This option only applies in the C mode (see Section 2.6.3, "C mode," on page 17). When Cryptol translates a Cryptol program to C, it can automatically compile and execute it for you as well. If this option is turned off, Cryptol will only produce the C file, but will not run it. See Section 2.6.3, "C mode," on page 17 for more information on the C mode, and see the command :c (described in Section 2.7.5, "Compile (:compile)," on page 20) for more info on how to use this feature.

To toggle between these two modes, use the commands :set +c and :set - c.

2.5.8  **Perform out-of-bounds checks in C code (+/- b)**

This option only applies in the C mode (see Section 2.6.3, "C mode," on page 17). The Cryptol-generated C code uses C-arrays for storing sized vectors. If this option is turned on, Cryptol will insert out-of-bounds-check code for array accesses in the generated code. This option is useful, as it means that every array access will be guaranteed to be within bounds in the generated code, but of course the check will incur a runtime cost. If this option is turned off, no bounds check will be performed by the generated C-code.

To toggle between these two modes, use the commands :set +b and :set - b.

2.5.9  **Restrict exports (+/- r)**

When Cryptol produces code as output, it will normally export all the names defined in the module. If this is undesirable, you can use the +r option to restrict the name of exports to only those names that are exported from the Cryptol program by an explicit export command.

To toggle between these two modes, use the commands :set +r and :set - r.
2.5.10 **Print result sequences prettily (+/- p)**
Cryptol uses the cols setting to print sequences in a more readable manner. To activate pretty printing, :set +p and :set cols as desired.

To toggle between these two modes, use the commands :set +p and :set -p.

2.5.11 **Echo expression string (+/- e)**
Cryptol’s default mode is to print the value of an expression:

```
Cryptol> [1 2 3] + [4 5 6]
[0x5 0x7 0x1]
```

For clarity, the expression generating the value can be included in the output:

```
Cryptol> [1 2 3] + [4 5 6]
[1 2 3] + [4 5 6] = [0x5 0x7 0x1]
```

To toggle between these two modes, use the commands :set +e and :set -e.

2.5.12 **Print bytes (and sequences of bytes) as ASCII (+/- a)**
Cryptol can print bytes as ASCII character strings:

```
Cryptol> :set +a
Cryptol> {30..100} "\R\Us !"#$%&'()*+,/
0123456789;::<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[^`abcd"
```

To toggle between these two modes, use the commands :set +a and :set -a.

2.5.13 **Show warnings (+/- w)**
Cryptol can detect some situations that might indicate errors, such as sequences of length 0.

To toggle between these two modes, use the commands :set +w and :set -w.

2.5.14 **Verbosity flag (+/- v)**
If turned on, the Cryptol interpreter will print diagnostic messages while running. These messages include any calls to external tools (such as when compiling), or any conditions that merit a warning.

To toggle between these two modes, use the commands :set +v and :set -v.

2.5.15 **Setting output base (:set base)**
Cryptol can display literal numbers in any base from 2 to 36. To set the output base to 18, for example, use :set base=18. By default, this value is set to 16, i.e. hexadecimal.

The reason for the range 2 to 36 is completely practical. Base 1 is too verbose to be useful, and the last letter in the alphabet, “Z”, corresponds to 35 in the order (i.e. A=10, B=11... Z=35).
Section 2.1, "Expressions,” on page 4 contains a discussion about the differences between printing numbers in hexadecimal and other bases and gives an example of the \texttt{:set base} command.

### 2.5.16 Setting output columns (\texttt{:set cols})

Cryptol can control the number of elements of the last dimension of a sequence that are printed on a line. By default, Cryptol prints each row of a sequence on a single line. Sometimes this causes the output to wrap, making readability of the sequence difficult. Settings cols to a value greater than 0 specifies how many elements should appear on a single output line:

```
Cryptol> :set cols=5
Cryptol> [1..20]
[0x01 0x02 0x03 0x04 0x05
 0x06 0x07 0x08 0x09 0x0a
 0x0b 0x0c 0x0d 0x0e 0x0f
 0x10 0x11 0x12 0x13 0x14]
```

### 2.5.17 Setting print depth (\texttt{:set depth})

Sets depth of printing of expressions when using the \texttt{:def} command.

### 2.5.18 Setting Cryptol modes

There are currently three evaluation modes in Cryptol (see also Section 2.6, "Cryptol Modes," on page 17). The following table lists the modes and corresponding names that refer to them in the interpreter:

<table>
<thead>
<tr>
<th>Mode Name</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit mode</td>
<td>bit</td>
</tr>
<tr>
<td>Word mode</td>
<td>word</td>
</tr>
<tr>
<td>C mode</td>
<td>C</td>
</tr>
</tbody>
</table>

For instance, to select the C mode, use:

```
cryptol> :set C
```

When Cryptol first starts, it defaults to the Bit mode.

### 2.5.19 Setting Output Directory (\texttt{:set outdir})

Cryptol uses a directory for temporary files that it generates while running. This directory is automatically created when Cryptol starts, and is deleted upon exit. (On Unix-like hosts, this directory will be created under /tmp.) The \texttt{:set} command shows where this directory is.

It might be desirable to set this directory to a user-defined place. This is achieved by using the \texttt{:set outdir} command:
Cryptol interpreter options and flags

Cryptol> :set outdir=/tmp/myTempCryptolDirectory

The specified directory must exist and must be accessible. If the output directory is explicitly specified by the user, Cryptol will not delete it upon exit.

To set the output directory from the command line, invoke Cryptol as follows:

% cryptol -f "outdir=/tmp/myTempCryptolDirectory"

Normally, end users should not bother with setting outdir themselves, unless they want to retain temporary files generated during a Cryptol session.
2.6 Cryptol Modes

The Cryptol interpreter has three evaluation mode(s). The modes determine how a Cryptol program is compiled or interpreted:

- Bit mode
- Word mode
- C mode

This section will briefly explain each of these modes.

2.6.1 Bit mode

The bit mode in Cryptol is a simple evaluator that models data literally as sequences of bits. It is most useful for interactive development.

The command :set bit selects this evaluator. This is the default mode selected when Cryptol starts.

2.6.2 Word mode

The word mode in Cryptol models data as sequences of computer words, instead of treating each bit independently, as does the bit mode. It is not significantly faster than the bit mode in practice (especially for interactive development), but is sometimes useful because it has a smaller memory footprint on some programs.

The command :set word selects this evaluator. To start Cryptol in the word mode, invoke Cryptol as follows:

```
% cryptol -f word
```

As usual, you can combine this with other flags, as in:

```
% cryptol -f "word base=17" --flags="+t"
```

This call will invoke Cryptol in the word mode, with base 17 for printing out numbers, and the print types option turned on.

2.6.3 C mode

In C mode, programs and expressions are translated to C. When a :load command is issued, the Cryptol file will be translated to C, and compiled. Top-level expressions will be translated to C, linked with the currently loaded file, and then executed to produce the result.

C mode currently has one important restriction: it only compiles monomorphic definitions. Any definitions that are polymorphic in the size of the their arguments will be ignored. Thus, the C mode is best used in conjunction with the +S flag. This flag specializes each polymorphic definition by creating a new definition for each size and shape of argument that the definition is actually used at.

The command :set C selects this evaluator. To start Cryptol in the C mode, invoke Cryptol as follows:
% cryptol -f C
2.7 Cryptol interpreter commands

This section will explain commands available in the Cryptol interpreter in detail.

As explained in Section 2.2, "Interacting with the Interpreter," on page 6, each command can be abbreviated as long as the abbreviation is not ambiguous. If there is an ambiguity, the first command that matches the list given in response to the :? command will be activated.

For each command, you can obtain online help by running :info cmd command within the interpreter.

2.7.1 Version (:version)
The command :version prints version information:

Cryptol> :v
Cryptol Version: 1.4

2.7.2 Load (:load)
The command :load loads a Cryptol script into the interpreter. For instance:

Cryptol> :l trial.cry

will load the file trial.cry from the current directory. The default extension is .cry. If the filename actually ends with the extension .cry, the interpreter will assume and append it by itself; hence, the above example is equivalent to :l trial. There are no restrictions on the extension to be used; any extension (even none) is acceptable.

When a program is loaded, Cryptol will type check the program and compile it into an internal representation. If the file is loaded in the C mode (see Section 2.6.3, "C mode," on page 17), it will be translated to C and compiled by a C compiler as well.

2.7.3 Reload (:reload)
The command :reload will repeat the last :load command. This is useful during interactive development, i.e. when we concurrently edit the program in some other window.

The :reload command doesn’t take any arguments:

Cryptol> :r

2.7.4 Editing (:edit)
The command :edit will run your favorite editor on the current file. Cryptol uses the value of the shell variable EDITOR to determine which editor to use. If the shell variable EDITOR is not set, vi is used as the default editor.

Setting the EDITOR variable depends on the particular operating system and the shell used. For instance, in Linux, using bash, you can say:

% export EDITOR=xemacs
before starting Cryptol. Now, every :edit command will invoke xemacs.

Upon completion of the editing session, the file will be reloaded into Cryptol, unless no changes have been made.

2.7.5 Compile (:compile)

Note: The :compile command is only available in C mode (see Section 2.6.3, "C mode," on page 17).

The command :compile compiles the definitions in scope to C. An optional filename can be supplied as an argument, in which case the translated code will be placed in the specified file. If no filename is given, the code is written to stdout.

For instance, let try.cry be a file containing:

\[
\begin{align*}
\text{perm} : \{a\}[4][a] & \rightarrow [4][a]; \\
\text{perm} [a b c d] &= [d b a c]; \\
\text{try} : [4][32]; \\
\text{try} &= \text{perm} [3 2 1 9];
\end{align*}
\]

The following session shows the usage of the :compile command:

```
% cryptol try.cry
Cryptol v1.0, Copyright (c) 2001 Galois Connections Inc.

Type :? for help
Loading "try.cry".. Checking types.. Compiling.. Done!...
try> :set C
try> :c try.c
Processing: try

try> :q
[Leaving Cryptol]
% cat try.c
/* Generated by Cryptol, version 1.0 */
#include stdio.h
#include "cryptol.h"

extern uint32* main();
uint32*
try ()
{ static uint32 arr0[4]={9UL, 2UL, 3UL, 1UL};
  return (arr0);
}
%
```

If the :compile command is issued without any arguments, the code produced will be written directly to standard output. This is useful when experimenting with the code generators.

2.7.6 Determining types (:type)

The command :type accepts an expression and prints its type.

As an example, assume we have the following definitions in a file named a.cry:

\[
\begin{align*}
x &= y + z \\
\text{where} \{ y = 12
\end{align*}
\]
z : {a} (a >= 3) => [a];
z = 3;
w : [32];
w = 2;
}

Assuming we load this file into Cryptol, we can say:

```
a> :t x
x : {a} (a >= 4) => [a]
a> :t x.y
x.y : {a} [a]
a> :t x.z
x.z : {a} [a]
a> :t x.w
x.w : [32]
```

Note: predicates will not be shown for variables that are not at the top level. (Top-level variables will properly have their predicates shown.) In the above example, for instance, x and w's types are correctly shown, but the types for y and z do not have their predicates (which in the first case is (a >= 4), as determined by type inference, and in the second case is (a >= 3), as explicitly given by the programmer).

This shortcoming, however, is only present when we print the types using the :type command. Local variables do have their predicates considered properly when type inference takes place, for instance.

The command :type can be used to get type information for Cryptol primitives as well:

```
Cryptol> :t +
+ : {a b} ([a]b,[a]b) -> [a]b
Cryptol> :t **
** : {a b} ([a]b,[a]b) -> [a]b
```

### 2.7.7 Browsing names (:browse)

The command :browse can be used to print the names and types of top-level bindings that are currently in scope.

If present, an optional argument to :browse specifies a regular expression against which each name at the top level is matched. In this case, only matching names will be printed.

However, the command :b * is not the same as :b only. The difference lies in the fact that if a regular expression pattern is present, the search is made for the primitive list as well (much like the :type command), but with no patterns present, only the user-defined names that are currently in scope will be printed.

For instance:

```
Cryptol> :b >*
> : {a} ([a],[a]) -> Bit >= : {a} ([a],[a]) -> Bit
>> : {a b} ([c >= lg2(a)) => ([a]b,[c]) -> [a]b
>>> : {a b c} ([c >= lg2(a)) => ([a]b,[c]) -> [a]b
```
Cryptol interpreter commands

To experiment with the command :browse, just load your favorite Cryptol program and type : with and without (a following regular expression) at the command prompt.

2.7.8 Setting and printing options and flags (:set)

The command :set is used for setting flags and options in the interpreter. If used without an argument, it prints the current settings. Each of these flags and options are explained in Section 2.5, “Cryptol interpreter options and flags,” on page 11, along with many examples of the use of the :set command.

2.7.9 Tracing variables (:trace)

Cryptol has a notion of traced variable—a variable whose value is printed whenever referenced. This mechanism provides a simple means for debugging.

To trace a variable named x, simply use the command :trace x. You can also use nested names.

As an example, assume we have the following program, stored in file trc.cry:

```plaintext
f : [4] -> [4];
f x = z * 2
  where { y = x + 1;
  z = y + 3;
}
```

Here is a Cryptol session showing the usage of :trace:

```
trc> :set base=10
trc> :trace f.z
Note: Switching to the "debug" mode..
trc> :showtr
  f.z : [4]
trc> f 3
     14
```

The expression is evaluated and the result 14 is displayed. To see the incremental values of the traced expression f.z, use :showTrace f.z. A list of the values encountered at f.z is collected. In this case, f.z was evaluated once with a value of 7. When an expression may be executed many times, trace output will show the sequence of values:

```
trc> :showtr f.z
  f.z = 7
trc> f 3 + f 1
     8
trc> :showtr f.z
  f.z = 7
      5
```

2.7.10 Inspecting the traced variables list (:showtr)

The command :showtr shows the names and types of all variables that are currently traced. (When non-top-level names are printed, predicates are not shown, as explained in Section 2.7.6, "Determining types (:type)," on page 20.)
2.7.11 Untracing variables (:deltr)
The command :deltr removes names from the traced variables list. It accepts a regular expression pattern, and any name matching the pattern in the traced variables list will become untraced.

To empty the trace list, simply use :deltr *.

2.7.12 Changing current directory (:cd)
Sometimes it is necessary to change the current working directory where the Cryptol executable is running. This is achieved by the command :cd.

Simply issue :cd path-to-the-directory-you-want-to-go to switch to another directory.

2.7.13 Quitting Cryptol (:quit)
The command :quit stops the interpreter and returns to the operating system.

2.7.14 Getting Information (:info)
The command :info (along with :help, explained below) forms the Cryptol online help system.

When invoked with a topic name, the command :info will print online help about the topic, if available. The online help material consists of short descriptions of main Cryptol concepts. For instance, issue :info drop to get a brief description of the drop function.

When invoked with no arguments, :info will print the list of available help topics.

2.7.15 Searching for help (:help)
Unlike :info, the command :help searches help topics for keywords, and suggests matching topics. For instance:

```
Cryptol> :help sequence
Cryptol Online Help topics matching "sequence":
@ @@ # width split join splitBy groupBy drop tail reverse
11 matching entries listed.
Use ":info item" to display the actual help text.
```

The argument to :help can be a regular expression as well:

```
Cryptol> :help *-mode
Cryptol Online Help topics matching "*-mode":
bit-mode word-mode
3 matching entries listed.
Use ":info item" to display the actual help text.
```
Just like `:info`, when invoked with no arguments, `:help` will print the list of available help topics.
2.8 Using environment variables

The C mode of Cryptol depends on external compilers to do compilation. Similarly, Cryptol uses several directories for storing and retrieving data while executing. The user can control these directories and external tools via environment variables. This section will review these flags.

Unfortunately, the exact details of how environment variables are set depend on the operating system and the shell being used. For instance, in Unix-like environments, with the bash shell:

```bash
export CRYPTOL_HC=/usr/local/bin/ghc
```

would set the environment variable CRYPTOL_HC accordingly. Check with the respective OS and shell manuals to determine the conventions used in your system.

2.8.1 The C compiler

The environment variable CRYPTOL_CC is consulted by Cryptol to get the path to the C compiler to be used in the C mode. If undefined, it defaults to gcc.

2.8.2 The CSP library

The environment variable CSP_LIBDIR is consulted by Cryptol to get the path to library archive libCryptol.a. If undefined, it defaults to TOLEVEL/build/lib/galois, where TOLEVEL is the top-level directory where Cryptol is installed in your system. End users should not need to set this directory explicitly, as the default will be correctly configured during Cryptol installation.

2.8.3 The data directory

The environment variable CSP_DATADIR is consulted by Cryptol to get the path to the data files that can be stored and referred to from within Cryptol. Currently, this directory stores AES test data files, but it can contain others as well. If undefined, it defaults to TOLEVEL/build/share/galois, where TOLEVEL is the top-level directory where Cryptol is installed in your system. This variable can be accessed via $datadir whenever a path is entered at the Cryptol prompt. End users should not need to set this directory explicitly, as the default will be correctly configured during Cryptol installation.

2.8.4 The Cryptol directory

The environment variable CRYPTOL_DIR is consulted by Cryptol to get the path to the directory where Cryptol is installed. If undefined, it defaults to TOLEVEL/build/cryptol, where TOLEVEL is the top-level directory where Cryptol is installed in your system. This variable can be accessed via $cryptol whenever a path is entered at the Cryptol prompt. A typical example is:
:cd $cryptol/Examples

which tells Cryptol to change its working directory to the Examples directory, which is distributed under the Cryptol distribution directory.

End users should not need to set this directory explicitly, as the default will be correctly configured during Cryptol installation.

### 2.8.5 Flags to the interpreter

The environment variable CRYFLAGS is consulted by Cryptol to get the options to the interpreter, as discussed in Section 2.5, "Cryptol interpreter options and flags," on page 11. If undefined, it defaults to "", i.e. no flags are used.

For instance, if you want Cryptol to start using big-endian literals, you can set your CRYFLAGS environment variable to +B which will be consulted when Cryptol starts.

### 2.8.6 The temporary directory

The environment variable TMPDIR is consulted by Cryptol to get the top-level directory where the temporary files generated by Cryptol will be stored. If undefined, it defaults to /tmp. Cryptol will create another directory under this one (with a random name), and will use that directory for temporary purposes.

See Section 2.5.19, "Setting Output Directory (set outdir)," on page 15 for more information about how Cryptol stores temporary files.

### 2.8.7 Summary of environment variables

This table summarizes the environments used by Cryptol, and their default values.

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPTOL_CC</td>
<td>Path to the C compiler</td>
<td>gcc</td>
</tr>
<tr>
<td>CSP_LIBDIR</td>
<td>Path to the CSP library</td>
<td>See “The CSP library” on page 25</td>
</tr>
<tr>
<td>CSP_DATADIR</td>
<td>Path to the data directory</td>
<td>See “The data directory” on page 25</td>
</tr>
<tr>
<td>CRYPTOL_DIR</td>
<td>Path to the Cryptol installation directory</td>
<td>See “The Cryptol directory” on page 25</td>
</tr>
<tr>
<td>CRYFLAGS</td>
<td>Options to the interpreter</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>TMPDIR</td>
<td>Top-level directory to store temporary files</td>
<td>/tmp/</td>
</tr>
</tbody>
</table>
3 Verifying Foreign VHDL Implementations
3.1 Introduction

The Cryptol suite provides equivalence checking tools for determining equivalence between Cryptol and Cryptol-derived programs. These tools are typically used within Cryptol for showing equivalence between unoptimized and optimized versions of the same piece of code. For example, while attempting to create a high-performance, pipelined implementation of AES, we use the equivalence checker to insure that we do not introduce any new bugs in the process.

While this functionality is incredibly useful for development of Cryptol programs, it occurred to us that this might also be helpful in showing the equivalence of validated Cryptol specifications with unvalidated, external VHDL source code. To enable this functionality, we have added support to both the Cryptol language and compiler for interacting with foreign VHDL code. Using these extensions, users can import their hand-written VHDL and test its equivalents against high-level Cryptol specifications.

This document is described as a walk through of the task of showing equivalence of foreign VHDL with Cryptol specifications. It assumes that you have previously read, or can easily refer to, the Cryptol reference manual. To elucidate the process, we have structured this paper as a walk through of the process. Further examples can be found at the end of this document.
3.2 Requirements

Proving the equivalence of foreign VHDL with Cryptol specifications requires a version of Cryptol with FPGA and verification support, your favorite text editor, and the Xilinx synthesis tool chain. To perform all the steps in this document, you will also require a CVS client and the GHDL VHDL simulator.
3.3 Getting the Example VHDL

For this walk-through, we will be using an open-source DES3 core available on OpenCores.org. To download this example program, execute the following at the command line (or translate the following into your favorite CVS client):

```bash
# export CVSROOT=:pserver:anonymous@cvs.opencores.org:/cvsroot/anonymous
# cvs login  CVS password: <enter>
# cvs -z9 co 3des_vhdl
```

These steps point CVS to the OpenCores repository, log you into the repository as an anonymous user, and check out the source code, respectively. It seems unlikely that the source code will drift during the lifetime of this document; as of writing (March, 2008) it has been roughly one and a half years since a commit to the CVS repository. However, if you find yourself running into bizarre problems, it may be useful to revert to the version as of March 7th, 2008.

This walk-through assumes you are creating files and executing Cryptol within the VHDL directory of this check-out.
3.4 Step #1: Examining the VHDL, Creating an Extern

The first step in showing the equivalence of arbitrary VHDL is to determine which component is the top-level component in the VHDL definition. In many cases, this information can be found in the documentation for the circuit. In other cases, it may require searching through the VHDL by hand. For our example, the top-level component for DES3 is located in “tdes_top.vhd”, and is called “tdes_top”.

The first step of the equivalence-checking process is to understand the interface for the external component. In our example, the top level interface is as follows:

```vhdl
entity tdes_top is
  port( key1_in: in std_logic_vector(0 to 63); key2_in: in std_logic_vector(0 to 63); key3_in: in std_logic_vector(0 to 63); function_select: in std_logic; data_in: in std_logic_vector(0 to 63); lddata: in std_logic; ldkey: in std_logic; clock: in std_logic );
end tdes_top;
```

As defined by the VHDL, the tdes_top circuit has nine inputs: the three keys, the function select, the data to be encrypted or decrypted, a signal to load the data, a signal to load the key, a reset signal, and the clock signal. It also has two outputs: the encrypted or decrypted data, and a signal indicating whether or not data is ready on the output.

The first step of the equivalence-checking process is to create, in Cryptol, an ‘extern’ declaration for the external component. Generating this specification requires the following information:

- The file the VHDL component appears in
- The name of the VHDL component
- All the input and output variables
- The names of the VHDL types for the component

In some sense, the basic type of this component is simple: it is a function from a tuple with nine components (the inputs) to a tuple of two components (the outputs). However, there are three important things to take note of before proceeding.

First, the reset and clock signals are not, in a sense, inputs to the semantics of the component. Certainly they can have powerful effects on the operation of the circuit, but they do not really make sense in a high-level universe. Cryptol programs, for example, do not explicitly make use of the clock and reset signals; these are introduced by the Cryptol compiler suite behind the scenes. At the same time, of course, we cannot simply ignore them. When writing ‘extern’ declarations, then, we include information about the reset and clock signals when referring to the external declaration, but do not include them in the type.
Second, the type of the component suggests that time is an important factor when discussing the execution of the circuit. The data input line, for instance, is only 64 bits, but the circuit can obviously encrypt arbitrarily large blocks of data. Further, the “out_ready” output suggests that there is some length of time between the input arriving and the output being generated. How do we reflect this time-sensitivity in a Cryptol program?

To do so, we define the type of this external component as a function from an infinite stream of the input tuple to an infinite stream of the output tuple. These infinite streams represent time in Cryptol, and allow us to interact with the component using a series of clock steps.

Finally, a note about the types within the input and output tuples. Both Cryptol and VHDL are intended to be used on low-level data. Thus, the types for both are, more or less, simply the widths of the variables. This makes translating VHDL types to Cryptol types very easy: std_logic is “bit”, and std_logic_vector(0..n-1) is [n].

Putting all of this together, we can define the type for our external component as follows:

```plaintext
extern vhdl("tdes_top.vhd",tdes_top,clock=clock,reset=reset)          extern_3des : [inf](key1_in:[64], key2_in:[64], key3_in:[64], function_select:Bit, data_in:[64], lddata:Bit, ldkey:Bit) -> [inf](data_out:[64],out_ready:Bit);
```

What does this say? It says that we are defining an external function from the file “tdes_top.vhd” that is called ‘tdes_top’ and has a clock signal called ‘clock’ and a reset signal called ‘reset’. When we refer to it from within Cryptol, we’d like it to be named ‘extern_3des’, and it is a function from an infinite series of 7-tuples (with the given types and names in VHDL) to an infinite series of tuples (with the given types and names in VHDL).

Note that Cryptol will automatically assume that an extern’s clock input is called ‘clock’ and its reset input called ‘reset’. Thus, this information can be elided, as follows:

```plaintext
extern vhdl("tdes_top.vhd",tdes_top)          extern_3des : [inf](key1_in:[64], key2_in:[64], key3_in:[64], function_select:Bit, data_in:[64], lddata:Bit, ldkey:Bit) -> [inf](data_out:[64],out_ready:Bit);
```
3.5 Discovering the Interface and Delay

Now that we have an external declaration in Cryptol, we can start interacting with it. Doing so will help us discover its interface and how long it takes for the circuit to begin generating output. Let’s run an example before worrying too much about how the circuit actually works:

Without looking at the documentation, one simple guess for the operation of this circuit is as follows: in one step, we send it the keys and data, set the function select bit, note that the data and keys are there, and then at some later point, we’ll get our output. Given that we don’t know how long this will take, we might write the following test function in Cryptol:

```
  tester (k1,k2,k3,data) = take(100, extern_3des( check # zero )) where {
    check = [(k1, k2, k3, True, data, True, True)];
  };
```

Looking at this in detail, what does it do? First it creates an infinite stream of tuples consisting of the tuple(s) in check followed by zeros. It uses this stream as input to the external component, and grabs the results of the first 100 clock cycles. The check stream consists of the inputs to the circuits: the keys, a True selecting encryption, the data, and Trues informing the circuit that the keys and data are ready.

How do we run this? First, load up cryptol and load the file:

```
  Cryptol version 1.7.2(r), (c) 2001-2008 Galois Inc.
  www.cryptol.net  Type ?: for help  Cryptol> :load "test.cry"
```

Then switch the compiler to VHDL mode:

```
  test> :set VHDL
```

In many cases, your external component will simply require one file: the one noted in your ‘extern’ declaration. In our example case, however, the ‘tdes_top’ component requires a whole host of VHDL files to function. Since Cryptol is not currently capable of determining this information, we have to inform Cryptol of all these other files by setting the “other_vhdl_files” option at the Cryptol command line:

```
  Cryptol> :set other_vhdl_files="add_key.vhd,add_left.vhd,
    block_top.vhd,des_cipher_top.vhd,des_top.vhd,
    e_expansion_function.vhd,key_schedule.vhd,p_box.vhd,s1_box.vhd,
    s2_box.vhd,s3_box.vhd,s4_box.vhd,s5_box.vhd,s6_box.vhd,
    s7_box.vhd,s8_box.vhd,s_box.vhd,tdes_top.vhd"
```

We can then try to execute our test function with some test input:

```
  test> tester (2,3,5,7)
```

This may generate a series of warnings about arithmetic operands that can be safely ignored. What cannot be ignored is the output:

```
  result is undefined  test>
```

What happened? As it turns out, and as experienced VHDL programmers probably already know, it is not necessary for every particular output variable to be defined at any given point of time. The computation being performed on an
input may not be complete after one clock cycle, causing the related output port to be an intermediate (or simply random) value. Cryptol interprets these undefined outputs literally, and reports that the output of our test is undefined because it contains undefined output.

This is good to know, but not very helpful for our testing purposes. To aid in testing, you can override this behavior by setting a global option within the Cryptol command line:

```
test> :set ghdl_unknowns_are=false
```

This causes Cryptol to translate any undefined outputs into False (or 0, depending on the type). If we now rerun our test program, we get the following:

```
test> tester(2,3,5,7)  [ ... elided warnings ... ]  [(0x0000000000000000, False) (0x0000000000000000, False) (0x0000000000000000, False) (0x0000000000000000, False) (0x0000000000000000, False) (0x0000000000000000, False) (0x0000000000000000, False) [ ... many more ... ]]
```

Well, all the zeros and Falsees suggest that either this circuit takes more than 100 cycles to generate output or we got the interface to the circuit wrong. Given that we just made up the simplest possible interface, the latter seems more likely. At this point, we can continue to guess at the interface and then check it, or we can consult the documentation. As the latter seems more efficient, open up the PDF found in the ‘Docs’ directory of the ‘3des_vhdl’ tree and flip to the last page.

The last page -- containing Figure 2 -- shows the interface to the circuit. First, one loads the keys on the inputs and sets the ‘ldkey’ signal high. Then we wait a cycle. After that, we load the data on the input and set the ‘lddata’ signal high, iteratively sending consecutive 64-bit chunks every cycle. Our next step is to translate this into Cryptol. Remembering that the infinite series of inputs is infinite over time, we can regenerate our tester (which works on one 64-bit chunk) fairly easily:

```
tester (k1,k2,k3,data) = take(100, extern_3des(check # zero)) where [    check0 = (k1, k2, k3, True, data, False, False);    check1 = (k1, k2, k3, True, data, False, True);    check2 = (k1, k2, k3, True, data, False, False);    check3 = (k1, k2, k3, True, data, True, False);    check4 = (k1, k2, k3, True, data, False, False);    check = [check0 check1 check2 check3 check4]; ];
```

What does this do? It sets up a sequence of five inputs over five cycles, followed by an infinite number of zero inputs. Looking at the individual tuples and the input of the external component, they perform the following steps:

* check0 == Set the keys and data, but clear the load bits   * check1 == Set the “ldkey” bit   * check2 ==> Clear the “ldkey” bit   * check3 == Set the “lddata” bit   * check4 == Clear the “lddata” bit

This performs exactly the series of steps described in the documentation, choosing to just keep the key and data bits defined for simplicity. We can run this in Cryptol, getting:

```
Cryptol> :set VHDL ghdl_unknowns_are=false  Cryptol> :load test.cry  Loading “test.cry”.. Checking types.. Processing.. Done!  test> :set other_vhdl_files="...
```

```
test> tester(2,3,5,7)  [ ... warnings elided ... ]  [(0x0000000000000000, False)
```

| 34 | Galois, Inc |
Ah-hah! A series of clocks pass with zero or undefined input, and then, after 59 cycles, we get the encrypted data! Let’s refine our tester to be less verbose:

```
tester(k1,k2,k3,data) = take(1, drop(59, extern_3des( check # zero ))) where {
  check0 = (k1, k2, k3, True, data, False, False);
  check1 = (k1, k2, k3, True, data, False, False);
  check2 = (k1, k2, k3, True, data, False, False);
  check3 = (k1, k2, k3, True, data, False, False);
  check4 = (k1, k2, k3, True, data, False, False);
  check = [check0 check1 check2 check3 check4];
};
```
And then run it again:
Cryptol> :set VHDL ghdl_unknowns_are=false  Cryptol> :load test.cry  Loading “test.cry”.. Checking types.. Processing.. Done!  test> :set other_vhdl_files=”…”  test> tester(2,3,5,7)  [... warnings elided ... ]  [(0x649782a13c894f07, True)]  test>

Now, before proceeding, let’s perform a sanity check. Is this value, 0x649782a13c894f07, really the result we should get when encrypting the 64-bit value ‘7’ with the keys ‘2’, ‘3’, and ‘5’? Since we’re going to be showing equivalence with Cryptol anyways, let’s check with Cryptol’s reference specification.

Cryptol contains a reference specification of DES, but not DES3. That’s OK, though; defining DES3 in terms of DES is very simple:

```plaintext
   include “DES.cry”;
   des3_enc : ([64],[64],[64],[64]) -> [64];
   des3_enc (k1,k2,k3,data) =
     blockEncrypt(k3, dx) where {
      ex = blockEncrypt(k1, data);
      dx = blockDecrypt(k2, ex);
    };
```

We can now run this within Cryptol, and see if we get the same answer as we got from the VHDL simulation:

Cryptol> :load test.cry  Loading “test.cry”.. Including “DES.cry”.. Checking types.. Processing.. Done!  test> des3_enc(2,3,5,7) 0x649782a13c894f07  test>

They match! Good. This means that based on our test input and our understanding of the VHDL component, we are getting the right output. Now we just need to verify that the two generate equivalent output for all inputs.
3.6 Generating Functions For Equivalence Checking

In order for the Cryptol equivalence checker to determine if two functions are equivalent, the types of the functions must be finite, monomorphic, and equal. Or, in other words, that:

* The types not involve any infinite streams. (finite) * The types not involve any type variables. (monomorphic) * The types look the same. (equal)

Checking the types of our two functions with Cryptol, we discover that our types are finite and monomorphic but not equal:

Cryptol> :load test.cry Loading “test.cry”.. Including “DES.cry”.. Checking types.. Processing.. Done! 

...: des3_enc des3_enc : ([64],[64],[64],[64]) -> [64] 

tester : ([64],[64],[64],[64]) -> [1]([64],[Bit]) test>

So how do we fix this? We can either choose to fix this by modifying ‘des3_enc’ (our gold standard) or ‘tester’ (our routine under test). We’ll choose to modify ‘tester’, just to make absolutely sure we don’t mess up our standard with the wrapper code. How do we do this? Well, the input types look equivalent: 4-tuples of 64-bit values. The outputs need to be modified, however. We need to turn a one-element stream containing a tuple of the output and the ready bit into just the output value.

Let’s start by clarifying some of what we’re doing in the ‘tester’ function:

tester (k1,k2,k3,data) = output where { // Generate the head of the input stream check0 = (k1, k2, k3, True, data, False, False); check1 = (k1, k2, k3, True, data, False, True); check2 = (k1, k2, k3, True, data, False, False); check3 = (k1, k2, k3, True, data, True, False); check4 = (k1, k2, k3, True, data, False, False); check = [check0 check1 check2 check3 check4]; // Run the external component component_output = extern_3des( check # zero ); output_without_delay = drop(59, component_output); output = take(1, output_without_delay); }; All this modification does is make explicit each of the output-conversion steps we’re performing. First, we get the output from the external component. Then we, pull off the cycles waiting for the output to show up. Then we grab the first one. Just to be sure, let’s run this and make sure we haven’t messed anything up:

Cryptol> :set VHDL ghdl_unknowns_are=false Cryptol> :load test.cry Loading “test.cry”.. Checking types.. Processing.. Done! 

...: tester(2,3,5,7) [ ... warnings elided ... ] [(0xd49782a13e894f07, True)] test>

Now all we have to do is modify the tester so that instead of getting a single-element stream containing a tuple, we just get the output we want. We can do this pretty simply using a pattern match for ‘output’ rather than a simple binding:

tester (k1,k2,k3,data) = output where { // Generate the head of the input stream check0 = (k1, k2, k3, True, data, False, False); check1 = (k1, k2, k3, True, data, False, True); check2 = (k1, k2, k3, True, data, False, False); check3 = (k1, k2, k3, True, data, True, False); check4 = (k1, k2, k3, True, data, False, False); check = [check0 check1 check2 check3 check4]; // Run the external component component_output = extern_3des( check # zero ); output_without_delay = drop(59, component_output); {((output,ready)) =
take(1, output_without_delay);  ]; This should get us where we want to be.

Checking in Cryptol:

Cryptol> :set VHDL ghdl_unknowns_are=false  Cryptol> :load test.cry  Loading “test.cry”.. Checking types.. Processing.. Done!  test> :set other_vhdl_files=”…”  test> tester(2,3,5,7)  [... warnings elided …]  0xd49782a13c894f07  test> :set bit test> des3_enc(2,3,5,7)  0xd49782a13c894f07  test> :t tester  tester : (((64],[64],[64],[64]) -> [64])  test> :t des3_enc  des3_enc : (((64],[64],[64],[64]) -> [64])  test>

Good. Our outputs and types are equivalent. Now to move on to equivalence checking.
3.7 Equivalence Checking

The final phase of this process is to generate the formal models of the functions and to check for equivalence. These steps each take some amount of time to execute.

One approach to dealing with this time delay is to create a Cryptol batch file to perform all the steps for you. The advantage of this is that you can give Cryptol all the steps you want to execute, start Cryptol, and then walk away from it. The disadvantage is that if there is an error in one of the first steps, you may either miss it or waste time. In contrast, you could perform all of the steps at the Cryptol command line. This allows you to catch errors immediately, but means you spend a lot of time waiting for Cryptol to finish running intermediate tasks before typing in the next command.

For this tutorial, we’re going to choose a middle ground, and iteratively develop two different batch files. The first will generate a formal model from the VHDL, and the second will check for equivalence. The advantage of this approach is that we can make sure the first script is correct before moving on to the second. Further, if we wanted to later ship this equivalence check off to a third party, we can simply concatenate the two files to run the equivalence check in one action.

To begin, we want to generate a formal model for our function under test. To do so, we create a simple text file containing the following actions for Cryptol:


These steps do the following:

1) Switch Cryptol into NSIM mode, which is an FPGA-based mode that allows for the creation of formal models. 2) Load our test file. 3) Notify Cryptol of any other VHDL files. 4) Generate the formal model, sending the output to the file “tester.jaig”.

Now let’s save this to “test1” and run it:

# cryptol -b test1 Loading “test.cry”.. Including “DES.cry”.. Checking types.. Processing.. Done! [... elided warnings ...] WARNING:Xst:1336 - (*) More than 100% of Device resources are used ERROR:Pack:18 - The design is too large for the given device and package. Command map returned unsuccessful exit status 2 Check log file for details: mainRBZZWC.mrp [Leaving Cryptol] #

Unfortunately, our external VHDL is too large to fit on Cryptol’s default FPGA board. In order to check for equivalence, we need to switch the board to something larger. Modifying the first line of our “test1” batch file from:

:set NSIM

to

:set NSIM fpga_part=xc4vlx60-12ff668

Should do it. Trying again after this change (this takes a little over 10 minutes on our machines):

# cryptol -b test1 Loading “test.cry”.. Including “DES.cry”.. Checking types.. Processing.. Done! [... elided warnings ...] [Leaving Cryptol] #
Great! No errors. We now have a file, ‘tester.jaig’, containing a formal model of our external VHDL component. At this point, all we have to do is check that that model is equivalent to our Cryptol standard code. We can do this by creating the following batch file that we’ll call ‘test2’:

:set LLSPIR :load “test.cry” :eq des3_enc “tester.jaig”

This file does the following:

1) Sets Cryptol into LLSPIR mode. LLSPIR can be used to generate formal models of Cryptol code that does not use extern statements. Since our other function, ‘des3_enc’, does not use any extern functions, we’re OK on that front. We could again use NSIM, instead, but LLSPIR mode appears to be somewhat faster.

2) Load our test file.

3) Check for equivalence between the function ‘des3_enc’ and the formal model in the file “tester.jaig”.

Note that when given a Cryptol function name rather than a file name, Cryptol will automatically generate the formal model for the function and then test for equivalence. This is useful shorthand. Technically, we could simply enter NSIM mode and test for equivalence directly, using “:eq des3_enc tester”. Again, however, generating formal models in LLSPIR mode is faster than doing so in NSIM mode, so we’ve chosen to only use NSIM mode when necessary.

Finally, we check for equivalence by running this batch file (this takes a little under an hour on our machines):

# cryptol -b test2  Loading “test.cry”.. Including “DES.cry”.. Checking types.. Processing.. Done!  True  #

Success! Our downloaded VHDL component is equivalent to our Cryptol specification.

If the component had not been equivalent, Cryptol would have printed “False” and shown a counterexample. You can then try running the counterexample in your functions, as usual.
3.8 Dealing With Overallocated Pins (a.k.a. IOBs)

When generating formal models of VHDL code, Cryptol takes a somewhat naive approach and assumes that each bit of the input and output will be appearing on a physical pin of the chip. In many cases, this works just fine. In other cases, the function will use more bits than the chip has available to it.

Future versions of Cryptol may use a different mechanism, but for now this can be a problem. If you get an error about overallocating IOBs while attempting to synthesize your code for formal verification, it is necessary to rework your Cryptol wrapper (and your Cryptol standard) such that your functions take their input over a series of steps over time.

For example, the following function could not be mapped onto a Xilinx board, as it required too many pins:

\[
\text{aes256 : ([256],[128])} \rightarrow [128]
\]

The input to this function is a 256-bit key and a 128-bit data block to encrypt. To get this to fit on the board, we rewrote the function, passing the key in over two clock cycles:

\[
\text{aes256 : [2]([128],[128])} \rightarrow [128]
\]

At first glance, this may seem like it is using even more space than the previous. After all, \(2 * (128 + 128) + 128\) is larger than \(256 + 128 + 128\). However, in the second version, the input is passed over time, rather than over space. In the first clock cycle, the first tuple is sent to the circuit. In the second, the second clock cycle is sent. Thus, at any point, only 384 pins \((128 + 128 + 128)\) are in use, which is within the size of the chip.

Rewriting the Cryptol gold standard for this case is trivial:

\[
\text{aes256 [(key\_lo, data) (key\_hi, junk)]} = \text{encrypt(key\_lo # key\_hi, data)};
\]

Rewriting the Cryptol wrapper for the external component, however can be slightly tricky. The basic idea is that we will write two wrappers, instead of our usual one. The first wrapper will, like the extern component, be from an infinite type to an infinite type. The second wrapper will interface with the first wrapper much like our ‘tester’ wrapper did, above.

Writing the first wrapper is somewhat formulaic. The basic idea is that we’re going to construct a state machine for passing the data to the extern. In state 0, we are reading in the low bits of the key and the data. In state 1, we are reading in the high bits of the key and the data. In state 2 and beyond, we are passing on subsequent data. The following pseudocode demonstrates how to do this:

\[
\text{split\_aes : [inf]([128],[128]), Bit} \rightarrow [inf][128];  \text{split\_aes} \text{ ins} = \text{aes\_outs} \quad \text{where} \quad \text{aes\_outs} = \text{aes256(aes\_inputs)};  \text{aes\_inputs} = [\text{inf} (\text{key\_lo} \# \text{key\_hi, data})]  \triangle \ (\text{state, key\_lo, key\_hi, data}) <\text{buffer} \quad [\text{inf}];  \text{buffer} = \text{[undefined] #}  \quad [\text{inf} \text{if request then} (0, \text{key\_in, zero, data\_in}) \quad \text{else if state == 0 then} (1, \text{key\_lo\_prev, key\_in, data\_prev}) \quad \text{else if state} = 0 \quad (2, \text{key\_lo\_prev, key\_hi\_prev, data\_in})  \quad \text{| (key\_in, data\_in, request)} <\text{ins}  \triangle (\text{state, key\_lo\_prev, key\_hi\_prev, data\_prev}) \triangle [\text{inf}];]
\]

Cryptol Toolset Users Guide
Note that this sort of state machines add cycles to the delay between when your input is presented to the function and when the output appears in the output stream. If you’ve previously calculated the delay using a ‘tester’ function, as suggested above, then you’ll have to redo your checks. However, the amount of additional time the circuit takes is usually highly predictable; in the case of the circuit the above example is based on, it added one additional cycle.