Introduction to Cryptol and High-Assurance Crypto Engineering

Dylan McNamee, Adam Foltzer
Galois, Inc.

galois

Goals

- Understand the purpose of the Cryptol language and its role in high-assurance engineering
- Create and modify cryptographic programs in Cryptol
- Use the interactive Cryptol interpreter to develop, test, and prove properties about Cryptol programs
- Learn how to participate in the Cryptol open source community
- Anything else?

Outline

- Introduce Cryptol (~30min)
- Learn Cryptol via examples and exercises (~45min)
  - Hands-on lab for Chapters 2-3
  - Get familiar with the Cryptol interpreter
  - Classical cryptosystems: Caesar, Vigenère, Scytale
- Break (~15min)

Outline

- Introduce property-driven development (~15min)
- Property-driven development exercises (~30min)
  - Hands-on lab for Chapter 5
  - Use `:check`, `:sat`, and `:prove`
- ZUC cipher demo and closing discussion (~15min)

Why formal methods matter

- How can software and systems be made robust (safe, secure, correct) in a cost-effective manner?
- How can one obtain high assurance that a design has been faithfully implemented?
- How can we ensure that other people’s systems are secure?
- How can we compose a secure solution from black-box components?

Vision for system software

Imagine software built with the same rigor and analysis as other engineers build bridges

- Let the software itself be trustworthy
  - Software artifacts to speak for themselves
  - Reduce reliance on the process that created them
- Use mathematical models to enable tractable analysis
  - Executable models and formal methods
  - A model is an abstraction that allows thought at a higher level
- Follow open standards
  - Build individual components with high internal integrity
  - Maximize interoperability

Cryptol is a "formal method"

- **"Formal Methods"** is a body of verification techniques that work by building a **mathematical model** of an artifact and **proving properties** about it
- Formal methods are complementary to testing
  - Testing techniques generate *weak* evidence about the *real artifact* [Worry: Have I tested enough?]
  - Formal methods generate *strong* evidence about a *model of the artifact* [Worry: Is the model faithful enough?]

Early Reference

Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge, England, June 1949. University Mathematical Laboratory.

Lack of clear reference implementations

```
#define MDS_GF_FDBK 0x169
#define LFSR1(x) ( ((x) >> 1)  ^ (((x) & 0x01) ?   MDS_GF_FDBK/2 : 0))
#define LFSR2(x) ( ((x) >> 2)  ^ (((x) & 0x02) ?
            MDS_GF_FDBK/2 : 0)
            MDS_GF_FDBK/4 : 0))
#define Mx_1(x)  ((DWORD)  (x))
#define Mx_X(x)  ((DWORD) ((x) ^ LFSR2
#define Mx_Y(x)  ((DWORD) ((x) ^ LFSR1
#define M00     Mul_1
#define M01     Mul_Y
return  ((M00(b[0]) ^ M01(b[1]) ^
     M02(b[2]) ^ M03(b[3]))    ) ^
    ((M10(b[0]) ^ M11(b[1]) ^
     M12(b[2])^ M13(b[3])) << 8)^
    ((M20(b[0]) ^ M21(b[1]) ^
     M22(b[2])^ M23(b[3])) <<16)^
    ((M30(b[0]) ^ M31(b[1]) ^
```

$$
\begin{aligned}
x_i &= \lfloor X/2^{8i} \rfloor \bmod 2^8 \qquad i = 0, \ldots, 3 \\
y_i &= s_i[x_i] \qquad i = 0, \ldots, 3
\end{aligned}
$$

$$
\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} \cdot & \cdots & \cdot \\ \vdots & \text{MDS} & \vdots \\ \cdot & \cdots & \cdot \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}
$$

$$
Z = \sum_{i=0}^{3} z_i \cdot 2^{8i}
$$

It's hard to relate implementations to the underlying math

galois

Cryptol

- Domain-specific language for specifying cryptographic algorithms*
- Size-polymorphic, statically-typed with type inference
- Lightweight Haskell-style module system
- Interpreter with a read-eval-print loop (REPL)
- Transparent integration with SAT and SMT solvers for proving properties expressed in Cryptol

* It's good for more than just cryptography, which we'll see throughout the day

Cryptol specifications

- File of mathematical definitions
  - Two kinds of definitions: values and functions
  - Definitions may be accompanied by a type declarations (a signature)
- Definitions are computationally neutral
  - Cryptol tools provide the computational content (interpreters, compilers, code generators, verifiers)

```
x :  [4][32]
x = [23, 13, 1, 0]

F : ([16],[16]) -> [16]
F (x, x') = 2 * x + x'
```

- Domain-specific data and control abstractions
  - Sequences
  - Recurrence relations (not for-loops)
- Powerful data transformations
  - Data may be viewed in many ways
  - Machine independent
- Algorithms parameterized on size
  - Size constraints are explicit in many specs
  - Number of iterations may depend on size
  - A sized type system captures and maintains size constraints

galois

Basic Cryptol commands

- Load file within the interpreter:
  - :m AES – *loads a module (AES.cry)*
  - :l myprogram.cry – *loads a file*
- Reload the current file
  - :r
- Edit the current file
  - :e
- Browse current definitions
  - :b
- Find the type of an expression
  - :t myfunction
  - :t 1+2
  - :t width
- Set output base
  - :set base=8
- Show 8-bit sequences as ASCII
  - :set ascii=on
- Quit Cryptol
  - :q
  - Ctrl-D
- Show all commands:
  - :help
- Showing what can be set
  - :set
- Use the tab key
  - Completion helps!

|galois|

Datatypes

- Homogeneous sequences
  - `[False, True, False, True, False, False, True]`
  - `[[1, 2, 3, 4], [5, 6, 7, 8]]`
- Numbers are represented as sequences of bits (“words”)
  - `123, 0xF4, 0b11110100`
- Quoted strings are just syntactic sugar for sequences of 8-bit words
  - `“abc” = [0x61, 0x62, 0x63]`
- Heterogenous data can be grouped together into tuples
  - `(13, “hello”, True)`
- Records: Tuples with named fields
  - `type Point3D = { x:[16], y:[16], z:[16] }`
  - `p1 = { x = 22, y = 35, z = 18 }:Point3D`
  - `p1.x = 22`

Standard operations

- Arithmetic operators
  - Result is modulo the word size of the arguments
  - `+ - * / % ^^`
- Boolean operators
  - From bits, to arbitrarily nested matrices of the same shape
  - `&& || ^ ~`
- Comparison operators
  - Equality, order
  - `== != < <= > >=`
  - returns a Bit
- Conditional operator

```
evenNum x = x + (if x % 2 == 0 then 0 else 1)
```

galois

Sequences

- Sequence operators
  - Concatenation (#), indexing (@)
  - `[1 .. 5] # [3, 6, 8] =`
    `[1, 2, 3, 4, 5, 3, 6, 8]`
  - `[50 .. 99] @ 10 = 60`

- Shifts and Rotations
  - Shifts (<<, >>), Rotations (<<<, >>>)
  - `[0, 1, 2, 3] <<  2 = [2, 3, 0, 0]`
  - `[0, 1, 2, 3] <<< 2 = [2, 3, 0, 1]`

Sequence comprehensions

- Comprehension notion borrowed from set theory
- Applying an operation to each element

```
[ 2*x + 3 | x <- [1, 2, 3, 4] ] # [15]
  = [5, 7, 9, 11, 15]
```

- This is the most-used control structure in Cryptol

Traversals

- Cartesian traversal

```
[ [x, y] |  x <- [0, 1, 2], y <- [3, 4] ]
    = [[0, 3], [0, 4],
       [1, 3], [1, 4],
       [2, 3], [2, 4]]
```

- Parallel traversal

```
[ x + y |  x <- [1, 2, 3]
        |  y <- [3, 4, 7, 11, 18] ]
   =  [4, 6, 10]
```

Exercises 1:7–43

Types

- Expressions have static, strong types
- The type system very flexibly keeps track of constraints
- Monomorphic (a single specific type)
  - `(2>=3) : Bit`
  - `[0x02, 0x14, 0x05, 0x30] : [4][8]Bit`
  - `(3,5,True) : ([8],[32],Bool)`
  - `F : ([16],[16]) -> [16]`

"Bit" here may be left out

- Polymorphic (a family of types)
  - `[2, 4, 5, 3] : {a}  [4][a]Bit`
  - `tail : {a, b} [1 + a]b -> [a]b` *(a is size, b is shape)*

Exercises 1:44–47

Modules

- A module `Foo` is defined in the file `Foo.cry`

```
module Foo where

import Bar

type K = [128]

f : K -> K
f k = k && 1
```

- `import` statements go before declarations
- Files without a module declaration are implicitly named `Main`

|galois|

Functions

- Functions are *mathematical functions*
  - Not procedures that return values

- Functions can have multiple arguments, and can return multiple results in a tuple

```
XYandXplusY : [8] -> [8] -> ([8], [8])
XYandXplusY x y = (xy, x + y)
     where xy = x * y
```

- Functions don’t have to be named

```
(\(x, y) -> 2 + x*y) : ([a], [a]) -> [a]
```

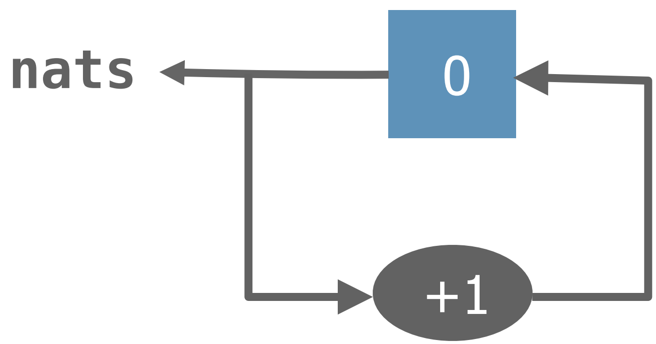
Exercises 1:48–54

galois

Recurrences

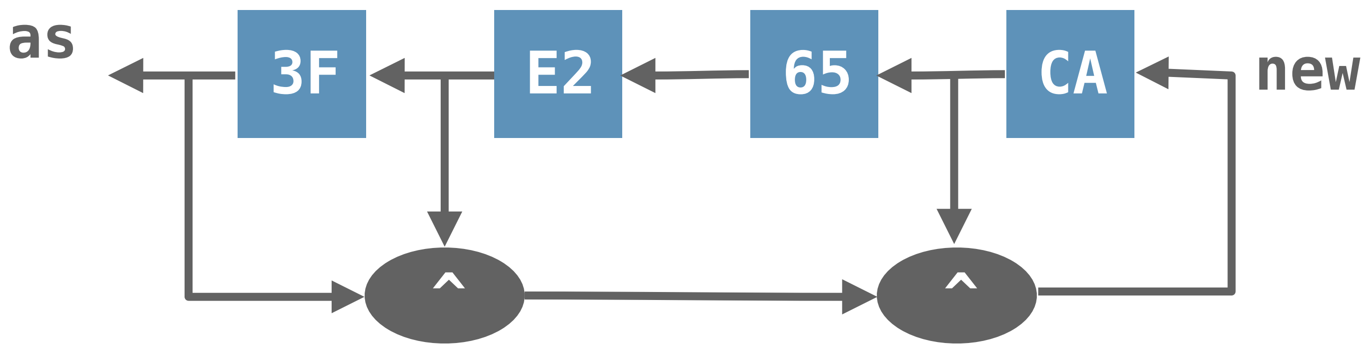
- Textual description of shift circuits
  - Follow mathematics: use *stream-equations*
  - Stream-definitions can be *recursive*
    - and can define infinite-length streams

```
nats = [0] # [ y+1 | y <- nats ]
```



Stream equations

```
as  = [0x3F, 0xE2, 0x65, 0xCA] # new
new = [ a ^ b ^ c | a <- as
                  | b <- drop`{1}as
                  | c <- drop`{3}as ]
```



| galois |

Basic functions

- Some are built-in, others defined in Cryptol's prelude
- You can view the prelude by invoking Cryptol with no argument, using **`:edit`**

```
> take`{2}[2 .. 10]
[0x2, 0x3]
> drop`{2}[2 .. 10]
[0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0xa]
```

Basic functions

```
> :t groupBy
groupBy : {each, parts, elem}(fin each)
            => [parts * each]elem
            -> [parts][each]elem
> groupBy`{2}[1 .. 100]
[[0x01, 0x02], [0x03, 0x04], … ]
> (split [1 .. 100])[2]_ // type inference
Assuming a = 7
[[0x01, 0x02, 0x03,…],
 [0x33, 0x34, 0x35…]]
```

A few more basics

- Cryptol's **`zero`** is very flexible: you can assign it any shape:

```
> zero:[2][2]Point3D
[[{x = 0x0000, y = 0x0000, z = 0x0000},
  {x = 0x0000, y = 0x0000, z = 0x0000}],
 [{x = 0x0000, y = 0x0000, z = 0x0000},
  {x = 0x0000, y = 0x0000, z = 0x0000}]]
```

- and you can negate it:

```
> ~zero:Point3D
{x = 0xffff, y = 0xffff, z = 0xffff}
```

Where clauses help with formatting

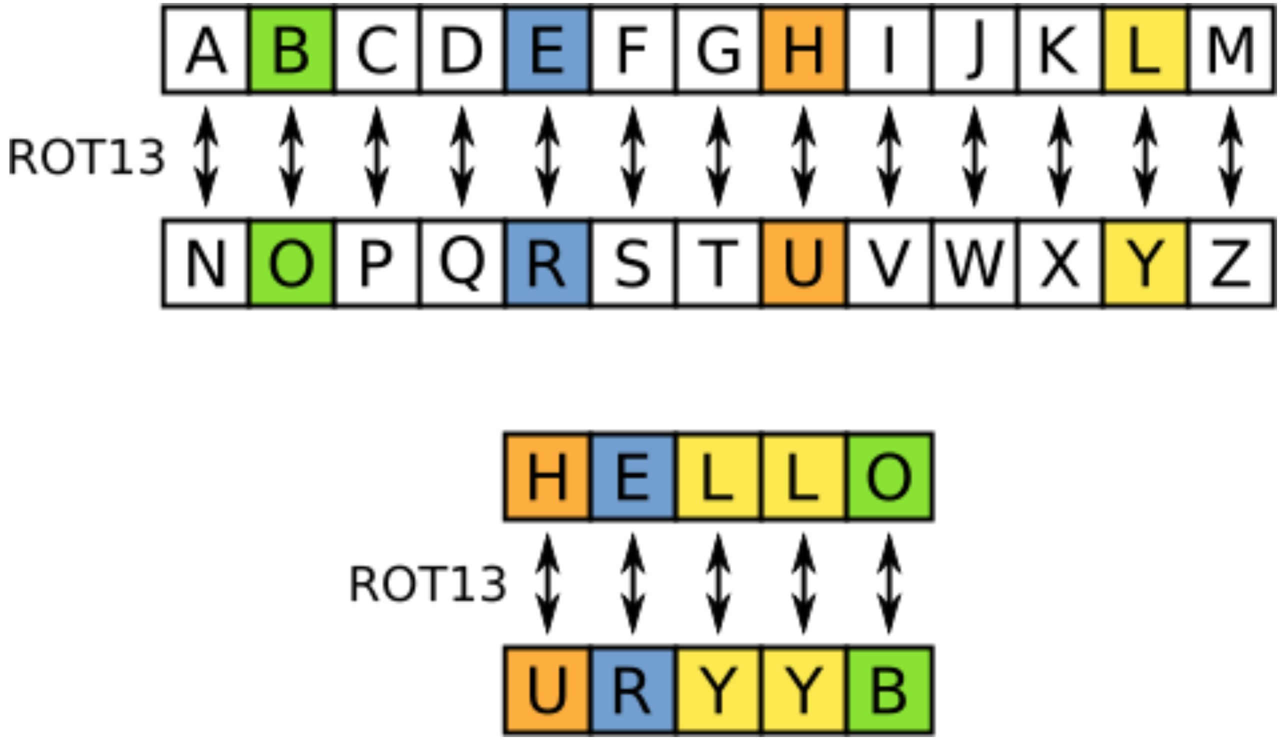
- At the repl:

```
> groupBy`{3}xs where \
  xs = [ x * 3 | x <- [1..99] ]
[[0x03, 0x06, 0x09], [0x0c, 0x0f,
0x12], …
```

- In function definitions:

```
isValid x = withinRange && isEven where
  withinRange = x > 5 && x < 10
  isEven = (x && 1) == 0
```

ROT13



"ROT13 table with example" by Benjamin D. Esham (bdesham) - Based upon ROT13.png by en:User:Matt Crypto. This version created by bdesham in Inkscape.This vector image was created with Inkscape.. Licensed under Public domain via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:ROT13_table_with_example.svg#mediaviewer/File:ROT13_table_with_example.svg

ROT13

- Substitution cipher
  - Each letter in the plaintext is replaced by a corresponding letter in the ciphertext
- ROT13(ROT13(x)) == x
- Hello World of cryptography (and Cryptol)

```
ROT13 : [n][8] -> [n][8]
ROT13 msg = [ shift x | x <- msg ]
  where map     = ['A' .. 'Z'] <<< 13
        shift c = map @ (c - 'A')
```

ROT13

- `map = ['A' .. 'Z'] <<< 13`
  - `map @@ [0,1,2,3,4,13] == "NOPQRA"`
- `shift c = map @ (c - 'A')`
  - `shift 'C' == 'P'`
    - `('C' - ‘A') == 2`
    - `map @ 2      == ‘P'`
- `[ shift x | x <- msg ]`
  - Maps the `shift` function over each character in the message

ROT13

- With `ROT13` defined in `ROT13.cry`:

```
Cryptol> :l ROT13.cry
Loading module Cryptol
Loading module Main
Main> :set ascii=on
Main> ROT13("HELLOWORLD")
“URYYBJBEYQ"
Main> ROT13(ROT13("HELLOWORLD"))
"HELLOWORLD"
```

galois

Lab time

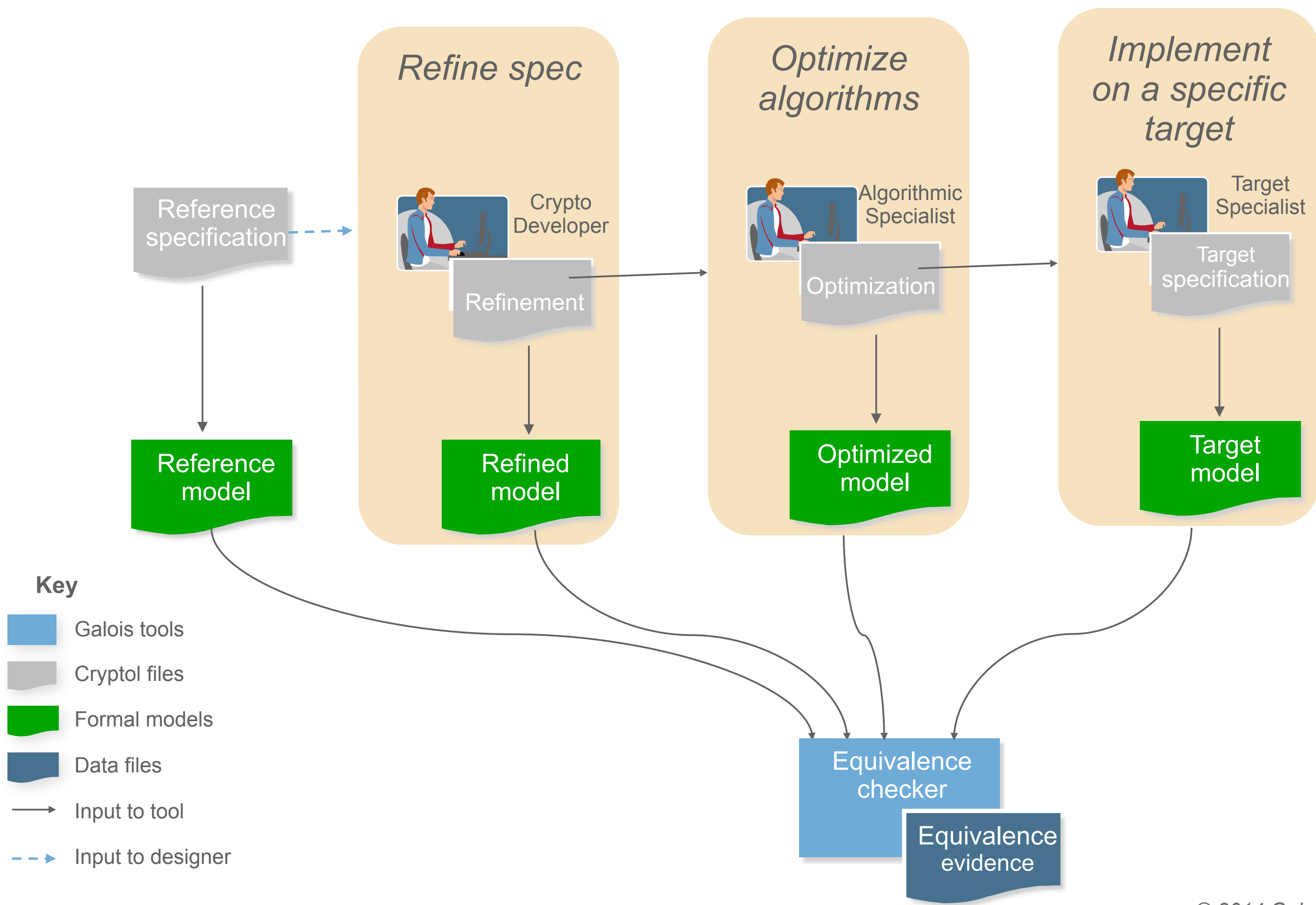
- Chapter 2: Crash Course
  - Detail on basic language structures
  - More exercises than we have time for; skim and refer to later
- Chapter 3: Classic Ciphers
  - Substitution ciphers: Caesar, Vigenère
  - Try to make it through Exercises 1-10
- Reconvene for intro to property-driven development at 10:30am

galois

Why properties in Cryptol?

- Properties could express
  - Correctness properties of a specification (for validation)
  - Equivalence of a high-level specification and an “implementation-specification”
  - Design principles that guide the development of a derived specification
  - The correctness of a compilation path
  - Equivalence of an implementation (outside Cryptol) and a specification

Design-refinement correctness



Properties in Cryptol

- Cryptol values of type `Bit`
  - `property two_plus_two = 2 + 2 = 4`
- Cryptol functions returning type `Bit`
  - `property refl x = x == x`
- Arguments to properties can be any type

galois

Properties in Cryptol

- Non-function properties good for test vectors
  - `property ROT13_hello =`
    `ROT13("HELLO") == "URYYB"`
- Function properties good for broad statements
  - `property plus_id_l x =`
    `0 + x == x`
  - `property plus_assoc x y z =`
    `x + (y + z) == (x + y) + z`

|galois|

Randomized testing

- `:check` command runs a property with random values (like QuickCheck)

```
Cryptol> :check \(x:[8]) -> x + 1 != x
Using random testing.
passed 100 tests.
Coverage: 39.06% (100 of 256 values)
```

- `:check` takes an expression, or no arguments to check all properties in a file
- Fast and easy to check properties as you go

Proving properties

- `:check` does not give a proof
- `:prove` has the same syntax, but proves properties for all values

```
Cryptol> :check \(x:[8]) -> x + 1 != x
Using random testing.
passed 100 tests.
Coverage: 39.06% (100 of 256 values)

Cryptol> :prove \(x:[8]) -> x + 1 != x
Q.E.D.
```

galois

Counterexamples

- If a property is wrong, `:check` and `:prove` give a counterexamples

```
Cryptol> :check (\x -> x != 0x7)
Using exhaustive testing.
FAILED for the following inputs:
0x7

Cryptol> let haystack x = x != 0xdeadbeef
Cryptol> :prove haystack
haystack 0xdeadbeef = False
```

- How long would `:check` have to run to find the `haystack` counterexample?

Monomorphic properties

- Cryptol can't automatically reason about polymorphic functions

```
property plus_id_l x = 0 + x == x

Cryptol> :prove plus_id_l
Not a monomorphic type:
{a} (fin a) => [a] -> Bit
```

galois

Monomorphic properties

- Provide monomorphic type signatures

```
plus_id_l : [32] -> Bit
property plus_id_l x = 0 + x == x
```

- Increase assurance by checking at multiple types

```
plus_id_l_1   : [1]   -> Bit
plus_id_l_8   : [8]   -> Bit
plus_id_l_32  : [32]  -> Bit
plus_id_l_128 : [128] -> Bit
property plus_id_l_1   x = plus_id_l x
property plus_id_l_8   x = plus_id_l x
property plus_id_l_32  x = plus_id_l x
property plus_id_l_128 x = plus_id_l x
```

Fun with matrices

- Cryptol doesn't know about matrix math, but it's easy to implement

```
mmult : {a, b, c, w} (fin a, fin b, fin w) =>
    [a][b][w] -> [b][c][w] -> [a][c][w]
mmult xss yss = [ [ sum (col * row) | col <- transpose yss ]
                  | row <- xss ]


sum : {a,n} (Arith a, fin n) => [n]a -> a
sum xs = sums!0
  where sums = [zero] # [ x + y | x <- xs | y <- sums ]
// 3x3 identity matrix
mi = [[1,0,0],
      [0,1,0],
      [0,0,1]]
```

Fun with SAT solvers

- Do you remember how to invert matrices?
- Let's use a SAT solver:

```
ma:[3][3][72]
ma = [[4,2,3],[8,5,2],[5,8,9]]
```

does there exist a matrix ***x*** *such that*

```
mmult ma x == mi ?
```

```
:sat \x -> mmult ma x == mi
```

Lab time

- Chapter 5: High-Assurance Programming
  - Example properties and intro to random testing, automated proving, and satisfiability checking
  - Try to make it through Exercises 1-9, 12, 14, 15, 17-19
- Reconvene for ZUC demo and closing discussion at 11:15am

ZUC demo

- ZUC: stream cipher in GSM standards
- Version 1.4 bug fixed in 1.5: find with `:prove`
- Detailed writeup at https://galois.com/blog/2011/06/zuc-in-cryptol/ (Cryptol 1)

Open source

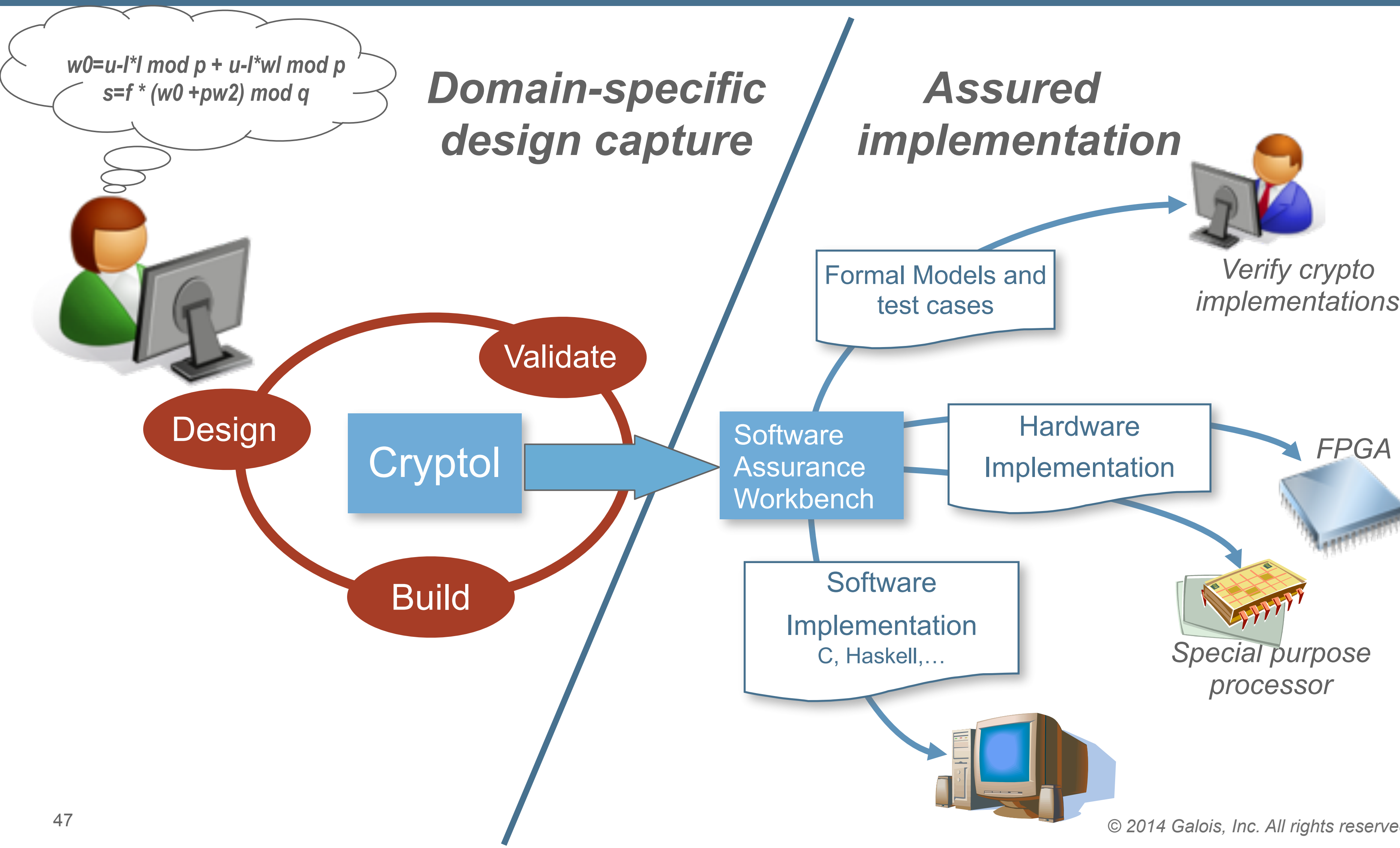
- Homepage
  - www.cryptol.net
- GitHub
  - github.com/GaloisInc/cryptol
- Mailing List
  - http://community.galois.com/mailman/listinfo/cryptol-users
- Community Contributions
  - /examples/contrib
  - 8 pull requests and counting from folks outside Galois

Bonus slides

- Software Assurance Workbench (SAW)
- Merge sort
- Proof of sorting property

galois

One specification – many uses



|galois|

Example: merge sort

```
mergeSort : {a, n} (fin n, Cmp a) => [n]a -> [n]a
mergeSort xs = fromList (mergeSortList (toList xs))

mergeSortList : {a} (Cmp a) => List a -> List a
mergeSortList txs = if isEmpty txs || isSingleton txs
                    then txs
                    else merge (mergeSortList left)
                               (mergeSortList right)
    where
        (left, right) = splitList txs

merge : {a} (Cmp a) => List a -> List a -> List a
merge xs ys = if isEmpty xs then ys
              |  isEmpty ys then xs
             else if listHead xs <= listHead ys
                  then take`{1}xs # merge (tail xs) ys
                  else take`{1}ys # merge xs (tail ys)
```

List API

Idea - take a finite list, transform it into an infinite stream of tuples (ValidBit, element)

```
type Cell a = (Bit, a)
type List a = [inf](Cell a)

toList : {n,a} (fin n) => [n]a -> List a
toList xs = [ (True, x) | x <- xs]
            # repeat (False, zero)

fromList : {n, a} (fin n) => List a -> [n]a
fromList txs = take [ x | (_,x) <- txs ]

splitList : {a} List a -> (List a, List a)
splitList xs = (lefts, rights) where
    pairs = split`{each=2} xs
    lefts =  [ left  | [left, _]  <- pairs]
    rights = [ right | [_, right] <- pairs]
```

```
listHead : {a} List a -> a
listHead txs = (txs@0).2

isEmpty : {a} List a -> Bit
isEmpty ([ (isValid,_) ] # _) =
        ~isValid

isSingleton : {a} List a -> Bit
isSingleton xs = isEmpty
                 (drop`{1}xs)
```

Unit testing

```
mergeSortOK =
      (mergeSort []                 == [])
   && (mergeSort [1, 1]             == [1, 1])
   && (mergeSort [1, 0, 3]          == [0, 1, 3])
   && (mergeSort [100, 99 .. 0]     == [0 .. 100])
   && (mergeSort [1, 3, 1, 1, 4, 5] == [1, 1, 1, 3, 4, 5])
```

But we'd like to do better!

Property examples

- Examples of different kinds of properties
  - The algorithm works correctly
  - The function defined is associative and commutative
  - Value returned is the minimum
  - For all values of key and plain-text, encryption followed by the decryption using the same key returns the plain-text
  - In Cryptol:

```
property encDec key pt = dec (key, enc(key, pt)) == pt
```

Property name

“for all” variables

Cryptol expression

Boolean property

galois

Proof obligations

- To prove sorting correct, we need to show
  - Output is in non-decreasing order
  - Output is a permutation of the input
- Strategy:
  - Define these as “predicates” in Cryptol
  - Write a property to capture correctness
- Example: recognizing non-decreasing sequences:

```
nonDecreasing : {a, b} (fin a, fin b) => [a][b] -> Bit
nonDecreasing xs = pairComps == ~zero
  where pairComps = [ x <= x' | x  <- [0] # xs
                              | x' <- xs
                    ]
```

- Recognizing permutations is a bit more complicated

Putting it together

- Express correctness by combining the two

```
property mergeSortIsCorrect =
     nonDecreasing(ys) && isPermutationOf(xs, ys)
  where ys = mergeSort(xs)
```

- Property declarations are first class citizens of Cryptol
  - Coexists with the code
  - No need to learn a separate “verification” language
  - Not comments, or “documentation”; but serve as great documentation
- Properties can be quickly `:check`’ed for fast feedback
  - Or, proved automatically using SAT/SMT based technologies
  - External tool usage is all transparent to the user
- Counter-examples are priceless!