Cryptol

Specification, Implementation and Verification of High-Grade Cryptographic Applications

CACE 2008, Porto

Lee Pike
leepike@galois.com

The Cryptol team, past and present:
Sally Browning, Levent Erkok, Sigbjorn Finne, Andy Gill, Fergus Henderson, John Launchbury, Jeff Lewis, Lee Pike, John Matthews, Thomas Nordin, Mark Shields, Joel Stanley, Frank Seaton Taylor, Jim Teisher, Philip Weaver, Adam Wick
Typical DoD Cryptographic Application

KG-255 Inline Network Encryptor
The Problem

“Of the 1.3 million cryptographic devices in the U.S. inventory, 73 percent will be replaced over the next 15 years ...”

“... a severe lack of diagnostic capabilities to evaluate software products to detect unintentional vulnerabilities and maliciously implanted functionality in a timely and cost-effective manner.”

“Software evaluation is very manpower intensive and doesn’t scale.”

Daniel G. Wolf, Director
Information Assurance Directorate
National Security Agency
January, 2006
Overview for the Talk

**Cryptol Project Vision:** To reduce the cost (in both time and money) of developing and certifying cryptographic applications

**Cryptol Specification**
- A Domain Specific Specification Language
  - Precise, Declarative Semantics
  - High level design exploration

**Automated Synthesis down to FPGA**
- Algebraic rewrite-based compilation
- Traceability back to specification

**Automated Verification**
- AIG-based Equivalence Checking
- SAT Solver technology
Requirements for a Crypto DSL

- Use a high-level domain-specific language for design capture, but also be executable
- Use Cryptol specifications to guide and document implementations
  - And even generate them
- Be neutral to implementation platform
  - Cannot make assumptions about underlying platform
  - Don’t bake in Von Neumann assumptions
- The language should be high-level, yet detailed
  - Can talk about the bits, but in a platform-independent way
## Cryptol: Specify interfaces unambiguously

**From the Advanced Encryption Standard definition†**

### 3.1 Inputs and Outputs

The input and output for the AES algorithm each consist of sequences of 128 bits (digits with values of 0 or 1). These sequences will sometimes be referred to as blocks and the number of bits they contain will be referred to as their length. The Cipher Key for the AES algorithm is a sequence of 128, 192 or 256 bits. Other input, output and Cipher Key lengths are not permitted by this standard.

### Cryptol

\[
\text{blockEncrypt : } \{ k \} \ (k \geq 2, \ 4 \geq k) \Rightarrow ([128], \ [64 \times k]) \rightarrow [128]
\]

- For all \( k \) between 2 and 4
- First input is a sequence of 128 bits
- Second input is a sequence of 128, 192, or 256 bits
- Output is a sequence of 128 bits

Cryptol: Express data flow details

**encrypt128** : ([4][32],[4][4][8]) → [4][4][8];
encrypt128 (keyAsWords, pt) = finalState where
{
  roundKeys = KeyExpansion128(keyAsWords);
  initialState = first(roundKeys) ^ pt;

  rounds = [ initialState ] #
  || nextState (prev_state, roundKey, round)
  || round <- [1..10]
  || prev_state <- rounds
  || roundKey <- drop (1, roundKeys)
  ||

  finalState = last(rounds);
};

**nextState** : ([4][4][8],[4][4][8],[4]) → [4][4][8];
nextState (state, key, round) =
  if round < 10
  then key ^ MixColumns(ShiftRows(SubBytes(state)))
  else key ^ ShiftRows(SubBytes(state));
Developing with Cryptol: Expected Benefits

- A clear and unambiguous model of the crypto algorithm
  Particularly details like bit-order and endian-ness
- A high-level notation that is familiar to domain experts
  Will simplify expression, inspection, and re-use
- A specification that can be validated
  Can validate any part of algorithm, not just end-to-end
- Re-usable models
  - Capture design and validate once, re-use many times
  - Specification for both hardware and software implementations
- Specifications that can be easily re-factored
  E.g., easy to separate mode from algorithm
Developing with Cryptol: Lessons Learned from GD

- Was a clear and unambiguous model of the crypto algorithm developed?
  - Early results suggested that use of Cryptol reduced development time by ~20%
  - A Cryptol spec replaces substantial English documentation

- Did the Cryptol language feel familiar to domain experts?
  - The syntax wasn’t as familiar/natural as we’d hoped: “stream model” is sometimes awkward
  - User feedback has guided the continued development of the language

- Can the specification be validated?
  - Yes, testing using published test vectors, but ignores the “waveform”: code surrounding the algorithm is more likely a source of faults or vulnerabilities
  - We now have a suite of tools that add equivalence checking capabilities.

- Were the models reusable and/or refactorable?
  - GD is currently developing a follow-on project using a similar development process and reusing applicable algorithms.
Overview for the Talk

Cryptol Project Vision: To reduce the cost (in both time and money) of developing and certifying cryptographic applications

Domain Specific Specification Language
• Precise, Declarative Semantics
• High level design exploration

Automated Synthesis down to FPGA
• Algebraic rewrite-based compilation
• Traceability back to specification

Verification
• AIG-based Equivalence Checking
• SAT Solver
Why implement crypto algorithms on FPGAs?

- For high-grade crypto, commodity hardware is not trusted. Thus, traditional software approaches are irrelevant.

- There is trust when the evaluators can see as much of the solution as possible. Thus, hardware-only solutions are the norm.

- Custom crypto chips are increasingly common:
  - But they are complex, and there is no standard.
  - FPGAs are more flexible, and built on simple building blocks.

- Natural match between Cryptography and FPGAs:
  - Manipulation of arbitrary length bit sequences.
  - Highly parallel stream processing.
Why use Cryptol to produce FPGAs?

Because Cryptol naturally models hardware

- It is purely functional
  - A Cryptol function maps inputs to outputs, with no side effects
  - Combinational hardware circuits are also pure!
  - Can still model state-holding elements within a stream
- Fixed-length sequences naturally model hardware bit-vectors
- It has a synchronous stream style
  - Sequences can be mapped over space or time
  - There is an implicit clock signal: one clock period per element in sequential sequences
  - The user can explore space-time tradeoffs without significantly changing the Cryptol source
- Declarative programming exploits parallelism
  - A set of independent declarations are naturally parallel
    - e.g., (x, y, z) where \( \{ x = 2*a; y = f(b); z = g(c); \} \)
- Sequentialization in Cryptol comes only from data-dependency — just like hardware
Simulation and Model Checking built into the Cryptol Toolchain

- Cryptol source code
  - Parse
    - Parse Tree
  - Type Check
    - Cryptol IR
  - Generate Code
    - SPIR
  - Optimization Rewrite
    - LLSPIR
  - Pretty Print
    - C
      - VHDL
  - Synthesis
    - Netlist

Simulation Capability
- Bit Model Interpreter
- Optimized Interpreter
- C backend
- SPIR Interpreter

Model Checking Capability
- Model Generator
- Equivalence Checker
Example: Compiling Cryptol to VHDL

- **Input** is an infinite stream
- **Output** is an infinite stream
  - Each element of the output is a sum of three elements from the input
  - The three values to be summed are “produced” in parallel according to the recipe above

If input is \{1 2 3 4 \ldots\}.

Then output will be \{1+2+3 2+3+4 3+4+5 4+5+6 \ldots\} \equiv \{6 9 12 15 \ldots\}.
Step 1: Compile Cryptol source to SPIR

- Signal Processor Intermediate Representation
- Not synthesizable
  e.g. drop, which looks back in time

Cryptol

```
sum3 : {a} [inf][a] -> [inf][a];
sum3 xs = [|| x0 + x1 + x2 || x0 ← xs
|| x1 ← drop (1, xs)
|| x2 ← drop (2, xs)
||];
```

Cryptol

```
Cry
```

```
.SPIR
```

```
```

```
Not.synthesizable
```

```
e.g. drop, which looks back in time
```

```
Input
```

```
Output
```

```
ADD
```

```
ADD
```

```
ADD
```

```
Drop.1
```

```
Drop.2
```

```
Cry
```

```
.SPIR
```

```
```

```
Not.synthesizable
```

```
e.g. drop, which looks back in time
```

```
Input
```

```
Output
```

```
ADD
```

```
ADD
```

```
ADD
```

```
Drop.1
```

```
Drop.2
```

```
Cry
```

```
.SPIR
```

```
```

```
Not.synthesizable
```

```
e.g. drop, which looks back in time
```

```
Input
```

```
Output
```

```
ADD
```

```
ADD
```

```
ADD
```

```
Drop.1
```

```
Drop.2
```

```
Cry
```

```
.SPIR
```

```
```

```
Not.synthesizable
```

```
e.g. drop, which looks back in time
```

```
Input
```

```
Output
```

```
ADD
```

```
ADD
```

```
ADD
```

```
Drop.1
```

```
Drop.2
```

```
Cry
```

```
.SPIR
```

```
```

```
Not.synthesizable
```

```
e.g. drop, which looks back in time
```

```
Input
```

```
Output
```

```
ADD
```

```
ADD
```

```
ADD
```

```
Drop.1
```

```
Drop.2
```

```
Cry
```

```
.SPIR
```

```
```

```
Not.synthesizable
```

```
e.g. drop, which looks back in time
```

```
Input
```

```
Output
```

```
ADD
```

```
ADD
```

```
ADD
```

```
Drop.1
```

```
Drop.2
```

```
Cry
```

```
.SPIR
```

```
```

```
Not.synthesizable
```

```
e.g. drop, which looks back in time
```

```
Input
```

```
Output
```

```
ADD
```

```
ADD
```

```
ADD
```

```
Drop.1
```

```
Drop.2
```

```
Cry
```

```
.SPIR
```

```
```

```
Not.synthesizable
```

```
e.g. drop, which looks back in time
```

```
Input
```

```
Output
```

```
ADD
```

```
ADD
```

```
ADD
```

```
Drop.1
```

```
Drop.2
```

```
Cry
```

```
.SPIR
```

```
```

```
Not.synthesizable
```

```
e.g. drop, which looks back in time
```

```
Input
```

```
Output
```

```
ADD
```

```
ADD
```

```
ADD
```

```
Drop.1
```

```
Drop.2
```

```
Cry
```

```
.SPIR
```

```
```

```
Not.synthesizable
```

```
e.g. drop, which looks back in time
```

```
Input
```

```
Output
```

```
ADD
```

```
ADD
```

```
ADD
```

```
Drop.1
```

```
Drop.2
```

```
Cry
```

```
.SPIR
```

```
```

```
Not.synthesizable
```

```
e.g. drop, which looks back in time
```

```
Input
```

```
Output
```

```
ADD
```

```
ADD
```

```
ADD
```

```
Drop.1
```

```
Drop.2
```

```
Cry
```

```
.SPIR
```

```
```

```
Not.synthesizable
```

```
e.g. drop, which looks back in time
```

```
Input
```

```
Output
```

```
ADD
```

```
ADD
```

```
ADD
```

```
Drop.1
```

```
Drop.2
```

```
Cry
```
Step 2: Rewrite SPIR as LLSPIR

Cryptol

\[ \text{sum3 : \{a\} [inf][a] -> [inf][a];} \]

\[ \text{sum3 \ xs = \| x0 + x1 + x2 \| x0 \leftarrow x0} \]

\[ \| x1 \leftarrow \text{drop (1, xs)} \]

\[ \| x2 \leftarrow \text{drop (2, xs)} \]

\[ \]|; \]

- **Low-Level SPIR**
- **Latency introduced**
  - Here, two clock cycles before first output value is available
  - Output is “undefined” during those two cycles
- **Synthesizable**
Step 3: Pretty-print LLSPIR as VHDL

entity test_circuit_entity is
  port (n1_xs_sum3 : in std_logic_vector(15 downto 0);
    n32_outputPort : out std_logic_vector(15 downto 0);
    n0_restart : in std_logic_onebit;
    clk : in std_logic_onebit);
  -- no pin assignments
end entity test_circuit_entity;

architecture test_circuit_behaviour of test_circuit_entity is
  signal n25_xs_sum3 : std_logic_vector(15 downto 0);
  signal n26_ADD : std_logic_vector(15 downto 0);
  signal n28_ADD : std_logic_vector(15 downto 0);
  signal n29_ADD : std_logic_vector(15 downto 0);
  begin
    delay_n1_xs_sum3_n25_xs_sum3_1: entity crypltol.delaycell
      generic map(n => 1, w => 16)
      port map(clk => clk, i => n1_xs_sum3, o => n25_xs_sum3);

    n26_ADD <= n25_xs_sum3 + n1_xs_sum3;

    delay_n26_ADD_n28_ADD_1: entity cryptol.delaycell
      generic map(n => 1, w => 16)
      port map(clk => clk, i => n26_ADD, o => n28_ADD);

    n29_ADD <= n28_ADD + n1_xs_sum3;

    n32_outputPort <= n29_ADD;
  end architecture test_circuit_behaviour;
Stepwise refinement to implementation

Key:
- Galois tools
- FPGA Vendor tools
- Evaluation/Certification evidence
- Data files
- Input to tool
- Feedback to designer

Reference Specification

Symbolic Interpreter

Reference Model

Equivalence Checker

Equivalence Evidence

Target Specification

Crypto Developer

Symbolic Evaluator

Target Model

Equivalence Evidence

Cryptol FPGA Compiler

VHDL

Synthesis

Symbolic Netlist Eval

Netlist Model

Place and Route

Bitfile
Case Study at Rockwell-Collins
High Speed Encryptor Project

Key
- Galois tools
- Third party tools
- Data files
- Evaluation/Certification evidence
- Input to tool
- Feedback to designer

©2007-8 Galois, Inc. CACE 2008
Cryptol FPGA: Some Lessons Learned

- High-level exploration of the design space yields huge benefits
  A skilled Cryptol developer can do in 1 week what might take an FPGA engineer month(s) to do
- Detailed implementation refinements can be effectively modeled in the high-level specification, e.g. pipelining
- System integration issues are important, e.g., space vs. time.
- We went after usability but got performance as well
  Heavily pipelined AES written in Cryptol is competitive with commercial cores (top grade cores run 40-50 Gb/s)
Overview for the Talk

**Cryptol Project Vision:** To reduce the cost (in both time and money) of developing and certifying cryptographic applications

**Cryptol Specification**
- Precise, Declarative Semantics
- High level design exploration

**Automated Synthesis down to FPGA**
- Algebraic rewrite-based compilation
- Traceability back to specification

**Verification**
- AIG-based Equivalence Checking
- SAT Solver
Verification Strategy

- Show function equivalence with the reference specification throughout the tool chain

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Galois tools</td>
<td>Galois tools</td>
</tr>
<tr>
<td>Xilinx tools</td>
<td>Xilinx tools</td>
</tr>
<tr>
<td>Cryptol files</td>
<td>Cryptol files</td>
</tr>
<tr>
<td>Formal Models</td>
<td>Formal Models</td>
</tr>
<tr>
<td>Data files</td>
<td>Data files</td>
</tr>
<tr>
<td>Input to tool</td>
<td>Input to tool</td>
</tr>
<tr>
<td>Input to designer</td>
<td>Input to designer</td>
</tr>
</tbody>
</table>
The Problem with Equivalence Checking...

- But equivalence checking is hard!
  - SAT is NP-complete
  - However, equivalence checking “similar” boolean functions is easier

- And we have sequential state-holding circuits (not boolean functions)
  - Use Symbolic Evaluation to reduce the sequential circuit to a combinational circuit
  - Limits us to specifications/implementations that are finite
    - Not polymorphic!
    - Not over infinite streams!
    - This is generally true for block cipher implementations
Example: Are These Circuits Equivalent?

Circuit #1

Circuit #2
Equivalence Checking Algorithm

- **Model Generation**
  - Create And-Invertor Graphs (AIG) as formal models of the two circuits
  - Identify some equivalent nodes while creating the AIG of the circuits

- **Random Simulation**
  - Send random vectors through the two circuits, collecting pairs of candidate equivalent nodes

- **SAT Sweeping**
  - Use candidate equivalent nodes to guide SAT searches, merging AIG nodes as indicated, thereby reducing the complexity of future SAT searches

Note: In the forthcoming example, we will show random simulation *before* model generation to facilitate the presentation, but the tool is implemented as shown above.
Random Simulation

1. Random Vector: \{a=True, b=True, c=True\}

<table>
<thead>
<tr>
<th>Circuit #1</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Circuit #1 Diagram" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Circuit #2</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image2" alt="Circuit #2 Diagram" /></td>
</tr>
</tbody>
</table>
Random Simulation

2. Random Vector: \{a=\text{False}, b=\text{False}, c=\text{False}\}

**Circuit #1**

**Circuit #2**
Random Simulation

3. Random Vector: \{a=\text{False}, \ b=\text{False}, \ c=\text{True}\}

Circuit #1

Circuit #2
Random Simulation

4. Random Vector: \{a=\text{True}, \ b=\text{True}, \ c=\text{False}\}

**Circuit #1**

**Circuit #2**
Exhaustive Simulation?

Random Vector: \{a=True, b=False, c=True\} 
equivalent to \{a=False, b=True, c=True\}

Circuit #1

Circuit #2

Random Vector: \{a=True, b=False, c=False\} 
equivalent to \{a=False, b=True, c=False\}
Model Generation

- AIGs are a “standard”

- Simple data structure used to represent combinational circuits

- DAG where nodes are ANDs and edges are True/False, possibly inverted

- Operations are fast (add node, merge nodes)
  - Structural hashing makes it fast

- Merging a pair of equivalent nodes can cause other nodes to be merged automatically, without need for a SAT proof
Model Generation

Key to Circuit Symbols
- \( \bigodot \) AND function, output numbered
- \( \bigodot \) NAND function, output numbered
- \( \bigoplus \) OR function, output numbered
- \( \bigcirc \) Invertor (NOT function)

Recall that \( \bigoplus \) is equivalent to \( \bigcirc \)

Circuit #1

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{c}
\end{array} \quad \begin{array}{c}
\bigoplus \\
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot
\end{array} \quad \begin{array}{c}
\text{a} \\
\text{b} \\
\text{c}
\end{array}
\]

AIG for Circuit #1

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{c}
\end{array} \quad \begin{array}{c}
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot
\end{array} \quad \begin{array}{c}
\text{x}
\end{array}
\]

Key to AIG symbols
- \( \bigodot \) an AND node, numbered as in the circuit
- \( \bigodot \) input to or output from a node
- \( \bigodot \) inverted input or output

Circuit #2

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{c}
\end{array} \quad \begin{array}{c}
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot
\end{array} \quad \begin{array}{c}
\text{a} \\
\text{b} \\
\text{c}
\end{array}
\]

AIG for Circuit #2

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{c}
\end{array} \quad \begin{array}{c}
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot \\
\bigodot
\end{array} \quad \begin{array}{c}
\text{y}
\end{array}
\]
Identifying shared structure

- Node 0 and 1 have equivalent inputs; collapse them into one.
Identifying shared structure

- Combine the \( c \) inputs
SAT Sweeping

- Use SAT to prove whether or not the candidate equivalent nodes, from the random simulation phase, are equivalent.
- The candidate equivalent nodes are used as cut points.
- Generate SAT problems that are solved from inputs to outputs using the candidate equivalent nodes as a guide [3].

SAT Instances

- Create one SAT instance for one (or more) pair of candidate equivalent nodes
- Each SAT search can result in the merger of equivalent AIG nodes, reducing the complexity of the AIG
SAT Sweeping

- Use the AIG forms of the circuits and the candidate equivalent nodes identified by random simulation.
SAT Sweeping: Is 3 ≡ 5?

- Determine if the nodes have the same value for all inputs.

```
+---+---+---+---+---+---+---+---+
| 0 | 4 | 6 | 7 | x |
| a | b |
```

```
+---+---+---+---+---+---+---+---+
| 2 | 6 |
| 3 | 5 |
| 7 | 8 |
```

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
```

**SAT Solver says YES!**
SAT Sweeping: Merge 3 and 5
SAT Sweeping: Is 2 ≡ 6?

- Frame the nodes as a miter circuit and trace the circuits back to the inputs.

**SAT Solver says YES!**
SAT Sweeping: Merge 2 and 6
SAT Sweeping: 7 structurally hashes to 8
SAT Sweeping: \( x \equiv y \therefore \text{Circuit \#1} \equiv \text{Circuit \#2} \)
Equivalence Checking: Some Lessons Learned

- Technique is very useful in practice
  - Equivalence Checking of various versions of AES and DES against both reference models and models of LLSPIR take < 5 minutes (most take < 30 seconds)
  - Models typically in range of half a million nodes

- However, it has its limitations
  - The Hash Function MD5 on 2 bits takes nearly 10 minutes...
  - The block cipher RC6, takes too long (32-bit multiply)
  - Only works on core ciphers - not modes (finite input/output)
  - The equivalence checker typically yields an answer right away, or it takes substantially longer - there isn’t much of a middle ground
  - Current research at Galois is exploring mechanical theorem-proving (Isabelle) + decision procedures (satisfiability modulo theories) to verify high-level correspondence
Reprise of the Talk

**Cryptol Project Vision:** To reduce the cost (in both time and money) of developing and certifying cryptographic applications

- **Domain Specific Specification Language**
  - Precise, Declarative Semantics
  - High level design exploration

- **Automated Synthesis down to FPGA**
  - Algebraic rewrite-based compilation
  - Traceability back to specification

- **Verification**
  - AIG-based Equivalence Checking
  - SAT Solver
Current Research Challenges

- Cryptol Language
  - Extension into other data-flow domains (like DSP)
  - Other code generators (like ASIC, GPU, proprietary microcode)

- FPGA Generation
  - Improved type constraint simplifications and error messages
  - Communicating back to user why a Cryptol spec is not synthesizeable
  - Synthesizing across multiple FPGAs
  - Supporting vendor optimizations in Cryptol

- Equivalence checking
  - end-to-end Xilinx tool flow
  - ECC and PKI algorithms
  - Large bit vector sizes, deep semantic optimizations
  - Equivalence checking for modes
  - Unbounded equivalence checking in general
Questions?

www.cryptol.net

Academic Licenses Available
Contact: cryptol@cryptol.net